

Cẩm nang lập trình thi đấu

Antti Laaksonen

Bản dịch tiếng Việt: cuom1999
lephuocdin99@gmail.com

Bản nháp 5 tháng 6, 2026

Mục lục

Lời nói đầu	ix
I Kỹ thuật cơ bản	1
1 Giới thiệu	3
1.1 Ngôn ngữ lập trình	3
1.2 Nhập và xuất	4
1.3 Làm việc với số	6
1.4 Rút gọn mã	8
1.5 Toán học	10
1.6 Cuộc thi và tài nguyên	15
2 Độ phức tạp thời gian	19
2.1 Quy tắc tính toán	19
2.2 Lớp độ phức tạp	22
2.3 Ước lượng hiệu quả	23
2.4 Tổng đoạn con lớn nhất	24
3 Sắp xếp	27
3.1 Lý thuyết sắp xếp	27
3.2 Sắp xếp trong C++	32
3.3 Tìm kiếm nhị phân	34
4 Cấu trúc dữ liệu	39
4.1 Mảng động	39
4.2 Cấu trúc tập hợp	41
4.3 Cấu trúc ánh xạ	42
4.4 Iterator và đoạn	43
4.5 Cấu trúc khác	46
4.6 So sánh với sắp xếp	50
5 Tìm kiếm toàn bộ (Complete Search)	53
5.1 Sinh tập con	53
5.2 Sinh hoán vị	55
5.3 Backtracking	56
5.4 Cắt tỉa tìm kiếm	58

5.5	Chia đôi tìm kiếm (Meet in the middle)	60
6	Thuật toán tham lam (Greedy Algorithms)	63
6.1	Bài toán đồng xu	63
6.2	Lập lịch	64
6.3	Công việc và hạn chót	66
6.4	Tối thiểu hóa tổng	67
6.5	Nén dữ liệu	68
7	Quy hoạch động (Dynamic Programming)	71
7.1	Bài toán đồng xu	71
7.2	Dãy con tăng dài nhất (Longest increasing subsequence) . .	76
7.3	Đường đi trong lưới	77
7.4	Các bài toán ba lô (Knapsack)	78
7.5	Khoảng cách chỉnh sửa (Edit distance)	80
7.6	Đếm cách lát gạch	81
8	Phân tích khẩu hao	83
8.1	Phương pháp hai con trỏ	83
8.2	Phần tử nhỏ hơn gần nhất	85
8.3	Giá trị nhỏ nhất của cửa sổ trượt	87
9	Truy vấn đoạn (Range Queries)	89
9.1	Truy vấn trên mảng tĩnh	90
9.2	Cây chỉ số nhị phân	92
9.3	Cây đoạn	95
9.4	Các kỹ thuật bổ sung	99
10	Thao tác bit	101
10.1	Biểu diễn bit	101
10.2	Các phép toán bit	102
10.3	Biểu diễn tập hợp	104
10.4	Tối ưu hóa bằng bit	106
10.5	Quy hoạch động	108
II	Thuật toán đồ thị	113
11	Cơ bản về đồ thị	115
11.1	Thuật ngữ đồ thị	115
11.2	Biểu diễn đồ thị	119
12	Duyệt đồ thị (Graph Traversal)	123
12.1	Tìm kiếm theo chiều sâu	123
12.2	Tìm kiếm theo chiều rộng	125
12.3	Ứng dụng	127

13 Đường đi ngắn nhất	131
13.1 Thuật toán Bellman–Ford	131
13.2 Thuật toán Dijkstra	134
13.3 Thuật toán Floyd–Warshall	137
14 Thuật toán trên cây	141
14.1 Duyệt cây	142
14.2 Đường kính	143
14.3 Mọi đường đi dài nhất	145
14.4 Cây nhị phân	147
15 Cây khung	149
15.1 Thuật toán Kruskal	150
15.2 Cấu trúc union-find	153
15.3 Thuật toán Prim	155
16 Đồ thị có hướng	157
16.1 Sắp xếp topo	157
16.2 Quy hoạch động	159
16.3 Đường đi kế tiếp	162
16.4 Phát hiện chu trình	163
17 Liên thông mạnh	165
17.1 Thuật toán Kosaraju	166
17.2 Bài toán 2SAT	168
18 Truy vấn trên cây	171
18.1 Tìm tổ tiên	171
18.2 Cây con và đường đi	172
18.3 Tổ tiên chung thấp nhất	175
18.4 Thuật toán offline	178
19 Đường đi và chu trình	183
19.1 Đường đi Euler	183
19.2 Đường đi Hamilton	187
19.3 Dãy De Bruijn	188
19.4 Hành trình quân mã	189
20 Luồng và lát cắt	191
20.1 Thuật toán Ford–Fulkerson	192
20.2 Các đường đi rời nhau	196
20.3 Ghép cặp cực đại	198
20.4 Phủ đường đi	201

III	Chủ đề nâng cao	205
21	Lý thuyết số	207
21.1	Số nguyên tố và ước	207
21.2	Số học modulo	211
21.3	Giải phương trình	214
21.4	Các kết quả khác	215
22	Tổ hợp	217
22.1	Hệ số nhị thức	218
22.2	Số Catalan	220
22.3	Nguyên lý bù trừ	222
22.4	Bổ đề Burnside	224
22.5	Công thức Cayley	225
23	Ma trận	227
23.1	Phép toán	227
23.2	Truy hồi tuyến tính	230
23.3	Đồ thị và ma trận	232
24	Xác suất	235
24.1	Tính toán	235
24.2	Sự kiện	236
24.3	Biến ngẫu nhiên	238
24.4	Xích Markov	240
24.5	Thuật toán ngẫu nhiên	241
25	Lý thuyết trò chơi	245
25.1	Trạng thái trò chơi	245
25.2	Trò chơi nim	247
25.3	Định lý Sprague–Grundy	248
26	Thuật toán chuỗi	253
26.1	Thuật ngữ chuỗi	253
26.2	Cấu trúc trie	254
26.3	Băm chuỗi	255
26.4	Thuật toán Z	258
27	Thuật toán căn bậc hai	261
27.1	Kết hợp thuật toán	262
27.2	Phân hoạch số nguyên	264
27.3	Thuật toán Mo	265
28	Cây phân đoạn nâng cao (Segment Trees Revisited)	267
28.1	Lan truyền lười (Lazy propagation)	268
28.2	Cây động	271
28.3	Cấu trúc dữ liệu	273
28.4	Hai chiều	274

29 Hình học	275
29.1 Số phức	276
29.2 Điểm và đường thẳng	278
29.3 Diện tích đa giác	281
29.4 Hàm khoảng cách	283
30 Thuật toán đường quét (Sweep Line Algorithms)	287
30.1 Điểm giao	288
30.2 Bài toán cặp gần nhất	289
30.3 Bài toán bao lồi	290
Tài liệu tham khảo	293
Chỉ mục	299

Lời nói đầu

Mục đích của cuốn sách này là đem lại cho bạn một phần giới thiệu đầy đủ về lập trình thi đấu. Sách giả định rằng bạn đã biết những kiến thức cơ bản về lập trình, nhưng không yêu cầu kinh nghiệm trước đó về lập trình thi đấu.

Cuốn sách đặc biệt hướng tới học sinh, sinh viên muốn học thuật toán và có thể tham gia Olympic Tin học Quốc tế (IOI) hoặc Kỳ thi Lập trình Sinh viên Quốc tế (ICPC). Dĩ nhiên, sách cũng phù hợp với bất kỳ ai quan tâm đến lập trình thi đấu.

Cần nhiều thời gian để trở thành một lập trình viên thi đấu giỏi, nhưng đó cũng là cơ hội để học được rất nhiều điều. Bạn có thể tin rằng mình sẽ có được hiểu biết tổng quát tốt về thuật toán nếu dành thời gian đọc sách, giải bài tập và tham gia các kỳ thi.

Cuốn sách vẫn đang được phát triển liên tục. Bạn luôn có thể gửi phản hồi về sách tới ahslaaks@cs.helsinki.fi.

Helsinki, August 2019
Antti Laaksonen

Phần I

Kỹ thuật cơ bản

Chương 1

Giới thiệu

Lập trình thi đấu (Competitive Programming) kết hợp hai chủ đề: (1) thiết kế thuật toán (design of algorithms) và (2) cài đặt thuật toán (implementation of algorithms).

Thiết kế thuật toán (design of algorithms) bao gồm giải quyết bài toán và tư duy toán học. Cần có kỹ năng phân tích bài toán và giải chúng một cách sáng tạo. Một thuật toán để giải một bài toán phải vừa đúng vừa hiệu quả, và cốt lõi của bài toán thường là nghĩ ra một thuật toán hiệu quả.

Kiến thức lý thuyết về thuật toán rất quan trọng đối với người lập trình thi đấu. Thông thường, lời giải cho một bài toán là sự kết hợp giữa các kỹ thuật quen thuộc và những quan sát mới. Các kỹ thuật xuất hiện trong lập trình thi đấu cũng tạo thành nền tảng cho nghiên cứu khoa học về thuật toán.

Cài đặt thuật toán (implementation of algorithms) đòi hỏi kỹ năng lập trình tốt. Trong lập trình thi đấu, lời giải được chấm bằng cách kiểm thử một thuật toán đã cài đặt trên một tập các bộ test. Vì vậy, ý tưởng của thuật toán đúng là chưa đủ; phần cài đặt cũng phải đúng.

Phong cách viết mã tốt trong các kỳ thi là trực tiếp và ngắn gọn. Chương trình cần được viết nhanh, vì thời gian không nhiều. Khác với kỹ nghệ phần mềm truyền thống, các chương trình thường ngắn (thường nhiều nhất chỉ vài trăm dòng mã), và không cần được bảo trì sau cuộc thi.

1.1 Ngôn ngữ lập trình

Hiện nay, các ngôn ngữ lập trình phổ biến nhất được dùng trong các cuộc thi là C++, Python và Java. Ví dụ, trong Google Code Jam 2017, trong số 3,000 thí sinh tốt nhất, 79 % dùng C++, 16 % dùng Python và 8 % dùng Java [29]. Một số thí sinh cũng dùng nhiều ngôn ngữ.

Nhiều người cho rằng C++ là lựa chọn tốt nhất cho một lập trình viên thi đấu, và C++ hầu như luôn có mặt trong các hệ thống thi. Lợi ích của việc dùng C++ là đây là một ngôn ngữ rất hiệu quả và thư viện chuẩn của nó chứa một tập lớn các cấu trúc dữ liệu và thuật toán.

Mặt khác, việc thành thạo nhiều ngôn ngữ và hiểu điểm mạnh của chúng

cũng rất hữu ích. Ví dụ, nếu bài toán cần số nguyên lớn, Python có thể là một lựa chọn tốt, vì nó có sẵn các phép toán để tính toán với số nguyên lớn. Dù vậy, hầu hết bài toán trong các cuộc thi lập trình được đặt ra sao cho việc dùng một ngôn ngữ lập trình cụ thể không tạo ra lợi thế không công bằng.

Tất cả chương trình ví dụ trong cuốn sách này được viết bằng C++, và các cấu trúc dữ liệu cùng thuật toán của thư viện chuẩn thường xuyên được sử dụng. Các chương trình tuân theo chuẩn C++11, chuẩn này hiện có thể dùng trong hầu hết các cuộc thi. Nếu bạn chưa biết lập trình bằng C++, bây giờ là thời điểm tốt để bắt đầu học.

Mẫu mã C++

Một mẫu mã C++ điển hình cho lập trình thi đấu có dạng như sau:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // solution comes here
}
```

Dòng `#include` ở đầu mã là một tính năng của trình biên dịch `g++` cho phép chúng ta đưa vào toàn bộ thư viện chuẩn. Vì vậy, không cần đưa riêng các thư viện như `iostream`, `vector` và `algorithm`; thay vào đó, chúng sẽ tự động sẵn dùng.

Dòng `using` khai báo rằng các lớp và hàm của thư viện chuẩn có thể được dùng trực tiếp trong mã. Nếu không có dòng `using`, chẳng hạn ta sẽ phải viết `std::cout`, nhưng giờ chỉ cần viết `cout`.

Mã có thể được biên dịch bằng lệnh sau:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Lệnh này tạo ra một tệp nhị phân `test` từ mã nguồn `test.cpp`. Trình biên dịch tuân theo chuẩn C++11 (`-std=c++11`), tối ưu mã (`-O2`) và hiển thị cảnh báo về các lỗi có thể xảy ra (`-Wall`).

1.2 Nhập và xuất

Trong hầu hết các cuộc thi, các luồng chuẩn được dùng để đọc input và ghi output. Trong C++, các luồng chuẩn là `cin` cho input và `cout` cho output. Ngoài ra, các hàm C `scanf` và `printf` cũng có thể được dùng.

Input của chương trình thường gồm các số và chuỗi được phân tách bằng dấu cách và ký tự xuống dòng. Chúng có thể được đọc từ luồng `cin` như sau:

```
int a, b;
```

```
string x;  
cin >> a >> b >> x;
```

Kiểu mã này luôn hoạt động, miễn là giữa mỗi phần tử trong input có ít nhất một dấu cách hoặc ký tự xuống dòng. Ví dụ, đoạn mã trên có thể đọc cả hai input sau:

```
123 456 monkey
```

```
123    456  
monkey
```

Luồng `cout` được dùng để ghi output như sau:

```
int a = 123, b = 456;  
string x = "monkey";  
cout << a << " " << b << " " << x << "\n";
```

Nhập và xuất đôi khi là nút thắt cổ chai của chương trình. Các dòng sau ở đầu mã làm cho nhập và xuất hiệu quả hơn:

```
ios::sync_with_stdio(0);  
cin.tie(0);
```

Lưu ý rằng ký tự xuống dòng `"\n"` chạy nhanh hơn `endl`, vì `endl` luôn gây ra một thao tác flush.

Các hàm C `scanf` và `printf` là một lựa chọn thay thế cho các luồng chuẩn của C++. Chúng thường nhanh hơn một chút, nhưng cũng khó dùng hơn. Đoạn mã sau đọc hai số nguyên từ input:

```
int a, b;  
scanf("%d %d", &a, &b);
```

Đoạn mã sau in hai số nguyên:

```
int a = 123, b = 456;  
printf("%d %d\n", a, b);
```

Đôi khi chương trình cần đọc cả một dòng từ input, có thể chứa dấu cách. Việc này có thể được thực hiện bằng cách dùng hàm `getline`:

```
string s;  
getline(cin, s);
```

Nếu lượng dữ liệu không biết trước, vòng lặp sau rất hữu ích:

```
while (cin >> x) {  
    // code  
}
```

Vòng lặp này đọc lần lượt các phần tử từ input, cho đến khi không còn dữ liệu nào trong input.

Trong một số hệ thống thi, các tệp được dùng cho input và output. Một cách đơn giản để xử lý là viết mã như bình thường bằng các luồng chuẩn, nhưng thêm các dòng sau vào đầu mã:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Sau đó, chương trình đọc input từ tệp "input.txt" và ghi output ra tệp "output.txt".

1.3 Làm việc với số

Số nguyên

Kiểu số nguyên được dùng nhiều nhất trong lập trình thi đấu là `int`, một kiểu 32 bit với miền giá trị $-2^{31} \dots 2^{31} - 1$ hay xấp xỉ $-2 \cdot 10^9 \dots 2 \cdot 10^9$. Nếu kiểu `int` không đủ, có thể dùng kiểu 64 bit `long long`. Nó có miền giá trị $-2^{63} \dots 2^{63} - 1$ hay xấp xỉ $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

Đoạn mã sau định nghĩa một biến `long long`:

```
long long x = 123456789123456789LL;
```

Hậu tố `LL` có nghĩa là kiểu của số là `long long`.

Một lỗi phổ biến khi dùng kiểu `long long` là kiểu `int` vẫn còn được dùng ở đâu đó trong mã. Ví dụ, đoạn mã sau chứa một lỗi khó nhận ra:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Dù biến `b` có kiểu `long long`, cả hai số trong biểu thức `a*a` đều có kiểu `int` và kết quả cũng có kiểu `int`. Vì vậy, biến `b` sẽ chứa một kết quả sai. Có thể giải quyết vấn đề bằng cách đổi kiểu của `a` thành `long long` hoặc đổi biểu thức thành `(long long)a*a`.

Thông thường, các bài toán thi được đặt sao cho kiểu `long long` là đủ. Dù vậy, cũng nên biết rằng trình biên dịch `g++` còn cung cấp kiểu 128 bit `__int128_t` với miền giá trị $-2^{127} \dots 2^{127} - 1$ hay xấp xỉ $-10^{38} \dots 10^{38}$. Tuy nhiên, kiểu này không có trong mọi hệ thống thi.

Số học modulo (Modular arithmetic)

Ta ký hiệu $x \bmod m$ là phần dư khi x được chia cho m . Ví dụ, $17 \bmod 5 = 2$, vì $17 = 3 \cdot 5 + 2$.

Đôi khi đáp án của một bài toán là một số rất lớn nhưng chỉ cần in nó "modulo m ", tức là phần dư khi đáp án được chia cho m (ví dụ, "modulo

$10^9 + 7$ ”). Ý tưởng là dù đáp án thật sự rất lớn, ta vẫn chỉ cần dùng các kiểu `int` và `long long`.

Một tính chất quan trọng của phần dư là trong phép cộng, phép trừ và phép nhân, có thể lấy phần dư trước khi thực hiện phép toán:

$$\begin{aligned}(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\(a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\(a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Vì vậy, ta có thể lấy phần dư sau mỗi phép toán và các số sẽ không bao giờ trở nên quá lớn.

Ví dụ, đoạn mã sau tính $n!$, giai thừa của n , modulo m :

```
long long x = 1;
for (int i = 2; i <= n; i++) {
    x = (x*i)%m;
}
cout << x%m << "\n";
```

Thông thường ta muốn phần dư luôn nằm trong khoảng $0 \dots m - 1$. Tuy nhiên, trong C++ và các ngôn ngữ khác, phần dư của một số âm có thể bằng không hoặc âm. Một cách đơn giản để bảo đảm không có phần dư âm là trước hết tính phần dư như bình thường rồi cộng thêm m nếu kết quả âm:

```
x = x%m;
if (x < 0) x += m;
```

Tuy nhiên, điều này chỉ cần thiết khi trong mã có phép trừ và phần dư có thể trở thành âm.

Số thực dấu phẩy động

Các kiểu số thực dấu phẩy động thường dùng trong lập trình thi đấu là `double` 64 bit và, như một phần mở rộng trong trình biên dịch `g++`, `long double` 80 bit. Trong hầu hết trường hợp, `double` là đủ, nhưng `long double` chính xác hơn.

Độ chính xác cần thiết của đáp án thường được nêu trong đề bài. Một cách đơn giản để in đáp án là dùng hàm `printf` và đưa số chữ số thập phân vào chuỗi định dạng. Ví dụ, đoạn mã sau in giá trị của x với 9 chữ số thập phân:

```
printf("%.9f\n", x);
```

Một khó khăn khi dùng số thực dấu phẩy động là một số số không thể được biểu diễn chính xác dưới dạng số thực dấu phẩy động, và sẽ có sai số làm tròn. Ví dụ, kết quả của đoạn mã sau khá bất ngờ:

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Do sai số làm tròn, giá trị của x nhỏ hơn 1 một chút, trong khi giá trị đúng phải là 1.

So sánh các số thực dấu phẩy động bằng toán tử `==` là rủi ro, vì có thể các giá trị đáng lẽ bằng nhau nhưng lại không bằng do sai số độ chính xác. Cách tốt hơn để so sánh số thực dấu phẩy động là giả sử hai số bằng nhau nếu hiệu giữa chúng nhỏ hơn ε , trong đó ε là một số nhỏ.

Trong thực tế, có thể so sánh các số như sau ($\varepsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {  
    // a and b are equal  
}
```

Lưu ý rằng tuy số thực dấu phẩy động không chính xác, các số nguyên đến một giới hạn nhất định vẫn có thể được biểu diễn chính xác. Ví dụ, khi dùng `double`, có thể biểu diễn chính xác mọi số nguyên có giá trị tuyệt đối không vượt quá 2^{53} .

1.4 Rút gọn mã

Mã ngắn là lý tưởng trong lập trình thi đấu, vì chương trình cần được viết nhanh nhất có thể. Vì vậy, các lập trình viên thi đấu thường định nghĩa tên ngắn hơn cho kiểu dữ liệu và các phần khác của mã.

Tên kiểu

Dùng lệnh `typedef` có thể đặt một tên ngắn hơn cho một kiểu dữ liệu. Ví dụ, tên `long long` khá dài, nên ta có thể định nghĩa tên ngắn hơn `ll`:

```
typedef long long ll;
```

Sau đó, đoạn mã

```
long long a = 123456789;  
long long b = 987654321;  
cout << a*b << "\n";
```

có thể được rút gọn như sau:

```
ll a = 123456789;  
ll b = 987654321;  
cout << a*b << "\n";
```

Lệnh `typedef` cũng có thể được dùng với các kiểu phức tạp hơn. Ví dụ, đoạn mã sau đặt tên `vi` cho một vector các số nguyên và tên `pi` cho một pair chứa hai số nguyên.

```
typedef vector<int> vi;  
typedef pair<int,int> pi;
```

Macro

Một cách khác để rút gọn mã là định nghĩa **macro (macros)**. Macro có nghĩa là một số chuỗi nhất định trong mã sẽ được thay đổi trước khi biên dịch. Trong C++, macro được định nghĩa bằng từ khóa `#define`.

Ví dụ, ta có thể định nghĩa các macro sau:

```
#define F first  
#define S second  
#define PB push_back  
#define MP make_pair
```

Sau đó, đoạn mã

```
v.push_back(make_pair(y1,x1));  
v.push_back(make_pair(y2,x2));  
int d = v[i].first + v[i].second;
```

có thể được rút gọn như sau:

```
v.PB(MP(y1,x1));  
v.PB(MP(y2,x2));  
int d = v[i].F + v[i].S;
```

Macro cũng có thể có tham số, nhờ đó có thể rút gọn vòng lặp và các cấu trúc khác. Ví dụ, ta có thể định nghĩa macro sau:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

Sau đó, đoạn mã

```
for (int i = 1; i <= n; i++) {  
    search(i);  
}
```

có thể được rút gọn như sau:

```
REP(i,1,n) {  
    search(i);  
}
```

Đôi khi macro gây ra lỗi có thể khó phát hiện. Ví dụ, xét macro sau tính bình phương của một số:

```
#define SQ(a) a*a
```

Macro này *không* phải lúc nào cũng hoạt động như mong đợi. Ví dụ, đoạn mã

```
cout << SQ(3+3) << "\n";
```

tương ứng với đoạn mã

```
cout << 3+3*3+3 << "\n"; // 15
```

Một phiên bản tốt hơn của macro là:

```
#define SQ(a) (a)*(a)
```

Bây giờ đoạn mã

```
cout << SQ(3+3) << "\n";
```

tương ứng với đoạn mã

```
cout << (3+3)*(3+3) << "\n"; // 36
```

1.5 Toán học

Toán học đóng vai trò quan trọng trong lập trình thi đấu, và không thể trở thành một lập trình viên thi đấu thành công nếu không có kỹ năng toán học tốt. Phần này thảo luận một số khái niệm và công thức toán học quan trọng cần dùng ở các phần sau của cuốn sách.

Công thức tổng

Mỗi tổng có dạng

$$\sum_{x=1}^n x^k = 1^k + 2^k + 3^k + \dots + n^k,$$

trong đó k là một số nguyên dương, có một công thức đóng là một đa thức bậc $k+1$. Ví dụ¹,

$$\sum_{x=1}^n x = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

và

$$\sum_{x=1}^n x^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

¹ Thậm chí còn có một công thức tổng quát cho các tổng như vậy, gọi là **công thức Faulhaber (Faulhaber's formula)**, nhưng nó quá phức tạp để trình bày ở đây.

Cấp số cộng (arithmetic progression) là một dãy số mà hiệu giữa hai số liên tiếp bất kỳ là hằng số. Ví dụ,

$$3, 7, 11, 15$$

là một cấp số cộng với công sai 4. Tổng của một cấp số cộng có thể được tính bằng công thức

$$\underbrace{a + \dots + b}_{n \text{ số}} = \frac{n(a + b)}{2}$$

trong đó a là số đầu tiên, b là số cuối cùng và n là số lượng số. Ví dụ,

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

Công thức dựa trên thực tế rằng tổng gồm n số và giá trị trung bình của mỗi số là $(a + b)/2$.

Cấp số nhân (geometric progression) là một dãy các số mà tỉ số giữa hai số liên tiếp bất kỳ là hằng số. Ví dụ,

$$3, 6, 12, 24$$

là một cấp số nhân với công bội 2. Tổng của một cấp số nhân có thể được tính bằng công thức

$$a + ak + ak^2 + \dots + b = \frac{bk - a}{k - 1}$$

trong đó a là số đầu tiên, b là số cuối cùng và tỉ số giữa hai số liên tiếp là k . Ví dụ,

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Công thức này có thể được suy ra như sau. Đặt

$$S = a + ak + ak^2 + \dots + b.$$

Nhân hai vế với k , ta được

$$kS = ak + ak^2 + ak^3 + \dots + bk,$$

và giải phương trình

$$kS - S = bk - a$$

sẽ thu được công thức.

Một trường hợp đặc biệt của tổng cấp số nhân là công thức

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1.$$

Tổng điều hòa (harmonic sum) là tổng có dạng

$$\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}.$$

Một cận trên cho tổng điều hòa là $\log_2(n) + 1$. Cụ thể, ta có thể thay đổi mỗi hạng tử $1/k$ sao cho k trở thành lũy thừa của hai gần nhất không vượt quá k . Ví dụ, khi $n = 6$, ta có thể ước lượng tổng như sau:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Cận trên này gồm $\log_2(n) + 1$ phần ($1, 2 \cdot 1/2, 4 \cdot 1/4$, v.v.), và giá trị của mỗi phần nhiều nhất là 1.

Lý thuyết tập hợp (Set theory)

Tập hợp (set) là một tập các phần tử. Ví dụ, tập

$$X = \{2, 4, 7\}$$

chứa các phần tử 2, 4 và 7. Ký hiệu $\emptyset$ biểu diễn tập rỗng, và $|S|$ biểu diễn kích thước của một tập S , tức là số phần tử trong tập. Ví dụ, trong tập trên, $|X| = 3$.

Nếu một tập S chứa một phần tử x , ta viết $x \in S$, và ngược lại ta viết $x \notin S$. Ví dụ, trong tập trên

$$4 \in X \quad \text{và} \quad 5 \notin X.$$

Các tập mới có thể được xây dựng bằng các phép toán trên tập hợp:

- **Giao (intersection)** $A \cap B$ gồm các phần tử nằm trong cả A và B . Ví dụ, nếu $A = \{1, 2, 5\}$ và $B = \{2, 4\}$, thì $A \cap B = \{2\}$.
- **Hợp (union)** $A \cup B$ gồm các phần tử nằm trong A hoặc B hoặc cả hai. Ví dụ, nếu $A = \{3, 7\}$ và $B = \{2, 3, 8\}$, thì $A \cup B = \{2, 3, 7, 8\}$.
- **Phần bù (complement)** $\bar{A}$ gồm các phần tử không nằm trong A . Cách hiểu phần bù phụ thuộc vào **tập vũ trụ (universal set)**, là tập chứa mọi phần tử có thể xét. Ví dụ, nếu $A = \{1, 2, 5, 7\}$ và tập vũ trụ là $\{1, 2, \dots, 10\}$, thì $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- **Hiệu (difference)** $A \setminus B = A \cap \bar{B}$ gồm các phần tử nằm trong A nhưng không nằm trong B . Lưu ý rằng B có thể chứa các phần tử không nằm trong A . Ví dụ, nếu $A = \{2, 3, 7, 8\}$ và $B = \{3, 5, 8\}$, thì $A \setminus B = \{2, 7\}$.

Nếu mỗi phần tử của A cũng thuộc về S , ta nói rằng A là một **tập con (subset)** của S , ký hiệu là $A \subset S$. Một tập S luôn có $2^{|S|}$ tập con, bao gồm cả tập rỗng. Ví dụ, các tập con của tập $\{2, 4, 7\}$ là

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\} \text{ và } \{2, 4, 7\}.$$

Một số tập thường dùng là $\mathbb{N}$ (số tự nhiên), $\mathbb{Z}$ (số nguyên), $\mathbb{Q}$ (số hữu tỉ) và $\mathbb{R}$ (số thực). Tập $\mathbb{N}$ có thể được định nghĩa theo hai cách, tùy tình huống: hoặc $\mathbb{N} = \{0, 1, 2, \dots\}$ hoặc $\mathbb{N} = \{1, 2, 3, \dots\}$.

Ta cũng có thể xây dựng một tập bằng một quy tắc có dạng

$$\{f(n) : n \in S\},$$

trong đó $f(n)$ là một hàm nào đó. Tập này chứa mọi phần tử có dạng $f(n)$, trong đó n là một phần tử trong S . Ví dụ, tập

$$X = \{2n : n \in \mathbb{Z}\}$$

chứa tất cả số nguyên chẵn.

Logic

Giá trị của một biểu thức logic là **đúng (true)** (1) hoặc **sai (false)** (0). Các toán tử logic quan trọng nhất là $\neg$ (**phủ định (negation)**), $\wedge$ (**hội (conjunction)**), $\vee$ (**tuyển (disjunction)**), $\Rightarrow$ (**kéo theo (implication)**) và $\Leftrightarrow$ (**tương đương (equivalence)**). Bảng sau cho biết ý nghĩa của các toán tử này:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Biểu thức $\neg A$ có giá trị đối ngược với A . Biểu thức $A \wedge B$ đúng nếu cả A và B đều đúng, và biểu thức $A \vee B$ đúng nếu A hoặc B hoặc cả hai đều đúng. Biểu thức $A \Rightarrow B$ đúng nếu mỗi khi A đúng thì B cũng đúng. Biểu thức $A \Leftrightarrow B$ đúng nếu A và B cùng đúng hoặc cùng sai.

Vị từ (predicate) là một biểu thức đúng hoặc sai tùy thuộc vào các tham số của nó. Vị từ thường được ký hiệu bằng chữ cái in hoa. Ví dụ, ta có thể định nghĩa vị từ $P(x)$ đúng khi và chỉ khi x là một số nguyên tố. Theo định nghĩa này, $P(7)$ đúng nhưng $P(8)$ sai.

Lượng từ (quantifier) kết nối một biểu thức logic với các phần tử của một tập. Các lượng từ quan trọng nhất là $\forall$ (**với mọi (for all)**) và $\exists$ (**tồn tại (there is)**). Ví dụ,

$$\forall x(\exists y(y < x))$$

có nghĩa là với mỗi phần tử x trong tập, tồn tại một phần tử y trong tập sao cho y nhỏ hơn x . Điều này đúng trong tập số nguyên, nhưng sai trong tập số tự nhiên.

Dùng ký hiệu đã mô tả ở trên, ta có thể biểu diễn nhiều loại mệnh đề logic. Ví dụ,

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

có nghĩa là nếu một số x lớn hơn 1 và không phải số nguyên tố, thì tồn tại các số a và b lớn hơn 1 và có tích bằng x . Mệnh đề này đúng trong tập số nguyên.

Hàm

Hàm $\lfloor x \rfloor$ làm tròn số x xuống một số nguyên, và hàm $\lceil x \rceil$ làm tròn số x lên một số nguyên. Ví dụ,

$$\lfloor 3/2 \rfloor = 1 \quad \text{và} \quad \lceil 3/2 \rceil = 2.$$

Các hàm $\min(x_1, x_2, \dots, x_n)$ và $\max(x_1, x_2, \dots, x_n)$ cho giá trị nhỏ nhất và lớn nhất trong các giá trị $x_1, x_2, \dots, x_n$. Ví dụ,

$$\min(1, 2, 3) = 1 \quad \text{và} \quad \max(1, 2, 3) = 3.$$

Giai thừa (factorial) $n!$ có thể được định nghĩa là

$$\prod_{x=1}^n x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

hoặc đệ quy là

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Số Fibonacci (Fibonacci numbers) xuất hiện trong nhiều tình huống. Chúng có thể được định nghĩa đệ quy như sau:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Các số Fibonacci đầu tiên là

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Cũng có một công thức đóng để tính số Fibonacci, đôi khi được gọi là **công thức Binet (Binet's formula)**:

$$f(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}.$$

Logarit

Logarit (logarithm) của một số x được ký hiệu là $\log_k(x)$, trong đó k là cơ số của logarit. Theo định nghĩa, $\log_k(x) = a$ đúng khi và chỉ khi $k^a = x$.

Một tính chất hữu ích của logarit là $\log_k(x)$ bằng số lần ta phải chia x cho k trước khi đạt tới số 1. Ví dụ, $\log_2(32) = 5$ vì cần 5 lần chia cho 2:

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Logarit thường được dùng trong phân tích thuật toán, vì nhiều thuật toán hiệu quả chia đôi một thứ gì đó ở mỗi bước. Do đó, ta có thể ước lượng độ hiệu quả của các thuật toán như vậy bằng logarit.

Logarit của một tích là

$$\log_k(ab) = \log_k(a) + \log_k(b),$$

và do đó,

$$\log_k(x^n) = n \cdot \log_k(x).$$

Ngoài ra, logarit của một thương là

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b).$$

Một công thức hữu ích khác là

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

và nhờ đó, có thể tính logarit với bất kỳ cơ số nào nếu có cách tính logarit với một cơ số cố định nào đó.

Logarit tự nhiên (natural logarithm) $\ln(x)$ của một số x là logarit có cơ số $e \approx 2.71828$. Một tính chất khác của logarit là số chữ số của một số nguyên x trong cơ số b là $\lfloor \log_b(x) + 1 \rfloor$. Ví dụ, biểu diễn của 123 trong cơ số 2 là 1111011 và $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Cuộc thi và tài nguyên

IOI

International Olympiad in Informatics (IOI) là một cuộc thi lập trình hàng năm dành cho học sinh trung học. Mỗi quốc gia được phép gửi một đội gồm bốn học sinh đến cuộc thi. Thường có khoảng 300 thí sinh từ 80 quốc gia.

IOI gồm hai cuộc thi, mỗi cuộc kéo dài năm giờ. Trong cả hai cuộc thi, thí sinh được yêu cầu giải ba bài toán thuật toán với độ khó khác nhau. Các bài toán được chia thành các subtasks, mỗi subtask có một số điểm được gán. Dù các thí sinh được chia theo đội, họ thi đấu với tư cách cá nhân.

Đề cương IOI [41] quy định các chủ đề có thể xuất hiện trong bài IOI. Hầu như tất cả chủ đề trong đề cương IOI đều được cuốn sách này bao quát.

Thí sinh dự IOI được chọn thông qua các cuộc thi quốc gia. Trước IOI, nhiều cuộc thi khu vực được tổ chức, như Baltic Olympiad in Informatics (BOI), Central European Olympiad in Informatics (CEOI) và Asia-Pacific Informatics Olympiad (APIO).

Một số quốc gia tổ chức các cuộc thi luyện tập trực tuyến cho thí sinh IOI tương lai, như Croatian Open Competition in Informatics [11] và USA Computing Olympiad [68]. Ngoài ra, một bộ sưu tập lớn các bài toán từ các cuộc thi Ba Lan cũng có sẵn trực tuyến [60].

ICPC

International Collegiate Programming Contest (ICPC) là một cuộc thi lập trình hằng năm dành cho sinh viên đại học. Mỗi đội trong cuộc thi gồm ba sinh viên, và khác với IOI, các sinh viên làm việc cùng nhau; mỗi đội chỉ có một máy tính.

ICPC gồm nhiều vòng, và cuối cùng những đội tốt nhất được mời đến World Finals. Dù có hàng chục nghìn thí sinh trong cuộc thi, chỉ có một số lượng nhỏ² suất chung kết, vì vậy ngay cả việc lọt vào chung kết cũng là một thành tích lớn ở một số khu vực.

Trong mỗi cuộc thi ICPC, các đội có năm giờ để giải khoảng mười bài toán thuật toán. Lời giải cho một bài toán chỉ được chấp nhận nếu nó giải tất cả bộ test một cách hiệu quả. Trong cuộc thi, thí sinh có thể xem kết quả của các đội khác, nhưng trong giờ cuối cùng bảng điểm bị đóng băng và không thể thấy kết quả của các lần nộp cuối.

Các chủ đề có thể xuất hiện ở ICPC không được quy định rõ như ở IOI. Dù thế nào, rõ ràng ICPC đòi hỏi nhiều kiến thức hơn, đặc biệt là nhiều kỹ năng toán học hơn.

Cuộc thi trực tuyến

Cũng có nhiều cuộc thi trực tuyến mở cho mọi người. Hiện nay, trang thi hoạt động sôi nổi nhất là Codeforces, tổ chức các cuộc thi gần như hằng tuần. Trong Codeforces, thí sinh được chia thành hai division: người mới thi ở Div2 và lập trình viên giàu kinh nghiệm hơn ở Div1. Các trang thi khác gồm AtCoder, CS Academy, HackerRank và Topcoder.

Một số công ty tổ chức các cuộc thi trực tuyến có vòng chung kết onsite. Ví dụ về các cuộc thi như vậy là Facebook Hacker Cup, Google Code Jam và Yandex.Algorithm. Tất nhiên, các công ty cũng dùng những cuộc thi đó để tuyển dụng: thi đấu tốt trong một cuộc thi là một cách hay để chứng minh kỹ năng.

Sách

Đã có một số cuốn sách (ngoài cuốn sách này) tập trung vào lập trình thi đấu và giải quyết bài toán thuật toán:

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim and F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]
- K. Diks et al.: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

²Số suất vào chung kết chính xác thay đổi theo từng năm; năm 2017 có 133 suất chung kết.

Hai cuốn đầu dành cho người mới bắt đầu, trong khi cuốn cuối chứa nội dung nâng cao.

Tất nhiên, các sách thuật toán tổng quát cũng phù hợp với lập trình viên thi đấu. Một số cuốn sách phổ biến là:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: *Introduction to Algorithms* [13]
- J. Kleinberg and É. Tardos: *Algorithm Design* [45]
- S. S. Skiena: *The Algorithm Design Manual* [58]

Chương 2

Độ phức tạp thời gian

Hiệu quả của thuật toán rất quan trọng trong lập trình thi đấu. Thông thường, thiết kế một thuật toán giải được bài toán một cách chậm chạp là việc khá dễ, nhưng thử thách thật sự là nghĩ ra một thuật toán nhanh. Nếu thuật toán quá chậm, nó sẽ chỉ nhận được một phần điểm hoặc không có điểm nào.

Độ phức tạp thời gian (Time Complexity) của một thuật toán ước lượng thuật toán sẽ dùng bao nhiêu thời gian cho một đầu vào nào đó. Ý tưởng là biểu diễn hiệu quả bằng một hàm mà tham số là kích thước đầu vào. Bằng cách tính độ phức tạp thời gian, ta có thể biết thuật toán có đủ nhanh hay không mà chưa cần cài đặt nó.

2.1 Quy tắc tính toán

Độ phức tạp thời gian của một thuật toán được ký hiệu là $O(\dots)$, trong đó ba dấu chấm biểu diễn một hàm nào đó. Thông thường, biến n biểu diễn kích thước đầu vào. Ví dụ, nếu đầu vào là một mảng các số, n sẽ là kích thước của mảng, còn nếu đầu vào là một chuỗi, n sẽ là độ dài của chuỗi.

Vòng lặp

Một lý do phổ biến khiến thuật toán chậm là nó chứa nhiều vòng lặp duyệt qua đầu vào. Thuật toán càng có nhiều vòng lặp lồng nhau thì càng chậm. Nếu có k vòng lặp lồng nhau, độ phức tạp thời gian là $O(n^k)$.

Ví dụ, độ phức tạp thời gian của đoạn mã sau là $O(n)$:

```
for (int i = 1; i <= n; i++) {  
    // code  
}
```

Còn độ phức tạp thời gian của đoạn mã sau là $O(n^2)$:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {
```

```
}  
    // code  
}
```

Bậc độ lớn

Độ phức tạp thời gian không cho ta biết chính xác đoạn mã bên trong vòng lặp được thực thi bao nhiêu lần, mà chỉ cho biết bậc độ lớn. Trong các ví dụ sau, đoạn mã bên trong vòng lặp được thực thi $3n$, $n+5$ và $\lceil n/2 \rceil$ lần, nhưng độ phức tạp thời gian của mỗi đoạn mã đều là $O(n)$.

```
for (int i = 1; i <= 3*n; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n+5; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n; i += 2) {  
    // code  
}
```

Một ví dụ khác, độ phức tạp thời gian của đoạn mã sau là $O(n^2)$:

```
for (int i = 1; i <= n; i++) {  
    for (int j = i+1; j <= n; j++) {  
        // code  
    }  
}
```

Các pha

Nếu thuật toán gồm nhiều pha liên tiếp, độ phức tạp thời gian tổng thể là độ phức tạp lớn nhất của một pha. Lý do là pha chậm nhất thường là nút thắt của chương trình.

Ví dụ, đoạn mã sau gồm ba pha có độ phức tạp thời gian $O(n)$, $O(n^2)$ và $O(n)$. Vì vậy, độ phức tạp thời gian tổng thể là $O(n^2)$.

```
for (int i = 1; i <= n; i++) {  
    // code  
}  
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // code  
    }  
}
```

```
    }  
}  
for (int i = 1; i <= n; i++) {  
    // code  
}
```

Nhiều biến

Đôi khi độ phức tạp thời gian phụ thuộc vào nhiều yếu tố. Trong trường hợp này, công thức độ phức tạp thời gian chứa nhiều biến.

Ví dụ, độ phức tạp thời gian của đoạn mã sau là $O(nm)$:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= m; j++) {  
        // code  
    }  
}
```

Đệ quy

Độ phức tạp thời gian của một hàm đệ quy phụ thuộc vào số lần hàm được gọi và độ phức tạp thời gian của một lần gọi. Độ phức tạp thời gian tổng cộng là tích của hai giá trị này.

Ví dụ, xét hàm sau:

```
void f(int n) {  
    if (n == 1) return;  
    f(n-1);  
}
```

Lời gọi $f(n)$ gây ra n lời gọi hàm, và độ phức tạp thời gian của mỗi lời gọi là $O(1)$. Vì vậy, độ phức tạp thời gian tổng cộng là $O(n)$.

Một ví dụ khác, xét hàm sau:

```
void g(int n) {  
    if (n == 1) return;  
    g(n-1);  
    g(n-1);  
}
```

Trong trường hợp này, mỗi lời gọi hàm sinh ra hai lời gọi khác, trừ khi $n = 1$. Hãy xem điều gì xảy ra khi g được gọi với tham số n . Bảng sau cho thấy các lời gọi hàm được tạo ra bởi một lời gọi duy nhất này:

lời gọi hàm	số lời gọi
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
$\dots$	$\dots$
$g(1)$	2^{n-1}

Dựa trên điều này, độ phức tạp thời gian là

$$1 + 2 + 4 + \dots + 2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 Lớp độ phức tạp

Danh sách sau chứa các độ phức tạp thời gian phổ biến của thuật toán:

- $O(1)$ Thời gian chạy của một thuật toán **hằng số** không phụ thuộc vào kích thước đầu vào. Một thuật toán thời gian hằng số điển hình là một công thức trực tiếp tính ra đáp án.
- $O(\log n)$ Một thuật toán **logarithmic** thường chia đôi kích thước đầu vào ở mỗi bước. Thời gian chạy của thuật toán như vậy có dạng logarithmic, vì $\log_2 n$ bằng số lần cần chia n cho 2 để nhận được 1.
- $O(\sqrt{n})$ Một **thuật toán căn bậc hai** chậm hơn $O(\log n)$ nhưng nhanh hơn $O(n)$. Một tính chất đặc biệt của căn bậc hai là $\sqrt{n} = n/\sqrt{n}$, nên căn bậc hai $\sqrt{n}$ nằm, theo một nghĩa nào đó, ở giữa đầu vào.
- $O(n)$ Một thuật toán **tuyến tính** duyệt qua đầu vào một số lần hằng số. Đây thường là độ phức tạp thời gian tốt nhất có thể, vì thường cần truy cập mỗi phần tử đầu vào ít nhất một lần trước khi đưa ra đáp án.
- $O(n \log n)$ Độ phức tạp thời gian này thường cho thấy thuật toán sắp xếp đầu vào, vì độ phức tạp thời gian của các thuật toán sắp xếp hiệu quả là $O(n \log n)$. Một khả năng khác là thuật toán dùng một cấu trúc dữ liệu mà mỗi thao tác mất $O(\log n)$ thời gian.
- $O(n^2)$ Một thuật toán **bậc hai** thường chứa hai vòng lặp lồng nhau. Ta có thể duyệt qua mọi cặp phần tử của đầu vào trong $O(n^2)$ thời gian.
- $O(n^3)$ Một thuật toán **bậc ba** thường chứa ba vòng lặp lồng nhau. Ta có thể duyệt qua mọi bộ ba phần tử của đầu vào trong $O(n^3)$ thời gian.
- $O(2^n)$ Độ phức tạp thời gian này thường cho thấy thuật toán duyệt qua tất cả tập con của các phần tử đầu vào. Ví dụ, các tập con của $\{1, 2, 3\}$ là $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ và $\{1, 2, 3\}$.
- $O(n!)$ Độ phức tạp thời gian này thường cho thấy thuật toán duyệt qua tất cả hoán vị của các phần tử đầu vào. Ví dụ, các hoán vị của $\{1, 2, 3\}$ là $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ và $(3, 2, 1)$.

Một thuật toán là **đa thức** nếu độ phức tạp thời gian của nó nhiều nhất là $O(n^k)$, trong đó k là một hằng số. Tất cả các độ phức tạp thời gian ở trên, ngoại trừ $O(2^n)$ và $O(n!)$, đều là đa thức. Trong thực tế, hằng số k thường nhỏ, và vì thế độ phức tạp thời gian đa thức thường có nghĩa là thuật toán *hiệu quả*.

Phần lớn thuật toán trong sách này là đa thức. Dù vậy, có nhiều bài toán quan trọng mà hiện chưa biết thuật toán đa thức nào, tức là chưa ai biết cách giải chúng hiệu quả. Các bài toán **NP-hard** là một tập quan trọng các bài toán mà chưa biết thuật toán đa thức nào¹.

2.3 Ước lượng hiệu quả

Bằng cách tính độ phức tạp thời gian của một thuật toán, ta có thể kiểm tra, trước khi cài đặt thuật toán, rằng nó có đủ hiệu quả cho bài toán hay không. Điểm xuất phát cho các ước lượng là thực tế rằng một máy tính hiện đại có thể thực hiện vài trăm triệu phép toán trong một giây.

Ví dụ, giả sử giới hạn thời gian của một bài toán là một giây và kích thước đầu vào là $n = 10^5$. Nếu độ phức tạp thời gian là $O(n^2)$, thuật toán sẽ thực hiện khoảng $(10^5)^2 = 10^{10}$ phép toán. Việc này sẽ mất ít nhất vài chục giây, nên thuật toán có vẻ quá chậm để giải bài toán.

Mặt khác, dựa trên kích thước đầu vào, ta có thể thử *đoán* độ phức tạp thời gian cần có của thuật toán giải bài toán. Bảng sau chứa một vài ước lượng hữu ích khi giả sử giới hạn thời gian là một giây.

kích thước đầu vào	độ phức tạp thời gian cần có
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ hoặc $O(n)$
n lớn	$O(1)$ hoặc $O(\log n)$

Ví dụ, nếu kích thước đầu vào là $n = 10^5$, có lẽ người ra đề kỳ vọng độ phức tạp thời gian của thuật toán là $O(n)$ hoặc $O(n \log n)$. Thông tin này giúp việc thiết kế thuật toán dễ hơn, vì nó loại bỏ những hướng tiếp cận dẫn tới thuật toán có độ phức tạp thời gian tệ hơn.

Dù vậy, cần nhớ rằng độ phức tạp thời gian chỉ là một ước lượng về hiệu quả, vì nó ẩn đi các *hệ số hằng*. Ví dụ, một thuật toán chạy trong $O(n)$ thời gian có thể thực hiện $n/2$ hoặc $5n$ phép toán. Điều này ảnh hưởng đáng kể đến thời gian chạy thực tế của thuật toán.

¹Một cuốn sách kinh điển về chủ đề này là *Computers and Intractability: A Guide to the Theory of NP-Completeness* của M. R. Garey và D. S. Johnson [28].

2.4 Tổng đoạn con lớn nhất

Thường có nhiều thuật toán khả dĩ để giải một bài toán, và độ phức tạp thời gian của chúng khác nhau. Mục này thảo luận một bài toán kinh điển có lời giải trực tiếp $O(n^3)$. Tuy nhiên, bằng cách thiết kế một thuật toán tốt hơn, ta có thể giải bài toán trong $O(n^2)$ thời gian và thậm chí trong $O(n)$ thời gian.

Cho một mảng gồm n số, nhiệm vụ của ta là tính **tổng đoạn con lớn nhất** (Maximum Subarray Sum), tức là tổng lớn nhất có thể của một dãy các giá trị liên tiếp trong mảng². Bài toán thú vị khi mảng có thể chứa các giá trị âm. Ví dụ, trong mảng

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

đoạn con sau tạo ra tổng lớn nhất 10:

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

Ta giả sử rằng đoạn con rỗng được cho phép, nên tổng đoạn con lớn nhất luôn ít nhất là 0.

Thuật toán 1

Một cách trực tiếp để giải bài toán là duyệt qua mọi đoạn con có thể, tính tổng các giá trị trong mỗi đoạn con và duy trì tổng lớn nhất. Đoạn mã sau cài đặt thuật toán này:

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
        best = max(best, sum);
    }
}
cout << best << "\n";
```

Các biến a và b cố định chỉ số đầu và chỉ số cuối của đoạn con, và tổng các giá trị được tính vào biến sum . Biến $best$ chứa tổng lớn nhất tìm được trong quá trình tìm kiếm.

Độ phức tạp thời gian của thuật toán là $O(n^3)$, vì nó gồm ba vòng lặp lồng nhau duyệt qua đầu vào.

²Cuốn sách *Programming Pearls* của J. Bentley [8] đã làm bài toán này trở nên nổi tiếng.

Thuật toán 2

Dễ dàng làm Thuật toán 1 hiệu quả hơn bằng cách loại bỏ một vòng lặp. Điều này có thể thực hiện bằng cách tính tổng đồng thời khi đầu phải của đoạn con di chuyển. Kết quả là đoạn mã sau:

```
int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best, sum);
    }
}
cout << best << "\n";
```

Sau thay đổi này, độ phức tạp thời gian là $O(n^2)$.

Thuật toán 3

Đáng ngạc nhiên là có thể giải bài toán trong $O(n)$ thời gian³, nghĩa là chỉ cần một vòng lặp. Ý tưởng là tính, với mỗi vị trí trong mảng, tổng lớn nhất của một đoạn con kết thúc tại vị trí đó. Sau đó, đáp án của bài toán là giá trị lớn nhất trong các tổng đó.

Xét bài toán con tìm đoạn con có tổng lớn nhất kết thúc tại vị trí k . Có hai khả năng:

1. Đoạn con chỉ chứa phần tử ở vị trí k .
2. Đoạn con gồm một đoạn con kết thúc tại vị trí $k - 1$, theo sau là phần tử ở vị trí k .

Trong trường hợp sau, vì ta muốn tìm một đoạn con có tổng lớn nhất, đoạn con kết thúc tại vị trí $k - 1$ cũng phải có tổng lớn nhất. Vì vậy, ta có thể giải bài toán hiệu quả bằng cách tính tổng đoạn con lớn nhất cho từng vị trí kết thúc từ trái sang phải.

Đoạn mã sau cài đặt thuật toán:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k], sum + array[k]);
    best = max(best, sum);
}
cout << best << "\n";
```

Thuật toán chỉ chứa một vòng lặp duyệt qua đầu vào, nên độ phức tạp thời gian là $O(n)$. Đây cũng là độ phức tạp thời gian tốt nhất có thể, vì mọi

³Trong [8], thuật toán tuyến tính này được ghi nhận cho J. B. Kadane, và thuật toán đôi khi được gọi là **thuật toán Kadane**.

thuật toán cho bài toán này đều phải xét tất cả phần tử của mảng ít nhất một lần.

So sánh hiệu quả

Việc nghiên cứu hiệu quả thực tế của các thuật toán là điều thú vị. Bảng sau cho thấy thời gian chạy của các thuật toán trên với những giá trị n khác nhau trên một máy tính hiện đại.

Trong mỗi phép thử, đầu vào được sinh ngẫu nhiên. Thời gian cần để đọc đầu vào không được đo.

kích thước mảng n	Thuật toán 1	Thuật toán 2	Thuật toán 3
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	> 10.0 s	0.1 s	0.0 s
10^5	> 10.0 s	5.3 s	0.0 s
10^6	> 10.0 s	> 10.0 s	0.0 s
10^7	> 10.0 s	> 10.0 s	0.0 s

So sánh cho thấy mọi thuật toán đều hiệu quả khi kích thước đầu vào nhỏ, nhưng đầu vào lớn hơn làm lộ ra những khác biệt đáng kể trong thời gian chạy của các thuật toán. Thuật toán 1 trở nên chậm khi $n = 10^4$, và Thuật toán 2 trở nên chậm khi $n = 10^5$. Chỉ Thuật toán 3 có thể xử lý cả những đầu vào lớn nhất gần như tức thì.

Chương 3

Sắp xếp

Sắp xếp (Sorting) là một bài toán thiết kế thuật toán cơ bản. Nhiều thuật toán hiệu quả dùng sắp xếp như một thủ tục con, vì thường dễ xử lý dữ liệu hơn nếu các phần tử đang ở thứ tự đã sắp xếp.

Ví dụ, bài toán “một mảng có chứa hai phần tử bằng nhau hay không?” rất dễ giải bằng sắp xếp. Nếu mảng chứa hai phần tử bằng nhau, chúng sẽ nằm cạnh nhau sau khi sắp xếp, nên ta dễ dàng tìm thấy chúng. Tương tự, bài toán “phần tử nào xuất hiện nhiều nhất trong một mảng?” cũng có thể được giải theo cách này.

Có nhiều thuật toán sắp xếp, và chúng cũng là những ví dụ tốt về cách áp dụng các kỹ thuật thiết kế thuật toán khác nhau. Các thuật toán sắp xếp tổng quát hiệu quả chạy trong $O(n \log n)$ thời gian, và nhiều thuật toán dùng sắp xếp làm thủ tục con cũng có độ phức tạp thời gian này.

3.1 Lý thuyết sắp xếp

Bài toán cơ bản trong sắp xếp như sau:

Cho một mảng chứa n phần tử, nhiệm vụ của bạn là sắp xếp các phần tử theo thứ tự tăng dần.

Ví dụ, mảng

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

sau khi sắp xếp sẽ trở thành:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Các thuật toán $O(n^2)$

Các thuật toán đơn giản để sắp xếp một mảng chạy trong $O(n^2)$ thời gian. Những thuật toán như vậy ngắn và thường gồm hai vòng lặp lồng nhau. Một

thuật toán sắp xếp thời gian $O(n^2)$ nổi tiếng là **bubble sort**, trong đó các phần tử “nổi bọt” trong mảng theo giá trị của chúng.

Bubble sort gồm n vòng. Ở mỗi vòng, thuật toán duyệt qua các phần tử của mảng. Bất cứ khi nào phát hiện hai phần tử liên tiếp không ở đúng thứ tự, thuật toán đổi chỗ chúng. Thuật toán có thể được cài đặt như sau:

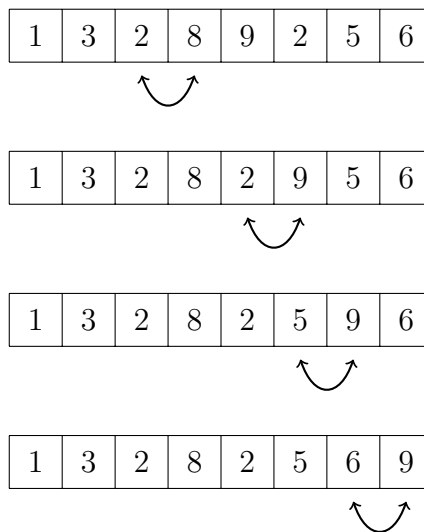
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n-1; j++) {  
        if (array[j] > array[j+1]) {  
            swap(array[j], array[j+1]);  
        }  
    }  
}
```

Sau vòng đầu tiên của thuật toán, phần tử lớn nhất sẽ ở đúng vị trí, và nói chung, sau k vòng, k phần tử lớn nhất sẽ ở đúng vị trí. Vì vậy, sau n vòng, toàn bộ mảng sẽ được sắp xếp.

Ví dụ, trong mảng

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

vòng đầu tiên của bubble sort đổi chỗ các phần tử như sau:



Nghịch thế

Bubble sort là một ví dụ về thuật toán sắp xếp luôn đổi chỗ các phần tử *liên tiếp* trong mảng. Hóa ra độ phức tạp thời gian của một thuật toán như vậy *luôn* ít nhất là $O(n^2)$, vì trong trường hợp xấu nhất, cần $O(n^2)$ lần đổi chỗ để sắp xếp mảng.

Một khái niệm hữu ích khi phân tích thuật toán sắp xếp là **nghịch thế** (inversion): một cặp phần tử mảng ($\text{array}[a], \text{array}[b]$) sao cho $a < b$ và $\text{array}[a] > \text{array}[b]$, tức là các phần tử đang ở sai thứ tự. Ví dụ, mảng

1	2	2	6	3	5	9	8
---	---	---	---	---	---	---	---

có ba nghịch thế: (6, 3), (6, 5) và (9, 8). Số nghịch thế cho biết cần bao nhiêu công việc để sắp xếp mảng. Một mảng đã được sắp xếp hoàn toàn khi không có nghịch thế nào. Mặt khác, nếu các phần tử mảng đang ở thứ tự ngược, số nghịch thế là lớn nhất có thể:

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

Đổi chỗ một cặp phần tử liên tiếp đang ở sai thứ tự loại bỏ đúng một nghịch thế khỏi mảng. Do đó, nếu một thuật toán sắp xếp chỉ có thể đổi chỗ các phần tử liên tiếp, mỗi lần đổi chỗ loại bỏ nhiều nhất một nghịch thế, và độ phức tạp thời gian của thuật toán ít nhất là $O(n^2)$.

Các thuật toán $O(n \log n)$

Ta có thể sắp xếp một mảng hiệu quả trong $O(n \log n)$ thời gian bằng các thuật toán không bị giới hạn ở việc đổi chỗ các phần tử liên tiếp. Một thuật toán như vậy là **merge sort**¹, dựa trên đệ quy.

Merge sort sắp xếp một mảng con `array[a ... b]` như sau:

1. Nếu $a = b$, không làm gì cả, vì mảng con đã được sắp xếp.
2. Tính vị trí của phần tử giữa: $k = \lfloor (a + b)/2 \rfloor$.
3. Sắp xếp đệ quy mảng con `array[a ... k]`.
4. Sắp xếp đệ quy mảng con `array[k + 1 ... b]`.
5. *Trộn* các mảng con đã sắp xếp `array[a ... k]` và `array[k + 1 ... b]` thành một mảng con đã sắp xếp `array[a ... b]`.

Merge sort là một thuật toán hiệu quả, vì nó chia đôi kích thước mảng con ở mỗi bước. Đệ quy gồm $O(\log n)$ mức, và xử lý mỗi mức mất $O(n)$ thời gian. Việc trộn các mảng con `array[a ... k]` và `array[k + 1 ... b]` có thể thực hiện trong thời gian tuyến tính, vì chúng đã được sắp xếp.

Ví dụ, xét việc sắp xếp mảng sau:

1	3	6	2	8	2	5	9
---	---	---	---	---	---	---	---

Mảng sẽ được chia thành hai mảng con như sau:

1	3	6	2
---	---	---	---

8	2	5	9
---	---	---	---

Sau đó, các mảng con sẽ được sắp xếp đệ quy như sau:

¹Theo [47], merge sort được J. von Neumann phát minh năm 1945.

1	2	3	6
---	---	---	---

2	5	8	9
---	---	---	---

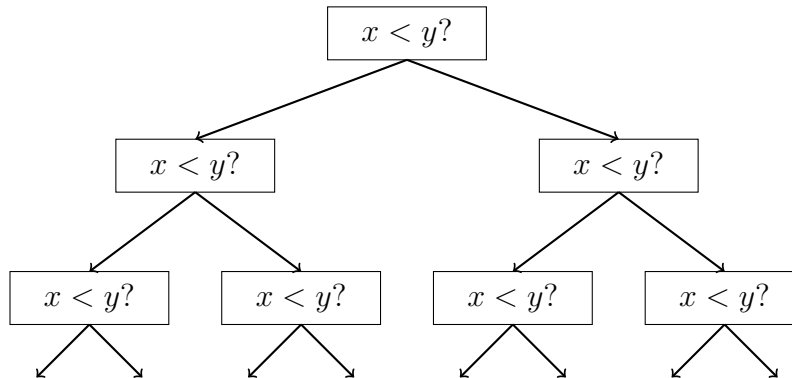
Cuối cùng, thuật toán trộn các mảng con đã sắp xếp và tạo ra mảng đã sắp xếp cuối cùng:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Cận dưới của sắp xếp

Có thể sắp xếp một mảng nhanh hơn $O(n \log n)$ thời gian hay không? Hóa ra điều này là *không* thể khi ta giới hạn ở các thuật toán sắp xếp dựa trên việc so sánh các phần tử mảng.

Cận dưới cho độ phức tạp thời gian có thể được chứng minh bằng cách xem sắp xếp như một quá trình mà mỗi lần so sánh hai phần tử cho thêm thông tin về nội dung của mảng. Quá trình tạo ra cây sau:



Ở đây “ $x < y$?” nghĩa là một số phần tử x và y được so sánh. Nếu $x < y$, quá trình tiếp tục sang trái, ngược lại thì sang phải. Kết quả của quá trình là các cách có thể để sắp xếp mảng, tổng cộng $n!$ cách. Vì lý do này, chiều cao của cây phải ít nhất là

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n).$$

Ta nhận được một cận dưới cho tổng này bằng cách chọn $n/2$ phần tử cuối và thay giá trị của mỗi phần tử bằng $\log_2(n/2)$. Điều này cho ước lượng

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

nên chiều cao của cây và số bước tối thiểu có thể có của một thuật toán sắp xếp trong trường hợp xấu nhất ít nhất là $n \log n$.

Counting sort

Cận dưới $n \log n$ không áp dụng cho các thuật toán không so sánh các phần tử mảng mà dùng một số thông tin khác. Một ví dụ về thuật toán như vậy

là **counting sort**, sắp xếp một mảng trong $O(n)$ thời gian với giả thiết mọi phần tử trong mảng là số nguyên từ $0 \dots c$ và $c = O(n)$.

Thuật toán tạo một mảng *ghi nhận*, mà các chỉ số của nó là các phần tử của mảng ban đầu. Thuật toán duyệt qua mảng ban đầu và tính mỗi phần tử xuất hiện bao nhiêu lần trong mảng.

Ví dụ, mảng

1	3	6	9	9	3	5	9
---	---	---	---	---	---	---	---

tương ứng với mảng ghi nhận sau:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

Ví dụ, giá trị ở vị trí 3 trong mảng ghi nhận là 2, vì phần tử 3 xuất hiện 2 lần trong mảng ban đầu.

Việc xây dựng mảng ghi nhận mất $O(n)$ thời gian. Sau đó, mảng đã sắp xếp có thể được tạo trong $O(n)$ thời gian vì số lần xuất hiện của mỗi phần tử có thể được lấy từ mảng ghi nhận. Vì vậy, độ phức tạp thời gian tổng cộng của counting sort là $O(n)$.

Counting sort là một thuật toán rất hiệu quả nhưng chỉ có thể dùng khi hằng số c đủ nhỏ, để các phần tử mảng có thể được dùng làm chỉ số trong mảng ghi nhận.

3.2 Sắp xếp trong C++

Trong một kỳ thi, gần như không bao giờ nên dùng một thuật toán sắp xếp tự viết, vì các ngôn ngữ lập trình đã có sẵn những cài đặt tốt. Ví dụ, thư viện chuẩn C++ chứa hàm `sort`, có thể dễ dàng dùng để sắp xếp mảng và các cấu trúc dữ liệu khác.

Việc dùng hàm thư viện có nhiều lợi ích. Thứ nhất, nó tiết kiệm thời gian vì không cần cài đặt hàm. Thứ hai, cài đặt của thư viện chắc chắn đúng và hiệu quả: rất khó có khả năng một hàm sắp xếp tự viết lại tốt hơn.

Trong mục này ta sẽ xem cách dùng hàm `sort` của C++. Đoạn mã sau sắp xếp một vector theo thứ tự tăng dần:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(), v.end());
```

Sau khi sắp xếp, nội dung của vector sẽ là `[2, 3, 3, 4, 5, 5, 8]`. Thứ tự sắp xếp mặc định là tăng dần, nhưng có thể sắp xếp ngược như sau:

```
sort(v.rbegin(), v.rend());
```

Một mảng thông thường có thể được sắp xếp như sau:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a, a+n);
```

Đoạn mã sau sắp xếp xâu s:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Sắp xếp một xâu nghĩa là các ký tự của xâu được sắp xếp. Ví dụ, xâu “monkey” trở thành “ekmnoy”.

Toán tử so sánh

Hàm sort yêu cầu một **toán tử so sánh** được định nghĩa cho kiểu dữ liệu của các phần tử cần sắp xếp. Khi sắp xếp, toán tử này sẽ được dùng bất cứ khi nào cần xác định thứ tự của hai phần tử.

Hầu hết kiểu dữ liệu C++ có toán tử so sánh dựng sẵn, và các phần tử thuộc những kiểu đó có thể được sắp xếp tự động. Ví dụ, số được sắp xếp theo giá trị và xâu được sắp xếp theo thứ tự chữ cái.

Các cặp (pair) được sắp xếp trước hết theo phần tử thứ nhất của chúng (first). Tuy nhiên, nếu phần tử thứ nhất của hai cặp bằng nhau, chúng được sắp xếp theo phần tử thứ hai (second):

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

Sau đó, thứ tự của các cặp là (1, 2), (1, 5) và (2, 3).

Tương tự, các bộ (tuple) được sắp xếp trước hết theo phần tử thứ nhất, sau đó theo phần tử thứ hai, v.v.²:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
```

Sau đó, thứ tự của các tuple là (1, 5, 3), (2, 1, 3) và (2, 1, 4).

Struct tự định nghĩa

Các struct tự định nghĩa không tự động có toán tử so sánh. Toán tử nên được định nghĩa bên trong struct dưới dạng một hàm operator<, có tham số là một phần tử khác cùng kiểu. Toán tử nên trả về true nếu phần tử nhỏ hơn tham số, và false trong trường hợp ngược lại.

²Lưu ý rằng trong một số trình biên dịch cũ, cần dùng hàm make_tuple để tạo tuple thay vì dấu ngoặc nhọn (ví dụ, make_tuple(2,1,4) thay vì {2,1,4}).

Ví dụ, struct P sau chứa tọa độ x và y của một điểm. Toán tử so sánh được định nghĩa sao cho các điểm được sắp xếp trước hết theo tọa độ x và sau đó theo tọa độ y.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Hàm so sánh

Cũng có thể truyền một **hàm so sánh** bên ngoài cho hàm sort như một hàm callback. Ví dụ, hàm so sánh comp sau sắp xếp các xâu trước hết theo độ dài và sau đó theo thứ tự chữ cái:

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Bây giờ một vector các xâu có thể được sắp xếp như sau:

```
sort(v.begin(), v.end(), comp);
```

3.3 Tìm kiếm nhị phân

Một phương pháp tổng quát để tìm kiếm một phần tử trong mảng là dùng vòng lặp for duyệt qua các phần tử của mảng. Ví dụ, đoạn mã sau tìm một phần tử x trong một mảng:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

Độ phức tạp thời gian của cách này là $O(n)$, vì trong trường hợp xấu nhất, cần kiểm tra tất cả phần tử của mảng. Nếu thứ tự của các phần tử là tùy ý, đây cũng là cách tốt nhất có thể, vì không có thông tin bổ sung nào cho biết ta nên tìm phần tử x ở đâu trong mảng.

Tuy nhiên, nếu mảng đã được *sắp xếp*, tình hình sẽ khác. Trong trường hợp này, có thể thực hiện tìm kiếm nhanh hơn nhiều, vì thứ tự của các phần

tử trong mảng dẫn hướng quá trình tìm kiếm. Thuật toán **tìm kiếm nhị phân** (Binary Search) sau tìm kiếm hiệu quả một phần tử trong mảng đã sắp xếp trong $O(\log n)$ thời gian.

Phương pháp 1

Cách thông thường để cài đặt tìm kiếm nhị phân giống như tìm một từ trong từ điển. Quá trình tìm kiếm duy trì một vùng đang xét trong mảng, ban đầu chứa tất cả phần tử mảng. Sau đó, một số bước được thực hiện, mỗi bước chia đôi kích thước vùng này.

Ở mỗi bước, quá trình tìm kiếm kiểm tra phần tử giữa của vùng đang xét. Nếu phần tử giữa là phần tử mục tiêu, tìm kiếm kết thúc. Ngược lại, tìm kiếm tiếp tục đệ quy sang nửa trái hoặc nửa phải của vùng, tùy theo giá trị của phần tử giữa.

Ý tưởng trên có thể được cài đặt như sau:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Trong cài đặt này, vùng đang xét là $a \dots b$, và ban đầu vùng này là $0 \dots n-1$. Thuật toán chia đôi kích thước vùng ở mỗi bước, nên độ phức tạp thời gian là $O(\log n)$.

Phương pháp 2

Một phương pháp khác để cài đặt tìm kiếm nhị phân dựa trên một cách hiệu quả để duyệt qua các phần tử của mảng. Ý tưởng là thực hiện các bước nhảy và giảm tốc khi ta đến gần phần tử mục tiêu.

Tìm kiếm đi qua mảng từ trái sang phải, và độ dài bước nhảy ban đầu là $n/2$. Ở mỗi bước, độ dài bước nhảy sẽ được chia đôi: đầu tiên $n/4$, rồi $n/8$, $n/16$, v.v., cho đến khi cuối cùng độ dài là 1. Sau các bước nhảy, hoặc phần tử mục tiêu đã được tìm thấy, hoặc ta biết rằng nó không xuất hiện trong mảng.

Đoạn mã sau cài đặt ý tưởng trên:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
```

```

    // x found at index k
}

```

Trong quá trình tìm kiếm, biến b chứa độ dài bước nhảy hiện tại. Độ phức tạp thời gian của thuật toán là $O(\log n)$, vì đoạn mã trong vòng lặp `while` được thực hiện nhiều nhất hai lần cho mỗi độ dài bước nhảy.

Hàm C++

Thư viện chuẩn C++ chứa các hàm sau dựa trên tìm kiếm nhị phân và chạy trong thời gian logarithmic:

- `lower_bound` trả về một con trỏ tới phần tử đầu tiên của mảng có giá trị ít nhất là x .
- `upper_bound` trả về một con trỏ tới phần tử đầu tiên của mảng có giá trị lớn hơn x .
- `equal_range` trả về cả hai con trỏ trên.

Các hàm giả sử rằng mảng đã được sắp xếp. Nếu không có phần tử như vậy, con trỏ trỏ tới vị trí sau phần tử cuối cùng của mảng. Ví dụ, đoạn mã sau xác định xem một mảng có chứa phần tử giá trị x hay không:

```

auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x found at index k
}

```

Sau đó, đoạn mã sau đếm số phần tử có giá trị bằng x :

```

auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";

```

Dùng `equal_range`, đoạn mã trở nên ngắn hơn:

```

auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";

```

Tìm nghiệm nhỏ nhất

Một ứng dụng quan trọng của tìm kiếm nhị phân là tìm vị trí mà giá trị của một hàm thay đổi. Giả sử ta muốn tìm giá trị nhỏ nhất k là một nghiệm hợp lệ cho một bài toán. Ta được cho một hàm $ok(x)$ trả về `true` nếu x là nghiệm hợp lệ và `false` trong trường hợp ngược lại. Ngoài ra, ta biết rằng $ok(x)$ là `false` khi $x < k$ và `true` khi $x \geq k$. Tình huống trông như sau:

x	0	1	$\dots$	$k-1$	k	$k+1$	$\dots$
$\text{ok}(x)$	false	false	$\dots$	false	true	true	$\dots$

Bây giờ, giá trị của k có thể được tìm bằng tìm kiếm nhị phân:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;
```

Tìm kiếm tìm giá trị lớn nhất của x mà với nó $\text{ok}(x)$ là false. Vì vậy, giá trị kế tiếp $k = x + 1$ là giá trị nhỏ nhất có thể mà với nó $\text{ok}(k)$ là true. Độ dài bước nhảy ban đầu z phải đủ lớn, chẳng hạn một giá trị mà ta biết trước rằng $\text{ok}(z)$ là true.

Thuật toán gọi hàm ok $O(\log z)$ lần, nên độ phức tạp thời gian tổng cộng phụ thuộc vào hàm ok . Ví dụ, nếu hàm chạy trong $O(n)$ thời gian, độ phức tạp thời gian tổng cộng là $O(n \log z)$.

Tìm giá trị lớn nhất

Tìm kiếm nhị phân cũng có thể được dùng để tìm giá trị lớn nhất của một hàm tăng trước rồi giảm sau. Nhiệm vụ của ta là tìm một vị trí k sao cho

- $f(x) < f(x+1)$ khi $x < k$, và
- $f(x) > f(x+1)$ khi $x \geq k$.

Ý tưởng là dùng tìm kiếm nhị phân để tìm giá trị lớn nhất của x mà với nó $f(x) < f(x+1)$. Điều này suy ra $k = x+1$ vì $f(x+1) > f(x+2)$. Đoạn mã sau cài đặt tìm kiếm:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Lưu ý rằng khác với tìm kiếm nhị phân thông thường, ở đây không được phép có hai giá trị liên tiếp của hàm bằng nhau. Trong trường hợp đó, ta sẽ không thể biết nên tiếp tục tìm kiếm như thế nào.

Chương 4

Cấu trúc dữ liệu

Một **cấu trúc dữ liệu** (Data Structure) là một cách lưu trữ dữ liệu trong bộ nhớ của máy tính. Việc chọn một cấu trúc dữ liệu phù hợp cho một bài toán là rất quan trọng, vì mỗi cấu trúc dữ liệu có ưu điểm và nhược điểm riêng. Câu hỏi then chốt là: những thao tác nào hiệu quả trong cấu trúc dữ liệu đã chọn?

Chương này giới thiệu các cấu trúc dữ liệu quan trọng nhất trong thư viện chuẩn C++. Nên dùng thư viện chuẩn bất cứ khi nào có thể, vì điều đó sẽ tiết kiệm rất nhiều thời gian. Về sau trong sách, ta sẽ học các cấu trúc dữ liệu tinh vi hơn không có sẵn trong thư viện chuẩn.

4.1 Mảng động

Một **mảng động** là một mảng mà kích thước có thể thay đổi trong quá trình chương trình thực thi. Mảng động phổ biến nhất trong C++ là cấu trúc vector, có thể được dùng gần giống như một mảng thông thường.

Đoạn mã sau tạo một vector rỗng và thêm ba phần tử vào nó:

```
vector<int> v;  
v.push_back(3); // [3]  
v.push_back(2); // [3,2]  
v.push_back(5); // [3,2,5]
```

Sau đó, có thể truy cập các phần tử như trong một mảng thông thường:

```
cout << v[0] << "\n"; // 3  
cout << v[1] << "\n"; // 2  
cout << v[2] << "\n"; // 5
```

Hàm `size` trả về số phần tử trong vector. Đoạn mã sau duyệt qua vector và in tất cả phần tử trong đó:

```
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```

Một cách ngắn hơn để duyệt qua vector là như sau:

```
for (auto x : v) {  
    cout << x << "\n";  
}
```

Hàm `back` trả về phần tử cuối cùng trong vector, và hàm `pop_back` xóa phần tử cuối cùng:

```
vector<int> v;  
v.push_back(5);  
v.push_back(2);  
cout << v.back() << "\n"; // 2  
v.pop_back();  
cout << v.back() << "\n"; // 5
```

Đoạn mã sau tạo một vector có năm phần tử:

```
vector<int> v = {2,4,2,5,1};
```

Một cách khác để tạo vector là cung cấp số phần tử và giá trị ban đầu cho mỗi phần tử:

```
// size 10, initial value 0  
vector<int> v(10);
```

```
// size 10, initial value 5  
vector<int> v(10, 5);
```

Cài đặt bên trong của vector dùng một mảng thông thường. Nếu kích thước của vector tăng lên và mảng trở nên quá nhỏ, một mảng mới được cấp phát và tất cả phần tử được chuyển sang mảng mới. Tuy nhiên, điều này không xảy ra thường xuyên và độ phức tạp thời gian trung bình của `push_back` là $O(1)$.

Cấu trúc string cũng là một mảng động có thể được dùng gần giống như vector. Ngoài ra, có cú pháp đặc biệt cho xâu không có trong các cấu trúc dữ liệu khác. Các xâu có thể được nối bằng ký hiệu `+`. Hàm `substr( $k, x$ )` trả về xâu con bắt đầu tại vị trí k và có độ dài x , và hàm `find( $t$ )` tìm vị trí của lần xuất hiện đầu tiên của xâu con t .

Đoạn mã sau trình bày một số thao tác với xâu:

```
string a = "hatti";  
string b = a+a;  
cout << b << "\n"; // hattihatti  
b[5] = 'v';  
cout << b << "\n"; // hattivatti  
string c = b.substr(3,4);
```

```
cout << c << "\n"; // tiva
```

4.2 Cấu trúc tập hợp

Một **tập hợp** là một cấu trúc dữ liệu duy trì một tập các phần tử. Các thao tác cơ bản của tập hợp là thêm phần tử, tìm kiếm và xóa.

Thư viện chuẩn C++ chứa hai cài đặt tập hợp: Cấu trúc `set` dựa trên một cây nhị phân cân bằng và các thao tác của nó chạy trong $O(\log n)$ thời gian. Cấu trúc `unordered_set` dùng băm, và các thao tác của nó chạy trong $O(1)$ thời gian trung bình.

Việc chọn cài đặt tập hợp nào để dùng thường là vấn đề sở thích. Lợi ích của cấu trúc `set` là nó duy trì thứ tự của các phần tử và cung cấp các hàm không có trong `unordered_set`. Mặt khác, `unordered_set` có thể hiệu quả hơn.

Đoạn mã sau tạo một tập hợp chứa các số nguyên, và minh họa một số thao tác. Hàm `insert` thêm một phần tử vào tập hợp, hàm `count` trả về số lần xuất hiện của một phần tử trong tập hợp, và hàm `erase` xóa một phần tử khỏi tập hợp.

```
set<int> s;  
s.insert(3);  
s.insert(2);  
s.insert(5);  
cout << s.count(3) << "\n"; // 1  
cout << s.count(4) << "\n"; // 0  
s.erase(3);  
s.insert(4);  
cout << s.count(3) << "\n"; // 0  
cout << s.count(4) << "\n"; // 1
```

Một tập hợp có thể được dùng gần giống như vector, nhưng không thể truy cập các phần tử bằng ký hiệu `[]`. Đoạn mã sau tạo một tập hợp, in số phần tử trong đó, rồi duyệt qua tất cả phần tử:

```
set<int> s = {2,5,6,8};  
cout << s.size() << "\n"; // 4  
for (auto x : s) {  
    cout << x << "\n";  
}
```

Một tính chất quan trọng của tập hợp là tất cả phần tử của chúng đều *phân biệt*. Vì vậy, hàm `count` luôn trả về hoặc 0 (phần tử không có trong tập hợp) hoặc 1 (phần tử có trong tập hợp), và hàm `insert` không bao giờ thêm một phần tử vào tập hợp nếu phần tử đó đã có sẵn. Đoạn mã sau minh họa điều này:

```
set<int> s;
```

```
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ cũng chứa các cấu trúc `multiset` và `unordered_multiset`, hoạt động giống `set` và `unordered_set` nhưng có thể chứa nhiều bản sao của một phần tử. Ví dụ, trong đoạn mã sau, cả ba bản sao của số 5 đều được thêm vào một `multiset`:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Hàm `erase` xóa tất cả bản sao của một phần tử khỏi `multiset`:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Thông thường, chỉ nên xóa một bản sao, việc này có thể làm như sau:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

4.3 Cấu trúc ánh xạ

Một **map** là một mảng tổng quát hóa gồm các cặp khóa-giá trị. Trong khi khóa trong một mảng thông thường luôn là các số nguyên liên tiếp $0, 1, \dots, n-1$, trong đó n là kích thước của mảng, khóa trong một `map` có thể thuộc bất kỳ kiểu dữ liệu nào và không cần là các giá trị liên tiếp.

Thư viện chuẩn C++ chứa hai cài đặt `map` tương ứng với các cài đặt `set`: cấu trúc `map` dựa trên một cây nhị phân cân bằng và truy cập phần tử mất $O(\log n)$ thời gian, trong khi cấu trúc `unordered_map` dùng băm và truy cập phần tử mất $O(1)$ thời gian trung bình.

Đoạn mã sau tạo một `map` mà các khóa là xâu và các giá trị là số nguyên:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Nếu giá trị của một khóa được yêu cầu nhưng `map` không chứa khóa đó, khóa sẽ tự động được thêm vào `map` với một giá trị mặc định. Ví dụ, trong đoạn mã sau, khóa “aybabbu” với giá trị 0 được thêm vào `map`.

```
map<string,int> m;  
cout << m["aybabbu"] << "\n"; // 0
```

Hàm `count` kiểm tra một khóa có tồn tại trong map hay không:

```
if (m.count("aybabbu")) {  
    // key exists  
}
```

Đoạn mã sau in tất cả khóa và giá trị trong một map:

```
for (auto x : m) {  
    cout << x.first << " " << x.second << "\n";  
}
```

4.4 Iterator và đoạn

Nhiều hàm trong thư viện chuẩn C++ làm việc với iterator. Một **iterator** là một biến trỏ tới một phần tử trong cấu trúc dữ liệu.

Các iterator thường dùng `begin` và `end` định nghĩa một đoạn chứa tất cả phần tử trong một cấu trúc dữ liệu. Iterator `begin` trỏ tới phần tử đầu tiên trong cấu trúc dữ liệu, và iterator `end` trỏ tới vị trí *sau* phần tử cuối cùng. Tình huống trông như sau:

```
    { 3, 4, 6, 8, 12, 13, 14, 17 }  
      ↑                               ↑  
    s.begin()                       s.end()
```

Lưu ý sự bất đối xứng của các iterator: `s.begin()` trỏ tới một phần tử trong cấu trúc dữ liệu, trong khi `s.end()` trỏ ra ngoài cấu trúc dữ liệu. Vì vậy, đoạn được định nghĩa bởi các iterator là *nửa mở*.

Làm việc với đoạn

Iterator được dùng trong các hàm thư viện chuẩn C++ nhận một đoạn các phần tử trong một cấu trúc dữ liệu. Thông thường, ta muốn xử lý tất cả phần tử trong một cấu trúc dữ liệu, nên các iterator `begin` và `end` được truyền cho hàm.

Ví dụ, đoạn mã sau sắp xếp một vector bằng hàm `sort`, sau đó đảo ngược thứ tự các phần tử bằng hàm `reverse`, và cuối cùng xáo trộn thứ tự các phần tử bằng hàm `random_shuffle`.

```
sort(v.begin(), v.end());  
reverse(v.begin(), v.end());  
random_shuffle(v.begin(), v.end());
```

Các hàm này cũng có thể được dùng với một mảng thông thường. Trong trường hợp này, các hàm nhận con trỏ tới mảng thay vì iterator:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Iterator của set

Iterator thường được dùng để truy cập các phần tử của set. Đoạn mã sau tạo một iterator `it` trỏ tới phần tử nhỏ nhất trong một set:

```
set<int>::iterator it = s.begin();
```

Một cách viết ngắn hơn là như sau:

```
auto it = s.begin();
```

Phần tử mà iterator trỏ tới có thể được truy cập bằng ký hiệu `*`. Ví dụ, đoạn mã sau in phần tử đầu tiên trong set:

```
auto it = s.begin();
cout << *it << "\n";
```

Iterator có thể được di chuyển bằng các toán tử `++` (tiến) và `--` (lùi), nghĩa là iterator di chuyển tới phần tử kế tiếp hoặc phần tử trước đó trong set.

Đoạn mã sau in tất cả phần tử theo thứ tự tăng dần:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Đoạn mã sau in phần tử lớn nhất trong set:

```
auto it = s.end(); it--;
cout << *it << "\n";
```

Hàm `find(x)` trả về một iterator trỏ tới một phần tử có giá trị x . Tuy nhiên, nếu set không chứa x , iterator sẽ là `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

Hàm `lower_bound(x)` trả về một iterator tới phần tử nhỏ nhất trong set có giá trị *ít nhất* là x , và hàm `upper_bound(x)` trả về một iterator tới phần tử nhỏ nhất trong set có giá trị *lớn hơn* x . Trong cả hai hàm, nếu phần tử như vậy không tồn tại, giá trị trả về là `end`. Các hàm này không được cấu trúc `unordered_set` hỗ trợ, vì cấu trúc này không duy trì thứ tự của các phần tử.

Ví dụ, đoạn mã sau tìm phần tử gần x nhất:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}
```

Đoạn mã giả sử rằng set không rỗng, và xét tất cả trường hợp có thể bằng iterator it . Đầu tiên, iterator trở tới phần tử nhỏ nhất có giá trị ít nhất là x . Nếu it bằng $begin$, phần tử tương ứng là gần x nhất. Nếu it bằng end , phần tử lớn nhất trong set là gần x nhất. Nếu không thuộc các trường hợp trên, phần tử gần x nhất hoặc là phần tử tương ứng với it , hoặc là phần tử trước đó.

4.5 Cấu trúc khác

Bitset

Một **bitset** là một mảng mà mỗi giá trị là 0 hoặc 1. Ví dụ, đoạn mã sau tạo một **bitset** chứa 10 phần tử:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Lợi ích của việc dùng **bitset** là chúng cần ít bộ nhớ hơn mảng thông thường, vì mỗi phần tử trong **bitset** chỉ dùng một bit bộ nhớ. Ví dụ, nếu n bit được lưu trong một mảng `int`, $32n$ bit bộ nhớ sẽ được dùng, nhưng một **bitset** tương ứng chỉ cần n bit bộ nhớ. Ngoài ra, các giá trị của **bitset** có thể được thao tác hiệu quả bằng các toán tử bit, điều này cho phép tối ưu hóa thuật toán bằng **bitset**.

Đoạn mã sau cho thấy một cách khác để tạo **bitset** ở trên:

```
bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Hàm `count` trả về số lượng bit 1 trong `bitset`:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

Đoạn mã sau cho thấy các ví dụ dùng toán tử bit:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Deque

Một **deque** là một mảng động mà kích thước có thể được thay đổi hiệu quả ở cả hai đầu của mảng. Giống như `vector`, deque cung cấp các hàm `push_back` và `pop_back`, nhưng nó cũng có các hàm `push_front` và `pop_front` không có trong `vector`.

Deque có thể được dùng như sau:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

Cài đặt bên trong của deque phức tạp hơn của `vector`, và vì lý do này, deque chậm hơn `vector`. Dù vậy, cả thêm và xóa phần tử đều mất $O(1)$ thời gian trung bình ở cả hai đầu.

Stack

Một **stack** là một cấu trúc dữ liệu cung cấp hai thao tác thời gian $O(1)$: thêm một phần tử vào đỉnh, và xóa một phần tử khỏi đỉnh. Chỉ có thể truy cập phần tử ở đỉnh của stack.

Đoạn mã sau cho thấy cách dùng stack:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Queue

Một **queue** cũng cung cấp hai thao tác thời gian $O(1)$: thêm một phần tử vào cuối hàng đợi, và xóa phần tử đầu tiên trong hàng đợi. Chỉ có thể truy cập phần tử đầu tiên và phần tử cuối cùng của queue.

Đoạn mã sau cho thấy cách dùng queue:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Priority queue

Một **priority queue** duy trì một tập các phần tử. Các thao tác được hỗ trợ là thêm phần tử và, tùy theo loại hàng đợi, lấy và xóa phần tử nhỏ nhất hoặc lớn nhất. Thêm và xóa mất $O(\log n)$ thời gian, còn lấy phần tử mất $O(1)$ thời gian.

Trong khi một set có thứ tự hỗ trợ hiệu quả tất cả thao tác của priority queue, lợi ích của việc dùng priority queue là nó có các hệ số hằng nhỏ hơn. Priority queue thường được cài đặt bằng cấu trúc heap, đơn giản hơn nhiều so với cây nhị phân cân bằng dùng trong set có thứ tự.

Theo mặc định, các phần tử trong một priority queue của C++ được sắp xếp theo thứ tự giảm dần, và có thể tìm và xóa phần tử lớn nhất trong hàng đợi. Đoạn mã sau minh họa điều này:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Nếu ta muốn tạo một priority queue hỗ trợ tìm và xóa phần tử nhỏ nhất, ta có thể làm như sau:

```
priority_queue<int,vector<int>,greater<int>> q;
```

Cấu trúc dữ liệu policy-based

Trình biên dịch g++ cũng hỗ trợ một số cấu trúc dữ liệu không thuộc thư viện chuẩn C++. Những cấu trúc như vậy được gọi là cấu trúc dữ liệu *policy-based*. Để dùng các cấu trúc này, cần thêm các dòng sau vào mã:

```
#include <ext/pb_ds/assoc_container.hpp>  
using namespace __gnu_pbds;
```

Sau đó, ta có thể định nghĩa một cấu trúc dữ liệu `indexed_set` giống `set` nhưng có thể được đánh chỉ số như một mảng. Định nghĩa cho các giá trị `int` như sau:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,  
            tree_order_statistics_node_update> indexed_set;
```

Bây giờ ta có thể tạo một set như sau:

```
indexed_set s;  
s.insert(2);  
s.insert(3);  
s.insert(7);  
s.insert(9);
```

Điểm đặc biệt của set này là ta có quyền truy cập các chỉ số mà các phần tử sẽ có trong một mảng đã sắp xếp. Hàm `find_by_order` trả về một iterator tới phần tử ở vị trí cho trước:

```
auto x = s.find_by_order(2);  
cout << *x << "\n"; // 7
```

Và hàm `order_of_key` trả về vị trí của một phần tử cho trước:

```
cout << s.order_of_key(7) << "\n"; // 2
```

Nếu phần tử không xuất hiện trong set, ta nhận được vị trí mà phần tử đó sẽ có trong set:

```
cout << s.order_of_key(6) << "\n"; // 2  
cout << s.order_of_key(8) << "\n"; // 3
```

Cả hai hàm đều chạy trong thời gian logarithmic.

4.6 So sánh với sắp xếp

Thường có thể giải một bài toán bằng cấu trúc dữ liệu hoặc bằng sắp xếp. Đôi khi có những khác biệt đáng kể về hiệu quả thực tế giữa các cách tiếp cận này, có thể bị che khuất bởi độ phức tạp thời gian.

Hãy xét một bài toán mà ta được cho hai danh sách A và B đều chứa n phần tử. Nhiệm vụ của ta là tính số phần tử thuộc cả hai danh sách. Ví dụ, với các danh sách

$$A = [5, 2, 8, 9] \quad \text{và} \quad B = [3, 2, 9, 5],$$

đáp án là 3 vì các số 2, 5 và 9 thuộc cả hai danh sách.

Một lời giải trực tiếp cho bài toán là duyệt qua mọi cặp phần tử trong $O(n^2)$ thời gian, nhưng tiếp theo ta sẽ tập trung vào các thuật toán hiệu quả hơn.

Thuật toán 1

Ta xây dựng một set gồm các phần tử xuất hiện trong A , và sau đó duyệt qua các phần tử của B rồi kiểm tra với mỗi phần tử xem nó cũng thuộc A hay không. Điều này hiệu quả vì các phần tử của A đang nằm trong một set. Dùng cấu trúc set, độ phức tạp thời gian của thuật toán là $O(n \log n)$.

Thuật toán 2

Không cần duy trì một set có thứ tự, nên thay vì cấu trúc set ta cũng có thể dùng cấu trúc `unordered_set`. Đây là một cách dễ dàng để làm thuật toán hiệu quả hơn, vì ta chỉ cần thay đổi cấu trúc dữ liệu nền. Độ phức tạp thời gian của thuật toán mới là $O(n)$.

Thuật toán 3

Thay vì cấu trúc dữ liệu, ta có thể dùng sắp xếp. Đầu tiên, ta sắp xếp cả hai danh sách A và B . Sau đó, ta duyệt qua cả hai danh sách cùng lúc và tìm các phần tử chung. Độ phức tạp thời gian của sắp xếp là $O(n \log n)$, và phần còn lại của thuật toán chạy trong $O(n)$ thời gian, nên độ phức tạp thời gian tổng cộng là $O(n \log n)$.

So sánh hiệu quả

Bảng sau cho thấy mức độ hiệu quả của các thuật toán trên khi n thay đổi và các phần tử của danh sách là các số nguyên ngẫu nhiên trong khoảng $1 \dots 10^9$:

n	Thuật toán 1	Thuật toán 2	Thuật toán 3
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

Thuật toán 1 và 2 giống nhau ngoại trừ việc chúng dùng các cấu trúc set khác nhau. Trong bài toán này, lựa chọn này ảnh hưởng quan trọng đến thời gian chạy, vì Thuật toán 2 nhanh hơn Thuật toán 1 khoảng 4-5 lần.

Tuy nhiên, thuật toán hiệu quả nhất là Thuật toán 3, dùng sắp xếp. Nó chỉ dùng một nửa thời gian so với Thuật toán 2. Điều thú vị là độ phức tạp thời gian của cả Thuật toán 1 và Thuật toán 3 đều là $O(n \log n)$, nhưng dù vậy, Thuật toán 3 nhanh hơn mười lần. Điều này có thể được giải thích bởi thực tế rằng sắp xếp là một thủ tục đơn giản và chỉ được thực hiện một lần ở đầu Thuật toán 3, và phần còn lại của thuật toán chạy trong thời gian tuyến tính. Mặt khác, Thuật toán 1 duy trì một cây nhị phân cân bằng phức tạp trong suốt toàn bộ thuật toán.

Chương 5

Tìm kiếm toàn bộ (Complete Search)

Complete search là một phương pháp tổng quát có thể dùng để giải hầu như mọi bài toán thuật toán. Ý tưởng là sinh tất cả các lời giải có thể của bài toán bằng vét cạn, rồi chọn lời giải tốt nhất hoặc đếm số lời giải, tùy theo bài toán.

Tìm kiếm toàn bộ là một kỹ thuật tốt nếu có đủ thời gian để duyệt qua mọi lời giải, vì việc tìm kiếm thường dễ cài đặt và luôn cho đáp án đúng. Nếu tìm kiếm toàn bộ quá chậm, có thể cần dùng các kỹ thuật khác, chẳng hạn thuật toán tham lam hoặc quy hoạch động.

5.1 Sinh tập con

Trước tiên ta xét bài toán sinh tất cả các tập con của một tập có n phần tử. Ví dụ, các tập con của $\{0, 1, 2\}$ là $\emptyset$, $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ và $\{0, 1, 2\}$. Có hai phương pháp phổ biến để sinh tập con: ta có thể thực hiện tìm kiếm đệ quy hoặc khai thác biểu diễn bit của số nguyên.

Phương pháp 1

Một cách gọn gàng để duyệt qua mọi tập con của một tập là dùng đệ quy. Hàm `search` sau đây sinh các tập con của tập $\{0, 1, \dots, n-1\}$. Hàm duy trì một vector `subset` sẽ chứa các phần tử của từng tập con. Quá trình tìm kiếm bắt đầu khi hàm được gọi với tham số 0.

```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

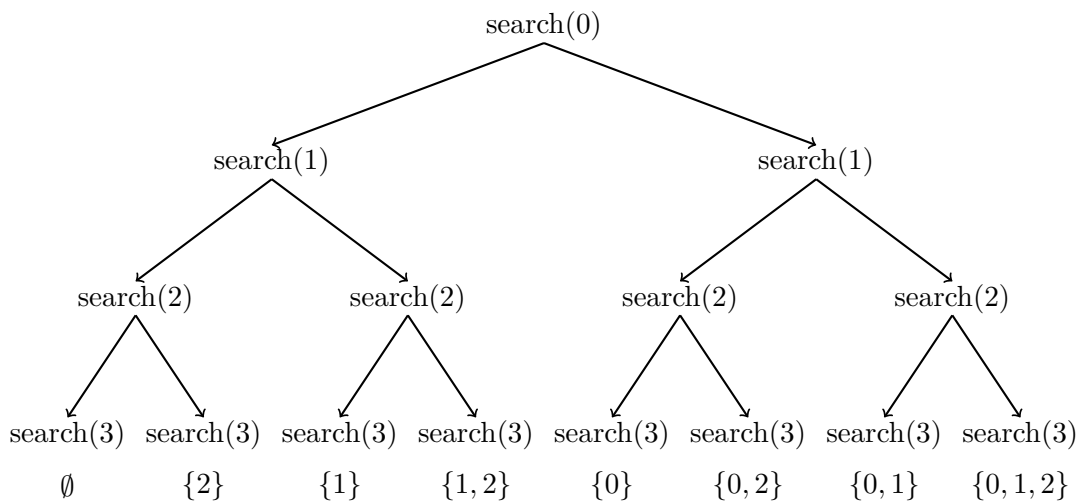
```

    }
}

```

Khi hàm `search` được gọi với tham số k , nó quyết định có đưa phần tử k vào tập con hay không, và trong cả hai trường hợp đều gọi lại chính nó với tham số $k + 1$. Tuy nhiên, nếu $k = n$, hàm nhận ra rằng tất cả các phần tử đã được xử lý và một tập con đã được sinh ra.

Cây sau minh họa các lời gọi hàm khi $n = 3$. Ta luôn có thể chọn nhánh trái (k không được đưa vào tập con) hoặc nhánh phải (k được đưa vào tập con).



Phương pháp 2

Một cách khác để sinh tập con dựa trên biểu diễn bit của số nguyên. Mỗi tập con của một tập có n phần tử có thể được biểu diễn bằng một dãy n bit, tương ứng với một số nguyên từ $0 \dots 2^n - 1$. Các bit 1 trong dãy bit cho biết những phần tử nào được đưa vào tập con.

Quy ước thường dùng là bit cuối cùng tương ứng với phần tử 0, bit kế cuối tương ứng với phần tử 1, và cứ tiếp tục như vậy. Ví dụ, biểu diễn bit của 25 là 11001, tương ứng với tập con $\{0, 3, 4\}$.

Đoạn mã sau duyệt qua các tập con của một tập có n phần tử

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}

```

Đoạn mã sau cho thấy cách tìm các phần tử của tập con tương ứng với một dãy bit. Khi xử lý từng tập con, đoạn mã xây dựng một vector chứa các phần tử trong tập con đó.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {

```



```

        if (b & (1 < i)) subset.push_back(i);
    }
}

```

5.2 Sinh hoán vị

Tiếp theo ta xét bài toán sinh tất cả các hoán vị của một tập có n phần tử. Ví dụ, các hoán vị của $\{0, 1, 2\}$ là $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ và $(2, 1, 0)$. Một lần nữa, có hai hướng tiếp cận: ta có thể dùng đệ quy hoặc duyệt qua các hoán vị theo cách lặp.

Phương pháp 1

Giống như tập con, hoán vị có thể được sinh bằng đệ quy. Hàm `search` sau đây duyệt qua các hoán vị của tập $\{0, 1, \dots, n-1\}$. Hàm xây dựng một vector `permutation` chứa hoán vị hiện tại, và quá trình tìm kiếm bắt đầu khi hàm được gọi không có tham số.

```

void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}

```

Mỗi lời gọi hàm thêm một phần tử mới vào `permutation`. Mảng `chosen` cho biết những phần tử nào đã có trong hoán vị. Nếu kích thước của `permutation` bằng kích thước của tập, một hoán vị đã được sinh ra.

Phương pháp 2

Một phương pháp khác để sinh hoán vị là bắt đầu với hoán vị $\{0, 1, \dots, n-1\}$ và liên tục dùng một hàm xây dựng hoán vị kế tiếp theo thứ tự tăng dần. Thư viện chuẩn C++ có hàm `next_permutation` có thể dùng cho việc này:

```

vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}

```

```

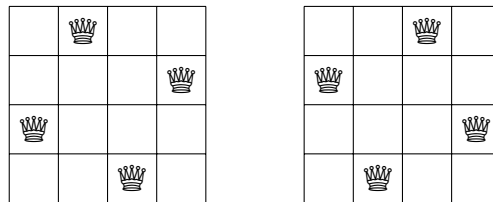
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));

```

5.3 Backtracking

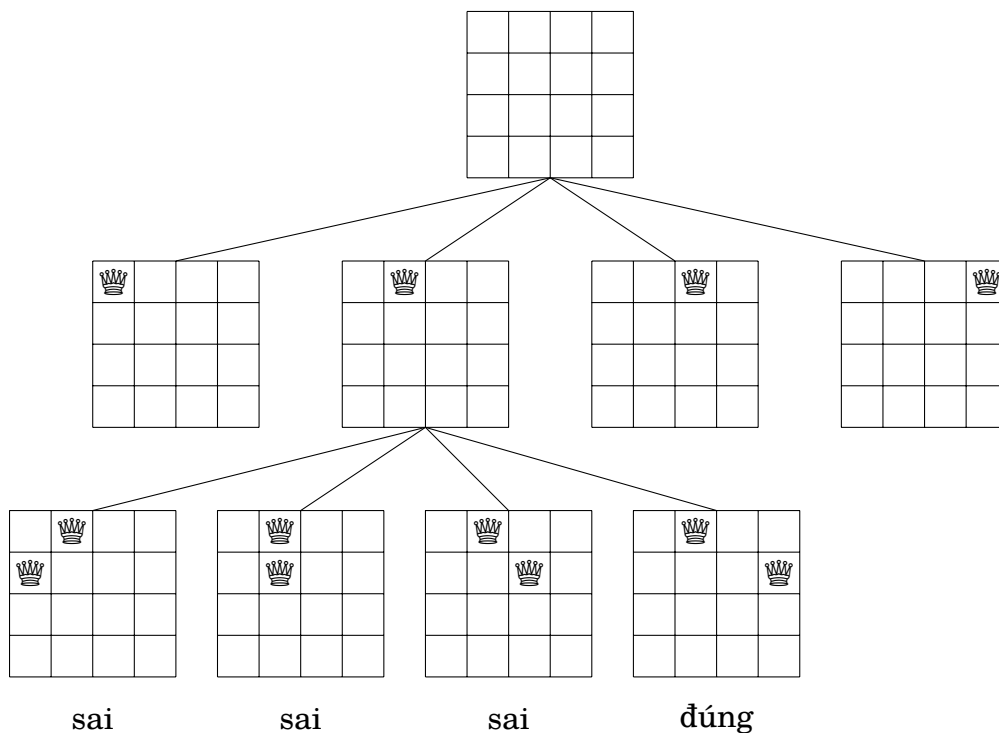
Thuật toán **backtracking** bắt đầu với một lời giải rỗng và mở rộng lời giải từng bước. Quá trình tìm kiếm duyệt đệ quy qua mọi cách khác nhau để xây dựng một lời giải.

Làm ví dụ, hãy xét bài toán tính số cách đặt n quân hậu trên bàn cờ $n \times n$ sao cho không có hai quân hậu nào tấn công nhau. Ví dụ, khi $n = 4$, có hai lời giải khả dĩ:



Bài toán có thể được giải bằng quay lui bằng cách đặt các quân hậu lên bàn cờ theo từng hàng. Cụ thể hơn, trên mỗi hàng sẽ đặt đúng một quân hậu sao cho nó không tấn công bất kỳ quân hậu nào đã được đặt trước đó. Ta tìm được một lời giải khi tất cả n quân hậu đã được đặt lên bàn cờ.

Ví dụ, khi $n = 4$, một số lời giải bộ phận được sinh bởi thuật toán quay lui như sau:



Ở tầng dưới cùng, ba cấu hình đầu là không hợp lệ, vì các quân hậu tấn công nhau. Tuy nhiên, cấu hình thứ tư là hợp lệ và có thể được mở rộng thành một lời giải hoàn chỉnh bằng cách đặt thêm hai quân hậu lên bàn cờ. Chỉ có một cách để đặt hai quân hậu còn lại.

Thuật toán có thể được cài đặt như sau:

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

Quá trình tìm kiếm bắt đầu bằng lời gọi `search(0)`. Kích thước bàn cờ là $n \times n$, và đoạn mã tính số lời giải vào biến `count`.

Đoạn mã giả sử rằng các hàng và cột của bàn cờ được đánh số từ 0 đến $n - 1$. Khi hàm `search` được gọi với tham số y , nó đặt một quân hậu trên hàng y rồi gọi lại chính nó với tham số $y + 1$. Sau đó, nếu $y = n$, một lời giải đã được tìm thấy và biến `count` được tăng thêm một.

Mảng `column` theo dõi các cột đã chứa quân hậu, còn các mảng `diag1` và `diag2` theo dõi các đường chéo. Không được thêm một quân hậu khác vào một cột hoặc đường chéo đã chứa quân hậu. Ví dụ, các cột và đường chéo của bàn cờ 4×4 được đánh số như sau:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

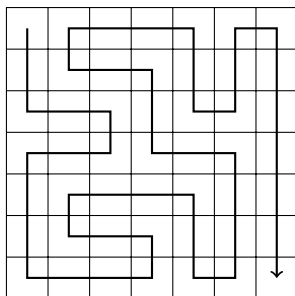
Gọi $q(n)$ là số cách đặt n quân hậu trên bàn cờ $n \times n$. Thuật toán quay lui ở trên cho ta biết rằng, chẳng hạn, $q(8) = 92$. Khi n tăng, việc tìm kiếm nhanh chóng trở nên chậm, vì số lời giải tăng theo cấp số nhân. Ví dụ, tính $q(16) = 14772512$ bằng thuật toán trên đã mất khoảng một phút trên một máy tính hiện đại¹.

¹Chưa có cách nào được biết để tính hiệu quả các giá trị lớn hơn của $q(n)$. Kỷ lục hiện tại là $q(27) = 234907967154122528$, được tính vào năm 2016 [55].

5.4 Cắt tỉa tìm kiếm

Ta thường có thể tối ưu quay lui bằng cách cắt tỉa cây tìm kiếm. Ý tưởng là thêm “sự thông minh” vào thuật toán để nó nhận ra sớm nhất có thể khi một lời giải bộ phận không thể được mở rộng thành lời giải hoàn chỉnh. Những tối ưu như vậy có thể có ảnh hưởng rất lớn đến hiệu quả của tìm kiếm.

Hãy xét bài toán tính số đường đi trong lưới $n \times n$ từ góc trên bên trái đến góc dưới bên phải sao cho đường đi thăm mỗi ô đúng một lần. Ví dụ, trong lưới 7×7 , có 111712 đường đi như vậy. Một trong các đường đi là:



Ta tập trung vào trường hợp 7×7 , vì độ khó của nó phù hợp với nhu cầu của ta. Ta bắt đầu với một thuật toán quay lui trực tiếp, rồi tối ưu nó từng bước bằng các nhận xét về cách cắt tỉa tìm kiếm. Sau mỗi tối ưu, ta đo thời gian chạy của thuật toán và số lời gọi đệ quy, để thấy rõ ảnh hưởng của từng tối ưu lên hiệu quả tìm kiếm.

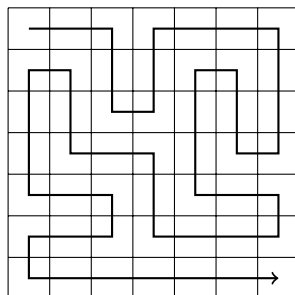
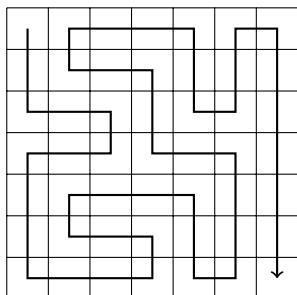
Thuật toán cơ bản

Phiên bản đầu tiên của thuật toán không có tối ưu nào. Ta chỉ dùng quay lui để sinh tất cả các đường đi có thể từ góc trên bên trái đến góc dưới bên phải và đếm số đường đi đó.

- thời gian chạy: 483 giây
- số lời gọi đệ quy: 76 tỷ

Tối ưu 1

Trong mọi lời giải, trước tiên ta đi một bước xuống dưới hoặc sang phải. Luôn có hai đường đi đối xứng nhau qua đường chéo của lưới sau bước đầu tiên. Ví dụ, các đường đi sau là đối xứng:

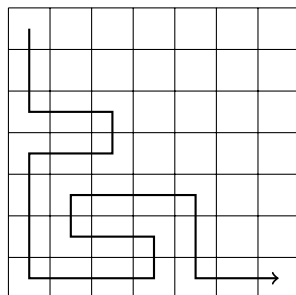


Vì vậy, ta có thể quy ước rằng trước tiên luôn đi một bước xuống dưới (hoặc sang phải), rồi cuối cùng nhân số lời giải với hai.

- thời gian chạy: 244 giây
- số lời gọi đệ quy: 38 tỷ

Tối ưu 2

Nếu đường đi đến ô dưới bên phải trước khi đã thăm tất cả các ô khác của lưới, rõ ràng sẽ không thể hoàn thành lời giải. Ví dụ cho trường hợp này là đường đi sau:

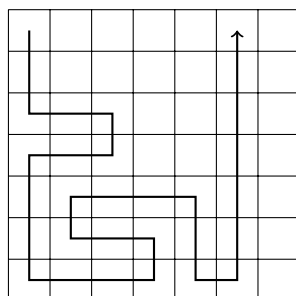


Dùng nhận xét này, ta có thể dừng tìm kiếm ngay lập tức nếu đến ô dưới bên phải quá sớm.

- thời gian chạy: 119 giây
- số lời gọi đệ quy: 20 tỷ

Tối ưu 3

Nếu đường đi chạm vào biên và có thể rẽ trái hoặc rẽ phải, lưới bị chia thành hai phần đều chứa các ô chưa thăm. Ví dụ, trong tình huống sau, đường đi có thể rẽ trái hoặc rẽ phải:

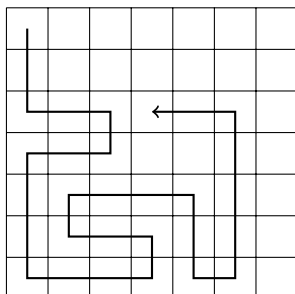


Trong trường hợp này, ta không thể thăm tất cả các ô nữa, vì vậy có thể dừng tìm kiếm. Tối ưu này rất hữu ích:

- thời gian chạy: 1.8 giây
- số lời gọi đệ quy: 221 triệu

Tối ưu 4

Ý tưởng của Tối ưu 3 có thể được tổng quát hóa: nếu đường đi không thể tiếp tục đi thẳng nhưng có thể rẽ trái hoặc rẽ phải, lưới bị chia thành hai phần đều chứa các ô chưa thăm. Ví dụ, xét đường đi sau:



Rõ ràng ta không thể thăm tất cả các ô nữa, vì vậy có thể dừng tìm kiếm. Sau tối ưu này, việc tìm kiếm rất hiệu quả:

- thời gian chạy: 0.6 giây
- số lời gọi đệ quy: 69 triệu

Bây giờ là thời điểm tốt để dùng tối ưu thuật toán và xem ta đã đạt được gì. Thời gian chạy của thuật toán ban đầu là 483 giây, còn sau các tối ưu, thời gian chạy chỉ còn 0.6 giây. Như vậy, thuật toán đã nhanh hơn gần 1000 lần sau các tối ưu.

Đây là hiện tượng thường gặp trong quay lui, vì cây tìm kiếm thường lớn và ngay cả các nhận xét đơn giản cũng có thể cắt tia tìm kiếm rất hiệu quả. Đặc biệt hữu ích là các tối ưu xuất hiện trong những bước đầu của thuật toán, tức ở phần trên của cây tìm kiếm.

5.5 Chia đôi tìm kiếm (Meet in the middle)

Meet in the middle là một kỹ thuật trong đó không gian tìm kiếm được chia thành hai phần có kích thước xấp xỉ nhau. Ta thực hiện tìm kiếm riêng cho từng phần, rồi cuối cùng kết hợp kết quả của các lần tìm kiếm.

Kỹ thuật này có thể dùng nếu có một cách hiệu quả để kết hợp kết quả tìm kiếm. Trong tình huống như vậy, hai lần tìm kiếm có thể cần ít thời gian hơn một lần tìm kiếm lớn. Thông thường, ta có thể biến một thừa số 2^n thành một thừa số $2^{n/2}$ bằng kỹ thuật meet in the middle.

Làm ví dụ, xét một bài toán trong đó ta được cho một danh sách n số và một số x , và muốn biết liệu có thể chọn một số số từ danh sách sao cho tổng của chúng là x hay không. Ví dụ, với danh sách $[2, 4, 5, 9]$ và $x = 15$, ta có thể chọn các số $[2, 4, 9]$ để được $2 + 4 + 9 = 15$. Tuy nhiên, nếu $x = 10$ với cùng danh sách đó, không thể tạo ra tổng này.

Một thuật toán đơn giản cho bài toán là duyệt qua mọi tập con của các phần tử và kiểm tra xem tổng của tập con nào đó có bằng x không. Thời gian chạy của thuật toán như vậy là $O(2^n)$, vì có 2^n tập con. Tuy nhiên, dùng kỹ thuật meet in the middle, ta có thể đạt được thuật toán hiệu quả hơn với thời gian $O(2^{n/2})^2$. Lưu ý rằng $O(2^n)$ và $O(2^{n/2})$ là các độ phức tạp khác nhau vì $2^{n/2}$ bằng $\sqrt{2^n}$.

Ý tưởng là chia danh sách thành hai danh sách A và B sao cho cả hai danh sách đều chứa khoảng một nửa số phần tử. Lần tìm kiếm thứ nhất sinh tất cả tập con của A và lưu tổng của chúng vào danh sách S_A . Tương tự, lần tìm kiếm thứ hai tạo danh sách S_B từ B . Sau đó, chỉ cần kiểm tra xem có thể chọn một phần tử từ S_A và một phần tử khác từ S_B sao cho tổng của chúng là x hay không. Điều này xảy ra đúng khi có một cách tạo tổng x bằng các số trong danh sách ban đầu.

Ví dụ, giả sử danh sách là $[2, 4, 5, 9]$ và $x = 15$. Đầu tiên, ta chia danh sách thành $A = [2, 4]$ và $B = [5, 9]$. Sau đó, ta tạo các danh sách $S_A = [0, 2, 4, 6]$ và $S_B = [0, 5, 9, 14]$. Trong trường hợp này, có thể tạo tổng $x = 15$, vì S_A chứa tổng 6, S_B chứa tổng 9, và $6 + 9 = 15$. Điều này tương ứng với lời giải $[2, 4, 9]$.

Ta có thể cài đặt thuật toán sao cho độ phức tạp thời gian là $O(2^{n/2})$. Trước tiên, ta sinh các danh sách *đã sắp xếp* S_A và S_B , có thể làm trong thời gian $O(2^{n/2})$ bằng một kỹ thuật giống trộn. Sau đó, vì các danh sách đã được sắp xếp, ta có thể kiểm tra trong thời gian $O(2^{n/2})$ xem tổng x có thể được tạo từ S_A và S_B hay không.

²Ý tưởng này được E. Horowitz và S. Sahni giới thiệu vào năm 1974 [39].

Chương 6

Thuật toán tham lam (Greedy Algorithms)

Thuật toán **tham lam** (**greedy algorithm**) là thuật toán xây dựng lời giải cho bài toán bằng cách luôn đưa ra lựa chọn có vẻ tốt nhất tại thời điểm hiện tại. Một thuật toán tham lam không bao giờ rút lại các lựa chọn của nó, mà trực tiếp xây dựng lời giải cuối cùng. Vì lý do này, các thuật toán tham lam thường rất hiệu quả.

Khó khăn khi thiết kế thuật toán tham lam là tìm một chiến lược tham lam luôn tạo ra lời giải tối ưu cho bài toán. Các lựa chọn tối ưu cục bộ trong một thuật toán tham lam cũng phải là tối ưu toàn cục. Thường rất khó chứng minh rằng một thuật toán tham lam là đúng.

6.1 Bài toán đồng xu

Làm ví dụ đầu tiên, ta xét một bài toán trong đó ta được cho một tập đồng xu và nhiệm vụ là tạo ra một tổng tiền n bằng các đồng xu đó. Giá trị của các đồng xu là $\text{coins} = \{c_1, c_2, \dots, c_k\}$, và mỗi đồng xu có thể được dùng bao nhiêu lần tùy ý. Cần ít nhất bao nhiêu đồng xu?

Ví dụ, nếu các đồng xu là các đồng euro (tính theo xu)

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

và $n = 520$, ta cần ít nhất bốn đồng xu. Lời giải tối ưu là chọn các đồng xu $200 + 200 + 100 + 20$ có tổng là 520.

Thuật toán tham lam

Một thuật toán tham lam đơn giản cho bài toán luôn chọn đồng xu lớn nhất có thể, cho đến khi tạo được tổng tiền cần thiết. Thuật toán này đúng trong ví dụ trên, vì trước tiên ta chọn hai đồng 200 xu, sau đó một đồng 100 xu và cuối cùng một đồng 20 xu. Nhưng thuật toán này có luôn đúng không?

Hóa ra nếu các đồng xu là đồng euro, thuật toán tham lam *luôn* đúng, tức là nó luôn tạo ra lời giải dùng ít đồng xu nhất có thể. Tính đúng đắn của thuật toán có thể được chứng minh như sau:

Thứ nhất, mỗi đồng 1, 5, 10, 50 và 100 xuất hiện nhiều nhất một lần trong một lời giải tối ưu, vì nếu lời giải chứa hai đồng như vậy, ta có thể thay chúng bằng một đồng xu và thu được lời giải tốt hơn. Ví dụ, nếu lời giải chứa các đồng 5 + 5, ta có thể thay bằng đồng 10.

Tương tự, các đồng 2 và 20 xuất hiện nhiều nhất hai lần trong một lời giải tối ưu, vì ta có thể thay các đồng 2 + 2 + 2 bằng 5 + 1 và các đồng 20 + 20 + 20 bằng 50 + 10. Hơn nữa, một lời giải tối ưu không thể chứa các đồng 2 + 2 + 1 hoặc 20 + 20 + 10, vì ta có thể thay chúng bằng các đồng 5 và 50.

Dựa trên các nhận xét này, ta có thể chứng minh với mỗi đồng xu x rằng không thể tạo tối ưu một tổng x hoặc lớn hơn chỉ bằng các đồng xu nhỏ hơn x . Ví dụ, nếu $x = 100$, tổng tối ưu lớn nhất dùng các đồng xu nhỏ hơn là $50 + 20 + 20 + 5 + 2 + 2 = 99$. Do đó, thuật toán tham lam luôn chọn đồng xu lớn nhất sẽ tạo ra lời giải tối ưu.

Ví dụ này cho thấy việc chứng minh một thuật toán tham lam là đúng có thể khó, ngay cả khi bản thân thuật toán rất đơn giản.

Trường hợp tổng quát

Trong trường hợp tổng quát, tập đồng xu có thể chứa bất kỳ giá trị nào và thuật toán tham lam *không* nhất thiết tạo ra lời giải tối ưu.

Ta có thể chứng minh một thuật toán tham lam không đúng bằng cách đưa ra một phản ví dụ trong đó thuật toán cho đáp án sai. Trong bài toán này ta dễ dàng tìm được phản ví dụ: nếu các đồng xu là $\{1, 3, 4\}$ và tổng mục tiêu là 6, thuật toán tham lam tạo ra lời giải $4 + 1 + 1$ trong khi lời giải tối ưu là $3 + 3$.

Chưa biết liệu bài toán đồng xu tổng quát có thể được giải bằng một thuật toán tham lam nào hay không¹. Tuy nhiên, như ta sẽ thấy trong Chương 7, trong một số trường hợp, bài toán tổng quát có thể được giải hiệu quả bằng thuật toán quy hoạch động luôn cho đáp án đúng.

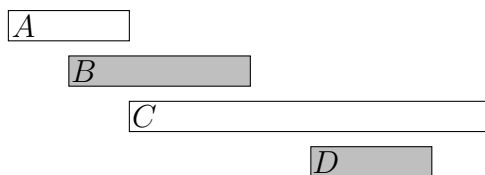
6.2 Lập lịch

Nhiều bài toán lập lịch có thể được giải bằng thuật toán tham lam. Một bài toán kinh điển như sau: Cho n sự kiện cùng thời điểm bắt đầu và kết thúc của chúng, hãy tìm một lịch chứa nhiều sự kiện nhất có thể. Không thể chọn một phần của một sự kiện. Ví dụ, xét các sự kiện sau:

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

¹Tuy nhiên, có thể *kiểm tra* trong thời gian đa thức xem thuật toán tham lam trình bày trong chương này có đúng với một tập đồng xu cho trước hay không [53].

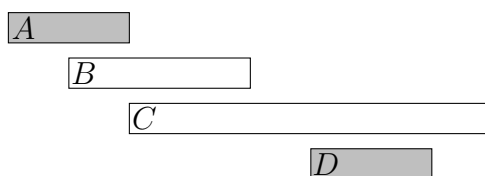
Trong trường hợp này, số sự kiện tối đa là hai. Ví dụ, ta có thể chọn các sự kiện B và D như sau:



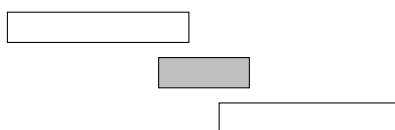
Có thể nghĩ ra nhiều thuật toán tham lam cho bài toán này, nhưng thuật toán nào đúng trong mọi trường hợp?

Thuật toán 1

Ý tưởng đầu tiên là chọn các sự kiện *ngắn* nhất có thể. Trong ví dụ, thuật toán này chọn các sự kiện sau:



Tuy nhiên, chọn các sự kiện ngắn không phải lúc nào cũng là chiến lược đúng. Ví dụ, thuật toán thất bại trong trường hợp sau:



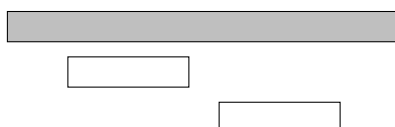
Nếu chọn sự kiện ngắn, ta chỉ có thể chọn một sự kiện. Tuy nhiên, vẫn có thể chọn cả hai sự kiện dài.

Thuật toán 2

Một ý tưởng khác là luôn chọn sự kiện tiếp theo có thể *bắt đầu sớm* nhất. Thuật toán này chọn các sự kiện sau:



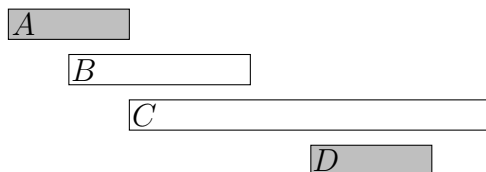
Tuy nhiên, ta cũng có thể tìm phản ví dụ cho thuật toán này. Ví dụ, trong trường hợp sau, thuật toán chỉ chọn một sự kiện:



Nếu chọn sự kiện đầu tiên, ta không thể chọn thêm sự kiện nào khác. Tuy nhiên, có thể chọn hai sự kiện còn lại.

Thuật toán 3

Ý tưởng thứ ba là luôn chọn sự kiện tiếp theo có thể *kết thúc sớm nhất*. Thuật toán này chọn các sự kiện sau:



Hóa ra thuật toán này *luôn* tạo ra lời giải tối ưu. Lý do là việc đầu tiên chọn một sự kiện kết thúc sớm nhất có thể luôn là một lựa chọn tối ưu. Sau đó, việc chọn sự kiện tiếp theo bằng cùng chiến lược cũng là tối ưu, và cứ tiếp tục như vậy cho đến khi không thể chọn thêm sự kiện nào.

Một cách chứng minh thuật toán đúng là xét điều gì xảy ra nếu trước tiên ta chọn một sự kiện kết thúc muộn hơn sự kiện kết thúc sớm nhất có thể. Khi đó, ta sẽ có nhiều nhất cùng số lựa chọn cho sự kiện tiếp theo. Vì vậy, chọn một sự kiện kết thúc muộn hơn không bao giờ cho lời giải tốt hơn, và thuật toán tham lam là đúng.

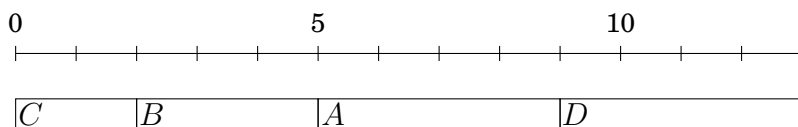
6.3 Công việc và hạn chót

Bây giờ hãy xét một bài toán trong đó ta được cho n công việc với thời lượng và hạn chót, và nhiệm vụ là chọn thứ tự thực hiện các công việc. Với mỗi công việc, ta nhận được $d - x$ điểm, trong đó d là hạn chót của công việc và x là thời điểm ta hoàn thành công việc. Tổng điểm lớn nhất có thể đạt được là bao nhiêu?

Ví dụ, giả sử các công việc như sau:

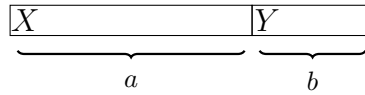
task	duration	deadline
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

Trong trường hợp này, một lịch tối ưu cho các công việc như sau:

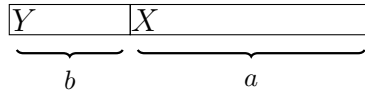


Trong lời giải này, *C* cho 5 điểm, *B* cho 0 điểm, *A* cho -7 điểm và *D* cho -8 điểm, nên tổng điểm là -10 .

Điều bất ngờ là lời giải tối ưu của bài toán hoàn toàn không phụ thuộc vào các hạn chót, mà một chiến lược tham lam đúng chỉ là thực hiện các công việc *theo thứ tự thời lượng tăng dần*. Lý do là nếu ta từng thực hiện hai công việc liên tiếp sao cho công việc đầu mất nhiều thời gian hơn công việc thứ hai, ta có thể có lời giải tốt hơn nếu đổi chỗ hai công việc. Ví dụ, xét lịch sau:



Ở đây $a > b$, nên ta nên đổi chỗ hai công việc:



Lúc này X cho ít hơn b điểm và Y cho nhiều hơn a điểm, nên tổng điểm tăng thêm $a - b > 0$. Trong một lời giải tối ưu, với mọi cặp công việc liên tiếp, công việc ngắn hơn phải đứng trước công việc dài hơn. Do đó, các công việc phải được thực hiện theo thứ tự thời lượng tăng dần.

6.4 Tối thiểu hóa tổng

Tiếp theo ta xét một bài toán trong đó ta được cho n số $a_1, a_2, \dots, a_n$ và nhiệm vụ là tìm một giá trị x làm nhỏ nhất tổng

$$|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c.$$

Ta tập trung vào các trường hợp $c = 1$ và $c = 2$.

Trường hợp $c = 1$

Trong trường hợp này, ta cần tối thiểu hóa tổng

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

Ví dụ, nếu các số là $[1, 2, 9, 2, 6]$, lời giải tốt nhất là chọn $x = 2$, tạo ra tổng

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

Trong trường hợp tổng quát, lựa chọn tốt nhất cho x là *trung vị* của các số, tức số ở giữa sau khi sắp xếp. Ví dụ, danh sách $[1, 2, 9, 2, 6]$ trở thành $[1, 2, 2, 6, 9]$ sau khi sắp xếp, nên trung vị là 2.

Trung vị là một lựa chọn tối ưu, vì nếu x nhỏ hơn trung vị, tổng sẽ nhỏ hơn khi tăng x , và nếu x lớn hơn trung vị, tổng sẽ nhỏ hơn khi giảm x . Vì vậy, lời giải tối ưu là x bằng trung vị. Nếu n chẵn và có hai trung vị, cả hai trung vị và mọi giá trị giữa chúng đều là lựa chọn tối ưu.

Trường hợp $c = 2$

Trong trường hợp này, ta cần tối thiểu hóa tổng

$$(a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2.$$

Ví dụ, nếu các số là $[1, 2, 9, 2, 6]$, lời giải tốt nhất là chọn $x = 4$, tạo ra tổng

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

Trong trường hợp tổng quát, lựa chọn tốt nhất cho x là *trung bình* của các số. Trong ví dụ, trung bình là $(1 + 2 + 9 + 2 + 6)/5 = 4$. Kết quả này có thể được suy ra bằng cách viết tổng như sau:

$$nx^2 - 2x(a_1 + a_2 + \dots + a_n) + (a_1^2 + a_2^2 + \dots + a_n^2)$$

Phần cuối không phụ thuộc vào x , nên ta có thể bỏ qua. Các phần còn lại tạo thành hàm $nx^2 - 2xs$, trong đó $s = a_1 + a_2 + \dots + a_n$. Đây là một parabol mở lên có các nghiệm $x = 0$ và $x = 2s/n$, và giá trị nhỏ nhất nằm tại trung bình của hai nghiệm, tức $x = s/n$, chính là trung bình của các số $a_1, a_2, \dots, a_n$.

6.5 Nén dữ liệu

A **binary code** assigns for each character trong một chuỗi một **codeword** gồm các bit. Ta có thể *nén* chuỗi bằng mã nhị phân bằng cách thay mỗi ký tự bằng codeword tương ứng. Ví dụ, mã nhị phân sau gán codeword cho các ký tự A–D:

character	codeword
A	00
B	01
C	10
D	11

Đây là mã **độ dài cố định**, nghĩa là độ dài của mọi codeword đều giống nhau. Ví dụ, ta có thể nén chuỗi AABACDACA bằng cách sau:

00 00 01 00 10 11 00 10 00

Dùng mã này, độ dài của chuỗi đã nén là 18 bit. Tuy nhiên, ta có thể nén chuỗi tốt hơn nếu dùng mã **độ dài biến đổi**, trong đó các codeword có thể có độ dài khác nhau. Khi đó ta có thể gán codeword ngắn cho các ký tự xuất hiện thường xuyên và codeword dài cho các ký tự hiếm khi xuất hiện. Hóa ra một mã **tối ưu** cho chuỗi trên là:

character	codeword
A	0
B	110
C	10
D	111

Một mã tối ưu tạo ra chuỗi nén ngắn nhất có thể. Trong trường hợp này, chuỗi nén dùng mã tối ưu là

00 1100 10 1110 100,

nên chỉ cần 15 bit thay vì 18 bit. Vì vậy, nhờ một mã tốt hơn, ta có thể tiết kiệm 3 bit trong chuỗi nén.

Ta yêu cầu không có codeword nào là tiền tố của một codeword khác. Ví dụ, một mã không được phép chứa đồng thời hai codeword 10 và 1011. Lý do là ta muốn có thể khôi phục chuỗi gốc từ chuỗi nén. Nếu một codeword có thể là tiền tố của codeword khác, điều này không phải lúc nào cũng thực hiện được. Ví dụ, mã sau *không* hợp lệ:

character	codeword
A	10
B	11
C	1011
D	111

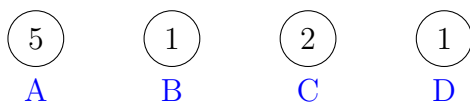
Dùng mã này, ta không thể biết chuỗi nén 1011 tương ứng với chuỗi AB hay chuỗi C.

Mã hóa Huffman (Huffman coding)

Huffman coding² là một thuật toán tham lam xây dựng mã tối ưu để nén một chuỗi cho trước. Thuật toán xây dựng một cây nhị phân dựa trên tần suất xuất hiện của các ký tự trong chuỗi, và codeword của mỗi ký tự có thể được đọc bằng cách đi theo đường từ gốc đến nút tương ứng. Đi sang trái tương ứng với bit 0, và đi sang phải tương ứng với bit 1.

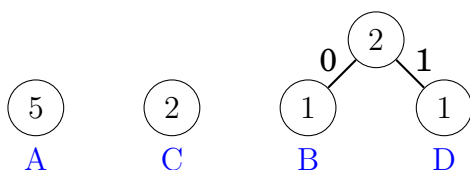
Ban đầu, mỗi ký tự của chuỗi được biểu diễn bởi một nút có trọng số là số lần ký tự đó xuất hiện trong chuỗi. Sau đó, ở mỗi bước, hai nút có trọng số nhỏ nhất được ghép lại bằng cách tạo một nút mới có trọng số bằng tổng trọng số của các nút ban đầu. Quá trình tiếp tục cho đến khi mọi nút đã được ghép.

Tiếp theo ta sẽ xem mã hóa Huffman tạo mã tối ưu cho chuỗi AABACDACA. Ban đầu, có bốn nút tương ứng với các ký tự của chuỗi:



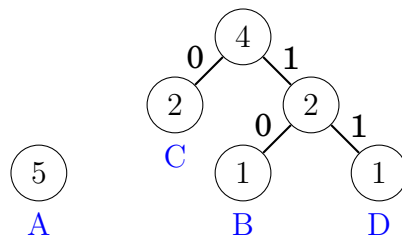
Nút biểu diễn ký tự A có trọng số 5 vì ký tự A xuất hiện 5 lần trong chuỗi. Các trọng số khác được tính tương tự.

Bước đầu tiên là ghép các nút tương ứng với ký tự B và D, cả hai đều có trọng số 1. Kết quả là:

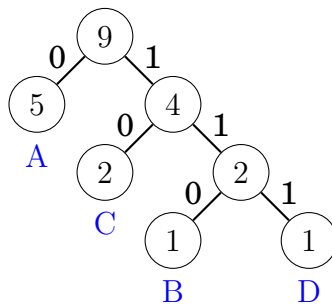


²D. A. Huffman discovered this method đã phát hiện phương pháp này khi giải một bài tập ở đại học và công bố thuật toán vào năm 1952 [40].

Sau đó, các nút có trọng số 2 được ghép lại:



Cuối cùng, hai nút còn lại được ghép lại:



Bây giờ mọi nút đều đã nằm trong cây, nên mã đã sẵn sàng. Các codeword sau có thể được đọc từ cây:

character	codeword
A	0
B	110
C	10
D	111

Chương 7

Quy hoạch động (Dynamic Programming)

Dynamic programming là một kỹ thuật kết hợp tính đúng đắn của tìm kiếm toàn bộ và hiệu quả của thuật toán tham lam. Quy hoạch động có thể được áp dụng nếu bài toán có thể chia thành các bài toán con chồng lấn có thể được giải độc lập.

Có hai cách dùng quy hoạch động:

- **Tìm lời giải tối ưu** (Finding an optimal solution): Ta muốn tìm một lời giải lớn nhất có thể hoặc nhỏ nhất có thể.
- **Đếm số lời giải** (Counting the number of solutions): Ta muốn tính tổng số lời giải có thể.

Trước tiên ta sẽ xem quy hoạch động có thể được dùng để tìm lời giải tối ưu như thế nào, rồi dùng cùng ý tưởng đó để đếm số lời giải.

Hiểu quy hoạch động là một cột mốc trong sự nghiệp của mọi lập trình viên thi đấu. Dù ý tưởng cơ bản khá đơn giản, thử thách nằm ở cách áp dụng quy hoạch động vào các bài toán khác nhau. Chương này giới thiệu một nhóm bài toán kinh điển là điểm khởi đầu tốt.

7.1 Bài toán đồng xu

Trước tiên ta tập trung vào một bài toán đã gặp ở Chương 6: Cho tập giá trị đồng xu $\text{coins} = \{c_1, c_2, \dots, c_k\}$ và tổng tiền mục tiêu n , nhiệm vụ là tạo tổng n bằng ít đồng xu nhất có thể.

Trong Chương 6, ta đã giải bài toán bằng thuật toán tham lam luôn chọn đồng xu lớn nhất có thể. Thuật toán tham lam đúng, chẳng hạn, khi các đồng xu là đồng euro, nhưng trong trường hợp tổng quát thuật toán tham lam không nhất thiết tạo ra lời giải tối ưu.

Bây giờ là lúc giải bài toán một cách hiệu quả bằng quy hoạch động, để thuật toán đúng với mọi tập đồng xu. Thuật toán quy hoạch động dựa trên một hàm đệ quy duyệt qua mọi khả năng để tạo tổng, giống như thuật toán

vét cạn. Tuy nhiên, thuật toán quy hoạch động hiệu quả vì nó dùng *ghi nhớ* (memoization) và chỉ tính đáp án cho mỗi bài toán con một lần.

Công thức đệ quy

Ý tưởng trong quy hoạch động là phát biểu bài toán theo cách đệ quy để lời giải của bài toán có thể được tính từ lời giải của các bài toán con nhỏ hơn. Trong bài toán đồng xu, một bài toán đệ quy tự nhiên là: cần ít nhất bao nhiêu đồng xu để tạo tổng x ?

Gọi $\text{solve}(x)$ là số đồng xu tối thiểu cần để tạo tổng x . Các giá trị của hàm phụ thuộc vào giá trị của các đồng xu. Ví dụ, nếu $\text{coins} = \{1, 3, 4\}$, các giá trị đầu tiên của hàm như sau:

$\text{solve}(0)$	$=$	0
$\text{solve}(1)$	$=$	1
$\text{solve}(2)$	$=$	2
$\text{solve}(3)$	$=$	1
$\text{solve}(4)$	$=$	1
$\text{solve}(5)$	$=$	2
$\text{solve}(6)$	$=$	2
$\text{solve}(7)$	$=$	2
$\text{solve}(8)$	$=$	2
$\text{solve}(9)$	$=$	3
$\text{solve}(10)$	$=$	3

Ví dụ, $\text{solve}(10) = 3$, vì cần ít nhất 3 đồng xu để tạo tổng 10. Lời giải tối ưu là $3 + 3 + 4 = 10$.

Tính chất cốt lõi của solve là các giá trị của nó có thể được tính đệ quy từ các giá trị nhỏ hơn. Ý tưởng là tập trung vào đồng xu *đầu tiên* ta chọn cho tổng. Ví dụ, trong tình huống trên, đồng xu đầu tiên có thể là 1, 3 hoặc 4. Nếu trước tiên chọn đồng 1, nhiệm vụ còn lại là tạo tổng 9 bằng số đồng xu ít nhất, đây là một bài toán con của bài toán ban đầu. Dĩ nhiên, điều tương tự cũng đúng với các đồng 3 và 4. Do đó, ta có thể dùng công thức đệ quy sau để tính số đồng xu tối thiểu:

$$\begin{aligned}\text{solve}(x) = \min(&\text{solve}(x - 1) + 1, \\ &\text{solve}(x - 3) + 1, \\ &\text{solve}(x - 4) + 1).\end{aligned}$$

Trường hợp cơ sở của đệ quy là $\text{solve}(0) = 0$, vì không cần đồng xu nào để tạo tổng rỗng. Ví dụ,

$$\text{solve}(10) = \text{solve}(7) + 1 = \text{solve}(4) + 2 = \text{solve}(0) + 3 = 3.$$

Bây giờ ta đã sẵn sàng đưa ra hàm đệ quy tổng quát tính số đồng xu tối

thiếu cần để tạo tổng x :

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x - c) + 1 & x > 0 \end{cases}$$

Trước hết, nếu $x < 0$, giá trị là ∞ , vì không thể tạo một tổng tiền âm. Sau đó, nếu $x = 0$, giá trị là 0, vì không cần đồng xu nào để tạo tổng rỗng. Cuối cùng, nếu $x > 0$, biến c duyệt qua mọi khả năng chọn đồng xu đầu tiên của tổng.

Khi đã tìm được hàm đệ quy giải bài toán, ta có thể cài đặt trực tiếp lời giải trong C++ (hằng INF biểu diễn vô cực):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Tuy nhiên, hàm này vẫn chưa hiệu quả, vì có thể có số cách tạo tổng theo cấp số nhân. Tiếp theo ta sẽ xem cách làm cho hàm hiệu quả bằng kỹ thuật gọi là memoization.

Dùng ghi nhớ

Ý tưởng của quy hoạch động là dùng **memoization** để tính hiệu quả các giá trị của một hàm đệ quy. Điều này có nghĩa là các giá trị của hàm được lưu trong một mảng sau khi tính xong. Với mỗi tham số, giá trị của hàm chỉ được tính đệ quy một lần, và sau đó có thể được lấy trực tiếp từ mảng.

Trong bài toán này, ta dùng các mảng

```
bool ready[N];
int value[N];
```

trong đó `ready[x]` cho biết giá trị của `solve(x)` đã được tính hay chưa, và nếu đã tính, `value[x]` chứa giá trị đó. Hằng N được chọn sao cho tất cả các giá trị cần thiết đều nằm vừa trong mảng.

Bây giờ hàm có thể được cài đặt hiệu quả như sau:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
```

```

    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}

```

Hàm xử lý các trường hợp cơ sở $x < 0$ và $x = 0$ như trước. Sau đó hàm kiểm tra `ready[x]` để biết `solve(x)` đã được lưu trong `value[x]` hay chưa, và nếu đã có, hàm trả về trực tiếp giá trị đó. Nếu không, hàm tính giá trị của `solve(x)` bằng đệ quy và lưu nó vào `value[x]`.

Hàm này hoạt động hiệu quả, vì đáp án cho mỗi tham số x chỉ được tính đệ quy một lần. Sau khi một giá trị của `solve(x)` đã được lưu trong `value[x]`, nó có thể được lấy ra hiệu quả bất cứ khi nào hàm được gọi lại với tham số x . Độ phức tạp thời gian của thuật toán là $O(nk)$, trong đó n là tổng mục tiêu và k là số lượng đồng xu.

Lưu ý rằng ta cũng có thể *lặp* xây dựng mảng `value` bằng một vòng lặp đơn giản tính tất cả các giá trị của `solve` với tham số $0 \dots n$:

```

value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}

```

Thực tế, hầu hết lập trình viên thi đấu thích cách cài đặt này hơn, vì nó ngắn hơn và có hằng số nhỏ hơn. Từ giờ trở đi, trong các ví dụ ta cũng dùng các cài đặt dạng lặp. Dù vậy, thường sẽ dễ suy nghĩ về lời giải quy hoạch động theo các hàm đệ quy hơn.

Xây dựng một lời giải

Đôi khi ta được yêu cầu vừa tìm giá trị của lời giải tối ưu vừa đưa ra một ví dụ về cách xây dựng lời giải đó. Trong bài toán đồng xu, chẳng hạn, ta có thể khai báo thêm một mảng cho biết với mỗi tổng tiền đồng xu đầu tiên trong một lời giải tối ưu:

```
int first[N];
```

Sau đó, ta có thể sửa thuật toán như sau:

```

value[0] = 0;
for (int x = 1; x <= n; x++) {

```

```

    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first [x] = c;
        }
    }
}

```

Sau đó, đoạn mã sau có thể được dùng để in các đồng xu xuất hiện trong một lời giải tối ưu cho tổng n :

```

while (n > 0) {
    cout << first[n] << "\n";
    n -= first [n];
}

```

Đếm số lời giải

Bây giờ hãy xét một phiên bản khác của bài toán đồng xu, trong đó nhiệm vụ là tính tổng số cách tạo ra tổng x bằng các đồng xu. Ví dụ, nếu $\text{coins} = \{1, 3, 4\}$ và $x = 5$, có tổng cộng 6 cách:

- $1 + 1 + 1 + 1 + 1$
- $3 + 1 + 1$
- $1 + 1 + 3$
- $1 + 4$
- $1 + 3 + 1$
- $4 + 1$

Một lần nữa, ta có thể giải bài toán bằng đệ quy. Gọi $\text{solve}(x)$ là số cách ta có thể tạo tổng x . Ví dụ, nếu $\text{coins} = \{1, 3, 4\}$, thì $\text{solve}(5) = 6$ và công thức đệ quy là

$$\text{solve}(x) = \text{solve}(x-1) + \text{solve}(x-3) + \text{solve}(x-4).$$

Khi đó, hàm đệ quy tổng quát như sau:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

Nếu $x < 0$, giá trị là 0, vì không có lời giải nào. Nếu $x = 0$, giá trị là 1, vì chỉ có một cách tạo tổng rỗng. Ngược lại, ta tính tổng của mọi giá trị dạng $\text{solve}(x-c)$ trong đó c thuộc coins .

Đoạn mã sau xây dựng mảng `count` sao cho `count[x]` bằng giá trị của $\text{solve}(x)$ với $0 \leq x \leq n$:

```

count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x-c >= 0) {
            count[x] += count[x-c];
        }
    }
}

```

Thường thì số lời giải rất lớn đến mức không cần tính con số chính xác, mà chỉ cần đưa ra đáp án modulo m , ví dụ $m = 10^9 + 7$. Điều này có thể được thực hiện bằng cách sửa mã sao cho mọi phép tính đều được thực hiện modulo m . Trong đoạn mã trên, chỉ cần thêm dòng

```
count[x] %= m;
```

sau dòng

```
count[x] += count[x-c];
```

Bây giờ ta đã thảo luận mọi ý tưởng cơ bản của quy hoạch động. Vì quy hoạch động có thể được dùng trong nhiều tình huống khác nhau, bây giờ ta sẽ đi qua một tập các bài toán minh họa thêm những khả năng của quy hoạch động.

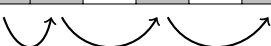
7.2 Dãy con tăng dài nhất (Longest increasing subsequence)

Bài toán đầu tiên là tìm **longest increasing subsequence** trong một mảng có n phần tử. Đây là một dãy các phần tử của mảng có độ dài lớn nhất, đi từ trái sang phải, và mỗi phần tử trong dãy lớn hơn phần tử đứng trước nó. Ví dụ, trong mảng

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

dãy con tăng dài nhất chứa 4 phần tử:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3



Gọi $\text{length}(k)$ là độ dài của dãy con tăng dài nhất kết thúc tại vị trí k . Do đó, nếu tính tất cả các giá trị $\text{length}(k)$ với $0 \leq k \leq n - 1$, ta sẽ tìm được độ

dài của dãy con tăng dài nhất. Ví dụ, các giá trị của hàm với mảng trên như sau:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

Ví dụ, $\text{length}(6) = 4$, vì dãy con tăng dài nhất kết thúc tại vị trí 6 gồm 4 phần tử.

Để tính giá trị $\text{length}(k)$, ta cần tìm một vị trí $i < k$ sao cho $\text{array}[i] < \text{array}[k]$ và $\text{length}(i)$ lớn nhất có thể. Khi đó ta biết rằng $\text{length}(k) = \text{length}(i) + 1$, vì đây là cách tối ưu để thêm $\text{array}[k]$ vào một dãy con. Tuy nhiên, nếu không có vị trí i như vậy, thì $\text{length}(k) = 1$, nghĩa là dãy con chỉ chứa $\text{array}[k]$.

Vì mọi giá trị của hàm có thể được tính từ các giá trị nhỏ hơn, ta có thể dùng quy hoạch động. Trong đoạn mã sau, các giá trị của hàm sẽ được lưu trong mảng `length`.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k], length[i] + 1);
        }
    }
}
```

Đoạn mã này chạy trong thời gian $O(n^2)$, vì nó gồm hai vòng lặp lồng nhau. Tuy nhiên, cũng có thể cài đặt phép tính quy hoạch động hiệu quả hơn trong thời gian $O(n \log n)$. Bạn có tìm được cách làm không?

7.3 Đường đi trong lưới

Bài toán tiếp theo là tìm một đường đi từ góc trên bên trái đến góc dưới bên phải của lưới $n \times n$, sao cho ta chỉ đi xuống hoặc sang phải. Mỗi ô chứa một số nguyên dương, và đường đi cần được xây dựng sao cho tổng các giá trị trên đường đi lớn nhất có thể.

Hình sau cho thấy một đường đi tối ưu trong lưới:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Tổng các giá trị trên đường đi là 67, và đây là tổng lớn nhất có thể trên một đường đi từ góc trên bên trái đến góc dưới bên phải.

Giả sử các hàng và cột của lưới được đánh số từ 1 đến n , và $\text{value}[y][x]$ bằng giá trị của ô (y, x) . Gọi $\text{sum}(y, x)$ là tổng lớn nhất trên một đường đi từ góc trên bên trái đến ô (y, x) . Khi đó $\text{sum}(n, n)$ cho ta tổng lớn nhất từ góc trên bên trái đến góc dưới bên phải. Ví dụ, trong lưới trên, $\text{sum}(5, 5) = 67$.

Ta có thể tính các tổng bằng đệ quy như sau:

$$\text{sum}(y, x) = \max(\text{sum}(y, x - 1), \text{sum}(y - 1, x)) + \text{value}[y][x]$$

Công thức đệ quy dựa trên nhận xét rằng một đường đi kết thúc tại ô (y, x) có thể đi đến từ ô $(y, x - 1)$ hoặc ô $(y - 1, x)$:

			↓	
		→		

Do đó, ta chọn hướng làm tổng lớn nhất. Ta giả sử $\text{sum}(y, x) = 0$ nếu $y = 0$ hoặc $x = 0$ (vì không tồn tại đường đi như vậy), nên công thức đệ quy cũng đúng khi $y = 1$ hoặc $x = 1$.

Vì hàm sum có hai tham số, mảng quy hoạch động cũng có hai chiều. Ví dụ, ta có thể dùng mảng

```
int sum[N][N];
```

và tính các tổng như sau:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];
    }
}
```

Độ phức tạp thời gian của thuật toán là $O(n^2)$.

7.4 Các bài toán ba lô (Knapsack)

Thuật ngữ **knapsack** chỉ các bài toán trong đó ta được cho một tập đồ vật, và cần tìm các tập con có một số tính chất. Các bài toán ba lô thường có thể

được giải bằng quy hoạch động.

Trong phần này, ta tập trung vào bài toán sau: Cho một danh sách trọng lượng $[w_1, w_2, \dots, w_n]$, hãy xác định tất cả các tổng có thể được tạo bằng các trọng lượng đó. Ví dụ, nếu các trọng lượng là $[1, 3, 3, 5]$, các tổng sau là có thể:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

Trong trường hợp này, mọi tổng từ $0 \dots 12$ đều có thể tạo được, trừ 2 và 10. Ví dụ, tổng 7 là có thể vì ta có thể chọn các trọng lượng $[1, 3, 3]$.

Để giải bài toán, ta tập trung vào các bài toán con trong đó chỉ dùng k trọng lượng đầu tiên để tạo tổng. Đặt $\text{possible}(x, k) = \text{true}$ nếu ta có thể tạo tổng x bằng k trọng lượng đầu tiên, và ngược lại $\text{possible}(x, k) = \text{false}$. Các giá trị của hàm có thể được tính đệ quy như sau:

$$\text{possible}(x, k) = \text{possible}(x - w_k, k - 1) \vee \text{possible}(x, k - 1)$$

Công thức dựa trên việc ta có thể dùng hoặc không dùng trọng lượng w_k trong tổng. Nếu dùng w_k , nhiệm vụ còn lại là tạo tổng $x - w_k$ bằng $k - 1$ trọng lượng đầu tiên, và nếu không dùng w_k , nhiệm vụ còn lại là tạo tổng x bằng $k - 1$ trọng lượng đầu tiên. Các trường hợp cơ sở là

$$\text{possible}(x, 0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

vì nếu không dùng trọng lượng nào, ta chỉ có thể tạo tổng 0.

Bảng sau cho thấy mọi giá trị của hàm với các trọng lượng $[1, 3, 3, 5]$ (ký hiệu “X” biểu thị các giá trị đúng):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

Sau khi tính các giá trị đó, $\text{possible}(x, n)$ cho ta biết liệu có thể tạo tổng x bằng *tất cả* trọng lượng hay không.

Gọi W là tổng của các trọng lượng. Lời giải quy hoạch động thời gian $O(nW)$ sau đây tương ứng với hàm đệ quy:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x - w[k] >= 0) possible[x][k] |= possible[x - w[k]][k - 1];
        possible[x][k] |= possible[x][k - 1];
    }
}
```

Tuy nhiên, đây là một cài đặt tốt hơn chỉ dùng mảng một chiều `possible[x]` cho biết liệu ta có thể tạo một tập con có tổng x hay không. Mẹo là cập nhật mảng từ phải sang trái với mỗi trọng lượng mới:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Lưu ý rằng ý tưởng tổng quát được trình bày ở đây có thể dùng trong nhiều bài toán ba lô. Ví dụ, nếu ta được cho các đồ vật có trọng lượng và giá trị, ta có thể xác định với mỗi tổng trọng lượng, tổng giá trị lớn nhất của một tập con.

7.5 Khoảng cách chỉnh sửa (Edit distance)

edit distance hay **Levenshtein distance**¹ là số thao tác chỉnh sửa ít nhất cần để biến một chuỗi thành một chuỗi khác. Các thao tác chỉnh sửa được phép là:

- chèn một ký tự (ví dụ $ABC \rightarrow ABCA$)
- xóa một ký tự (ví dụ $ABC \rightarrow AC$)
- sửa một ký tự (ví dụ $ABC \rightarrow ADC$)

Ví dụ, khoảng cách chỉnh sửa giữa `LOVE` và `MOVIE` là 2, vì trước tiên ta có thể thực hiện thao tác $LOVE \rightarrow MOVE$ (sửa) rồi thao tác $MOVE \rightarrow MOVIE$ (chèn). Đây là số thao tác nhỏ nhất có thể, vì rõ ràng chỉ một thao tác là không đủ.

Giả sử ta được cho chuỗi x có độ dài n và chuỗi y có độ dài m , và muốn tính khoảng cách chỉnh sửa giữa x và y . Để giải bài toán, ta định nghĩa hàm $distance(a, b)$ cho biết khoảng cách chỉnh sửa giữa các tiền tố $x[0 \dots a]$ và $y[0 \dots b]$. Do đó, dùng hàm này, khoảng cách chỉnh sửa giữa x và y bằng $distance(n-1, m-1)$.

Ta có thể tính các giá trị của $distance$ như sau:

$$distance(a, b) = \min(\begin{aligned} &distance(a, b-1) + 1, \\ &distance(a-1, b) + 1, \\ &distance(a-1, b-1) + cost(a, b) \end{aligned}).$$

Ở đây $cost(a, b) = 0$ nếu $x[a] = y[b]$, và ngược lại $cost(a, b) = 1$. Công thức xét các cách sau để chỉnh sửa chuỗi x :

¹Khoảng cách này được đặt theo tên V. I. Levenshtein, người nghiên cứu nó trong liên hệ với mã nhị phân [49].

- $\text{distance}(a, b - 1)$: chèn một ký tự vào cuối x
- $\text{distance}(a - 1, b)$: xóa ký tự cuối của x
- $\text{distance}(a - 1, b - 1)$: khớp hoặc sửa ký tự cuối của x

Trong hai trường hợp đầu, cần một thao tác chỉnh sửa (chèn hoặc xóa). Trong trường hợp cuối, nếu $x[a] = y[b]$, ta có thể khớp các ký tự cuối mà không cần chỉnh sửa, và ngược lại cần một thao tác chỉnh sửa (sửa).

Bảng sau cho thấy các giá trị của distance trong ví dụ:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

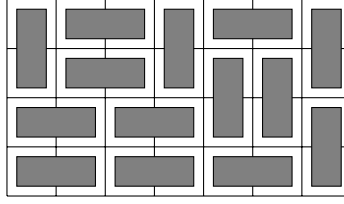
Góc dưới bên phải của bảng cho ta biết rằng khoảng cách chỉnh sửa giữa LOVE và MOVIE là 2. Bảng cũng cho thấy cách xây dựng dãy thao tác chỉnh sửa ngắn nhất. Trong trường hợp này, đường đi như sau:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

Các ký tự cuối của LOVE và MOVIE bằng nhau, nên khoảng cách chỉnh sửa giữa chúng bằng khoảng cách chỉnh sửa giữa LOV và MOVI. Ta có thể dùng một thao tác chỉnh sửa để xóa ký tự I khỏi MOVI. Do đó, khoảng cách chỉnh sửa lớn hơn một đơn vị so với khoảng cách chỉnh sửa giữa LOV và MOV, v.v.

7.6 Đếm cách lát gạch

Đôi khi các trạng thái của một lời giải quy hoạch động phức tạp hơn những tổ hợp cố định của các số. Làm ví dụ, xét bài toán tính số cách khác nhau để lát một lưới $n \times m$ bằng các viên gạch kích thước 1×2 và 2×1 . Ví dụ, một lời giải hợp lệ cho lưới 4×7 là



và tổng số lời giải là 781.

Bài toán có thể được giải bằng quy hoạch động bằng cách duyệt lưới theo từng hàng. Mỗi hàng trong một lời giải có thể được biểu diễn bằng một chuỗi chứa m ký tự từ tập $\{\sqcup, \sqcup, \sqcup, \sqcup\}$. Ví dụ, lời giải ở trên gồm bốn hàng tương ứng với các chuỗi sau:

- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$

Gọi $\text{count}(k, x)$ là số cách xây dựng lời giải cho các hàng $1 \dots k$ của lưới sao cho chuỗi x tương ứng với hàng k . Ở đây có thể dùng quy hoạch động, vì trạng thái của một hàng chỉ bị ràng buộc bởi trạng thái của hàng trước đó.

Một lời giải là hợp lệ nếu hàng 1 không chứa ký tự $\sqcup$, hàng n không chứa ký tự $\sqcup$, và mọi cặp hàng liên tiếp đều *tương thích*. Ví dụ, các hàng $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ và $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ là tương thích, trong khi các hàng $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ và $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ là không tương thích.

Vì một hàng gồm m ký tự và mỗi ký tự có bốn lựa chọn, số hàng khác nhau nhiều nhất là 4^m . Do đó, độ phức tạp thời gian của lời giải là $O(n4^{2m})$ vì ta có thể duyệt qua $O(4^m)$ trạng thái có thể cho mỗi hàng, và với mỗi trạng thái, có $O(4^m)$ trạng thái có thể của hàng trước đó. Trong thực tế, nên xoay lưới sao cho cạnh ngắn hơn có độ dài m , vì thừa số 4^{2m} chi phối độ phức tạp thời gian.

Có thể làm lời giải hiệu quả hơn bằng cách dùng biểu diễn gọn hơn cho các hàng. Hóa ra chỉ cần biết những cột nào của hàng trước chứa ô phía trên của một viên gạch dọc. Vì vậy, ta có thể biểu diễn một hàng chỉ bằng các ký tự $\sqcup$ và $\sqcup$, trong đó $\sqcup$ là một tổ hợp của các ký tự $\sqcup$, $\sqcup$ và $\sqcup$. Dùng biểu diễn này, chỉ có 2^m hàng khác nhau và độ phức tạp thời gian là $O(n2^{2m})$.

Cuối cùng, cũng có một công thức trực tiếp khá bất ngờ để tính số cách lát gạch²:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right)$$

Công thức này rất hiệu quả, vì nó tính số cách lát gạch trong thời gian $O(nm)$, nhưng vì đáp án là một tích của các số thực, một vấn đề khi dùng công thức là cách lưu các kết quả trung gian một cách chính xác.

²Điều bất ngờ là công thức này được phát hiện vào năm 1961 bởi hai nhóm nghiên cứu [43, 67] làm việc độc lập.

Chương 8

Phân tích khấu hao

Độ phức tạp thời gian của một thuật toán thường dễ phân tích chỉ bằng cách xem xét cấu trúc của thuật toán: thuật toán có những vòng lặp nào và các vòng lặp được thực hiện bao nhiêu lần. Tuy nhiên, đôi khi một phân tích trực tiếp không cho thấy bức tranh đúng về hiệu quả của thuật toán.

Phân tích khấu hao (amortized analysis) có thể được dùng để phân tích các thuật toán chứa những thao tác có độ phức tạp thời gian thay đổi. Ý tưởng là ước lượng tổng thời gian dùng cho tất cả các thao tác như vậy trong suốt quá trình thực thi thuật toán, thay vì tập trung vào từng thao tác riêng lẻ.

8.1 Phương pháp hai con trỏ

Trong **phương pháp hai con trỏ (two pointers method)**, hai con trỏ được dùng để duyệt qua các giá trị của mảng. Cả hai con trỏ chỉ có thể di chuyển theo một hướng, điều này đảm bảo thuật toán hoạt động hiệu quả. Tiếp theo ta thảo luận hai bài toán có thể giải bằng phương pháp hai con trỏ.

Tổng đoạn con

Lấy ví dụ đầu tiên, xét một bài toán trong đó ta được cho một mảng gồm n số nguyên dương và một tổng mục tiêu x , và ta muốn tìm một đoạn con có tổng bằng x hoặc thông báo rằng không có đoạn con như vậy.

Ví dụ, mảng

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

có một đoạn con có tổng bằng 8:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Bài toán này có thể được giải trong thời gian $O(n)$ bằng phương pháp hai con trỏ. Ý tưởng là duy trì các con trỏ tới giá trị đầu tiên và cuối cùng

của một đoạn con. Ở mỗi lượt, con trỏ trái di chuyển một bước sang phải, và con trỏ phải di chuyển sang phải chừng nào tổng của đoạn con thu được vẫn không vượt quá x . Nếu tổng trở thành đúng bằng x , ta đã tìm được một lời giải.

Ví dụ, xét mảng sau và tổng mục tiêu $x = 8$:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Đoạn con ban đầu chứa các giá trị 1, 3 và 2, có tổng bằng 6:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Sau đó, con trỏ trái di chuyển một bước sang phải. Con trỏ phải không di chuyển, vì nếu không tổng đoạn con sẽ vượt quá x .

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Một lần nữa, con trỏ trái di chuyển một bước sang phải, và lần này con trỏ phải di chuyển ba bước sang phải. Tổng đoạn con là $2 + 5 + 1 = 8$, nên một đoạn con có tổng bằng x đã được tìm thấy.

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Thời gian chạy của thuật toán phụ thuộc vào số bước mà con trỏ phải di chuyển. Dù không có chặn trên hữu ích cho số bước mà con trỏ có thể di chuyển trong *một* lượt, ta biết rằng con trỏ di chuyển *tổng cộng* $O(n)$ bước trong suốt thuật toán, vì nó chỉ di chuyển sang phải.

Vì cả con trỏ trái và con trỏ phải đều di chuyển $O(n)$ bước trong thuật toán, thuật toán chạy trong thời gian $O(n)$.

Bài toán 2SUM

Một bài toán khác có thể giải bằng phương pháp hai con trỏ là bài toán sau, còn được gọi là **bài toán 2SUM**: cho một mảng gồm n số và một tổng mục tiêu x , hãy tìm hai giá trị trong mảng sao cho tổng của chúng bằng x , hoặc thông báo rằng không tồn tại các giá trị như vậy.

Để giải bài toán, trước hết ta sắp xếp các giá trị của mảng theo thứ tự tăng dần. Sau đó, ta duyệt mảng bằng hai con trỏ. Con trỏ trái bắt đầu ở giá trị đầu tiên và di chuyển một bước sang phải ở mỗi lượt. Con trỏ phải bắt đầu ở giá trị cuối cùng và luôn di chuyển sang trái cho đến khi tổng của giá trị bên trái và bên phải không vượt quá x . Nếu tổng đúng bằng x , ta đã tìm được một lời giải.

Ví dụ, xét mảng sau và tổng mục tiêu $x = 12$:

1	4	5	6	7	9	9	10
---	---	---	---	---	---	---	----

Vị trí ban đầu của các con trỏ như sau. Tổng của các giá trị là $1 + 10 = 11$, nhỏ hơn x .

1	4	5	6	7	9	9	10
↑							↑

Sau đó con trỏ trái di chuyển một bước sang phải. Con trỏ phải di chuyển ba bước sang trái, và tổng trở thành $4 + 7 = 11$.

1	4	5	6	7	9	9	10
	↑			↑			

Sau đó, con trỏ trái lại di chuyển một bước sang phải. Con trỏ phải không di chuyển, và lời giải $5 + 7 = 12$ đã được tìm thấy.

1	4	5	6	7	9	9	10
		↑		↑			

Thời gian chạy của thuật toán là $O(n \log n)$, vì trước hết nó sắp xếp mảng trong thời gian $O(n \log n)$, và sau đó cả hai con trỏ di chuyển $O(n)$ bước.

Lưu ý rằng có thể giải bài toán theo một cách khác trong thời gian $O(n \log n)$ bằng tìm kiếm nhị phân (binary search). Trong lời giải như vậy, ta duyệt qua mảng và với mỗi giá trị của mảng, ta cố tìm một giá trị khác tạo ra tổng x . Điều này có thể thực hiện bằng cách làm n phép tìm kiếm nhị phân, mỗi phép mất thời gian $O(\log n)$.

Một bài toán khó hơn là **bài toán 3SUM**, yêu cầu tìm *ba* giá trị trong mảng có tổng bằng x . Dường ý tưởng của thuật toán trên, bài toán này có thể được giải trong thời gian $O(n^2)$ ¹. Bạn có thấy cách làm không?

8.2 Phần tử nhỏ hơn gần nhất

Phân tích khấu hao thường được dùng để ước lượng số thao tác được thực hiện trên một cấu trúc dữ liệu. Các thao tác có thể được phân bố không đều, sao cho phần lớn thao tác xảy ra trong một giai đoạn nhất định của thuật toán, nhưng tổng số thao tác vẫn bị giới hạn.

Ví dụ, xét bài toán tìm cho mỗi phần tử của mảng **phần tử nhỏ hơn gần nhất (nearest smaller element)**, tức là phần tử nhỏ hơn đầu tiên đứng trước phần tử đó trong mảng. Có thể không tồn tại phần tử như vậy, trong trường hợp đó thuật toán cần thông báo điều này. Tiếp theo ta sẽ thấy cách giải bài toán hiệu quả bằng cấu trúc ngăn xếp.

¹Trong một thời gian dài, người ta cho rằng không thể giải bài toán 3SUM hiệu quả hơn thời gian $O(n^2)$. Tuy nhiên, vào năm 2014, kết quả [30] cho thấy điều này không đúng.

Ta đi qua mảng từ trái sang phải và duy trì một ngăn xếp các phần tử của mảng. Tại mỗi vị trí trong mảng, ta loại bỏ các phần tử khỏi ngăn xếp cho đến khi phần tử trên đỉnh nhỏ hơn phần tử hiện tại, hoặc ngăn xếp rỗng. Sau đó, ta thông báo rằng phần tử trên đỉnh là phần tử nhỏ hơn gần nhất của phần tử hiện tại, hoặc nếu ngăn xếp rỗng, không có phần tử như vậy. Cuối cùng, ta thêm phần tử hiện tại vào ngăn xếp.

Ví dụ, xét mảng sau:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

Đầu tiên, các phần tử 1, 3 và 4 được thêm vào ngăn xếp, vì mỗi phần tử đều lớn hơn phần tử trước đó. Do đó, phần tử nhỏ hơn gần nhất của 4 là 3, và phần tử nhỏ hơn gần nhất của 3 là 1.

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 3 → 4

Phần tử tiếp theo 2 nhỏ hơn hai phần tử trên cùng trong ngăn xếp. Do đó, các phần tử 3 và 4 bị loại khỏi ngăn xếp, rồi phần tử 2 được thêm vào ngăn xếp. Phần tử nhỏ hơn gần nhất của nó là 1:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2

Sau đó, phần tử 5 lớn hơn phần tử 2, nên nó sẽ được thêm vào ngăn xếp, và phần tử nhỏ hơn gần nhất của nó là 2:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2 → 5

Sau đó, phần tử 3 bị loại khỏi ngăn xếp và các phần tử 3 và 4 được thêm vào ngăn xếp:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2 → 3 → 4

Cuối cùng, tất cả các phần tử trừ 1 bị loại khỏi ngăn xếp và phần tử cuối cùng 2 được thêm vào ngăn xếp:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2

Hiệu quả của thuật toán phụ thuộc vào tổng số thao tác trên ngăn xếp. Nếu phần tử hiện tại lớn hơn phần tử trên đỉnh ngăn xếp, nó được thêm trực tiếp vào ngăn xếp, việc này hiệu quả. Tuy nhiên, đôi khi ngăn xếp có thể chứa nhiều phần tử lớn hơn và cần thời gian để loại bỏ chúng. Dù vậy, mỗi phần tử được thêm *đúng một lần* vào ngăn xếp và bị loại khỏi ngăn xếp *nhều nhất một lần*. Do đó, mỗi phần tử gây ra $O(1)$ thao tác trên ngăn xếp, và thuật toán chạy trong thời gian $O(n)$.

8.3 Giá trị nhỏ nhất của cửa sổ trượt

Cửa sổ trượt (sliding window) là một đoạn con có kích thước cố định di chuyển từ trái sang phải qua mảng. Tại mỗi vị trí của cửa sổ, ta muốn tính một số thông tin về các phần tử nằm trong cửa sổ. Trong mục này, ta tập trung vào bài toán duy trì **giá trị nhỏ nhất của cửa sổ trượt (sliding window minimum)**, nghĩa là ta cần báo cáo giá trị nhỏ nhất bên trong mỗi cửa sổ.

Giá trị nhỏ nhất của cửa sổ trượt có thể được tính bằng một ý tưởng tương tự ý tưởng ta đã dùng để tính các phần tử nhỏ hơn gần nhất. Ta duy trì một hàng đợi trong đó mỗi phần tử lớn hơn phần tử trước nó, và phần tử đầu tiên luôn tương ứng với phần tử nhỏ nhất bên trong cửa sổ. Sau mỗi lần cửa sổ di chuyển, ta loại bỏ các phần tử ở cuối hàng đợi cho đến khi phần tử cuối của hàng đợi nhỏ hơn phần tử mới của cửa sổ, hoặc hàng đợi trở nên rỗng. Ta cũng loại bỏ phần tử đầu tiên của hàng đợi nếu nó không còn nằm trong cửa sổ. Cuối cùng, ta thêm phần tử mới của cửa sổ vào cuối hàng đợi.

Ví dụ, xét mảng sau:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

Giả sử kích thước của cửa sổ trượt là 4. Ở vị trí đầu tiên của cửa sổ, giá trị nhỏ nhất là 1:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 4 → 5

Sau đó cửa sổ di chuyển một bước sang phải. Phần tử mới 3 nhỏ hơn các phần tử 4 và 5 trong hàng đợi, nên các phần tử 4 và 5 bị loại khỏi hàng đợi và phần tử 3 được thêm vào hàng đợi. Giá trị nhỏ nhất vẫn là 1.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 3

Sau đó, cửa sổ lại di chuyển, và phần tử nhỏ nhất 1 không còn thuộc cửa sổ nữa. Do đó, nó bị loại khỏi hàng đợi và giá trị nhỏ nhất bây giờ là 3. Phần tử mới 4 cũng được thêm vào hàng đợi.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

3

 →

4

Phần tử mới tiếp theo 1 nhỏ hơn tất cả các phần tử trong hàng đợi. Do đó, mọi phần tử bị loại khỏi hàng đợi và hàng đợi sẽ chỉ chứa phần tử 1:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1

Cuối cùng của số đến vị trí cuối cùng. Phần tử 2 được thêm vào hàng đợi, nhưng giá trị nhỏ nhất bên trong của số vẫn là 1.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1

 →

2

Vì mỗi phần tử của mảng được thêm vào hàng đợi đúng một lần và bị loại khỏi hàng đợi nhiều nhất một lần, thuật toán chạy trong thời gian $O(n)$.

Chương 9

Truy vấn đoạn (Range Queries)

Trong chương này, chúng ta thảo luận về các cấu trúc dữ liệu cho phép xử lý hiệu quả các truy vấn đoạn. Trong một **range query** (truy vấn đoạn), nhiệm vụ của chúng ta là tính một giá trị dựa trên một mảng con của một mảng. Các truy vấn đoạn thường gặp là:

- $\text{sum}_q(a, b)$: tính tổng các giá trị trong đoạn $[a, b]$
- $\text{min}_q(a, b)$: tìm giá trị nhỏ nhất trong đoạn $[a, b]$
- $\text{max}_q(a, b)$: tìm giá trị lớn nhất trong đoạn $[a, b]$

Ví dụ, xét đoạn $[3, 6]$ trong mảng sau:

0	1	2	3	4	5	6	7
1	3	8	4	6	1	3	4

Trong trường hợp này, $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ và $\text{max}_q(3, 6) = 6$.

Một cách đơn giản để xử lý truy vấn đoạn là dùng một vòng lặp đi qua tất cả các giá trị của mảng trong đoạn. Ví dụ, hàm sau có thể được dùng để xử lý các truy vấn tổng trên một mảng:

```
int sum(int a, int b) {  
    int s = 0;  
    for (int i = a; i <= b; i++) {  
        s += array[i];  
    }  
    return s;  
}
```

Hàm này chạy trong thời gian $O(n)$, trong đó n là kích thước của mảng. Vì vậy, dùng hàm này ta có thể xử lý q truy vấn trong thời gian $O(nq)$. Tuy nhiên, nếu cả n và q đều lớn, cách tiếp cận này sẽ chậm. May mắn là có những cách xử lý truy vấn đoạn hiệu quả hơn nhiều.

9.1 Truy vấn trên mảng tĩnh

Trước hết, ta tập trung vào tình huống mảng là *tĩnh*, tức là các giá trị của mảng không bao giờ được cập nhật giữa các truy vấn. Trong trường hợp này, chỉ cần xây dựng một cấu trúc dữ liệu tĩnh cho ta câu trả lời cho mọi truy vấn có thể có.

Truy vấn tổng

Ta có thể dễ dàng xử lý các truy vấn tổng trên một mảng tĩnh bằng cách xây dựng một **prefix sum array** (mảng tổng tiền tố). Mỗi giá trị trong mảng tổng tiền tố bằng tổng các giá trị trong mảng ban đầu đến vị trí đó, tức là giá trị tại vị trí k là $\text{sum}_q(0, k)$. Mảng tổng tiền tố có thể được xây dựng trong thời gian $O(n)$.

Ví dụ, xét mảng sau:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Mảng tổng tiền tố tương ứng như sau:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Vì mảng tổng tiền tố chứa tất cả các giá trị của $\text{sum}_q(0, k)$, ta có thể tính bất kỳ giá trị nào của $\text{sum}_q(a, b)$ trong thời gian $O(1)$ như sau:

$$\text{sum}_q(a, b) = \text{sum}_q(0, b) - \text{sum}_q(0, a - 1)$$

Bằng cách định nghĩa $\text{sum}_q(0, -1) = 0$, công thức trên cũng đúng khi $a = 0$.

Ví dụ, xét đoạn $[3, 6]$:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Trong trường hợp này $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Tổng này có thể được tính từ hai giá trị của mảng tổng tiền tố:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Do đó, $\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$.

Cũng có thể tổng quát hóa ý tưởng này lên các chiều cao hơn. Ví dụ, ta có thể xây dựng một mảng tổng tiền tố hai chiều, dùng để tính tổng của bất kỳ mảng con hình chữ nhật nào trong thời gian $O(1)$. Mỗi tổng trong một mảng như vậy tương ứng với một mảng con bắt đầu từ góc trên bên trái của mảng.

Hình sau minh họa ý tưởng này:

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

Tổng của mảng con màu xám có thể được tính bằng công thức

$$S(A) - S(B) - S(C) + S(D),$$

trong đó $S(X)$ ký hiệu tổng các giá trị trong một mảng con hình chữ nhật từ góc trên bên trái đến vị trí của X .

Truy vấn nhỏ nhất

Truy vấn nhỏ nhất khó xử lý hơn truy vấn tổng. Dù vậy, có một phương pháp tiên xử lý khá đơn giản trong thời gian $O(n \log n)$, sau đó ta có thể trả lời mọi truy vấn nhỏ nhất trong thời gian $O(1)$ ¹. Lưu ý rằng vì truy vấn nhỏ nhất và lớn nhất có thể được xử lý tương tự, ta có thể tập trung vào truy vấn nhỏ nhất.

Ý tưởng là tính trước tất cả các giá trị của $\min_q(a, b)$ trong đó $b - a + 1$ (độ dài của đoạn) là một lũy thừa của hai. Ví dụ, với mảng

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

các giá trị sau được tính:

a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$
0	0	1	0	1	1	0	3	1
1	1	3	1	2	3	1	4	3
2	2	4	2	3	4	2	5	1
3	3	8	3	4	6	3	6	1
4	4	6	4	5	1	4	7	1
5	5	1	5	6	1	0	7	1
6	6	4	6	7	2			
7	7	2						

¹Kỹ thuật này được giới thiệu trong [7] và đôi khi được gọi là phương pháp **sparse table**. Cũng có những kỹ thuật phức tạp hơn [22] với thời gian tiên xử lý chỉ là $O(n)$, nhưng các thuật toán như vậy không cần thiết trong lập trình thi đấu.

Số lượng giá trị được tính trước là $O(n \log n)$, vì có $O(\log n)$ độ dài đoạn là lũy thừa của hai. Các giá trị có thể được tính hiệu quả bằng công thức truy hồi

$$\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b)),$$

trong đó $b - a + 1$ là một lũy thừa của hai và $w = (b - a + 1)/2$. Việc tính tất cả các giá trị đó mất thời gian $O(n \log n)$.

Sau đó, bất kỳ giá trị nào của $\min_q(a, b)$ cũng có thể được tính trong thời gian $O(1)$ như giá trị nhỏ nhất của hai giá trị đã tính trước. Gọi k là lũy thừa lớn nhất của hai không vượt quá $b - a + 1$. Ta có thể tính giá trị của $\min_q(a, b)$ bằng công thức

$$\min_q(a, b) = \min(\min_q(a, a + k - 1), \min_q(b - k + 1, b)).$$

Trong công thức trên, đoạn $[a, b]$ được biểu diễn như hợp của hai đoạn $[a, a + k - 1]$ và $[b - k + 1, b]$, cả hai đều có độ dài k .

Ví dụ, xét đoạn $[1, 6]$:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Độ dài của đoạn là 6, và lũy thừa lớn nhất của hai không vượt quá 6 là 4. Do đó đoạn $[1, 6]$ là hợp của hai đoạn $[1, 4]$ và $[3, 6]$:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Vì $\min_q(1, 4) = 3$ và $\min_q(3, 6) = 1$, ta kết luận rằng $\min_q(1, 6) = 1$.

9.2 Cây chỉ số nhị phân

Có thể xem **binary indexed tree** (cây chỉ số nhị phân) hay **Fenwick tree**² như một biến thể động của mảng tổng tiền tố. Nó hỗ trợ hai thao tác trên mảng trong thời gian $O(\log n)$: xử lý một truy vấn tổng trên đoạn và cập nhật một giá trị.

Ưu điểm của cây chỉ số nhị phân là nó cho phép ta cập nhật hiệu quả các giá trị của mảng giữa các truy vấn tổng. Điều này không thể làm được bằng mảng tổng tiền tố, vì sau mỗi lần cập nhật, cần xây dựng lại toàn bộ mảng tổng tiền tố trong thời gian $O(n)$.

²Cấu trúc cây chỉ số nhị phân được P. M. Fenwick trình bày vào năm 1994 [21].

Cấu trúc

Dù tên của cấu trúc là một cây chỉ số nhị phân, nó thường được biểu diễn bằng một mảng. Trong phần này, ta giả sử mọi mảng được đánh chỉ số từ một, vì điều đó làm cho việc cài đặt dễ hơn.

Gọi $p(k)$ là lũy thừa lớn nhất của hai chia hết k . Ta lưu một cây chỉ số nhị phân dưới dạng một mảng $tree$ sao cho

$$tree[k] = \text{sum}_q(k - p(k) + 1, k),$$

tức là mỗi vị trí k chứa tổng các giá trị trong một đoạn của mảng ban đầu có độ dài $p(k)$ và kết thúc tại vị trí k . Ví dụ, vì $p(6) = 2$, $tree[6]$ chứa giá trị của $\text{sum}_q(5, 6)$.

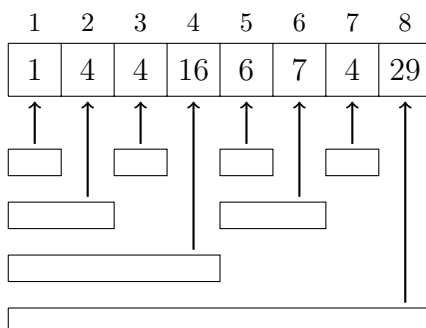
Ví dụ, xét mảng sau:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

Cây chỉ số nhị phân tương ứng như sau:

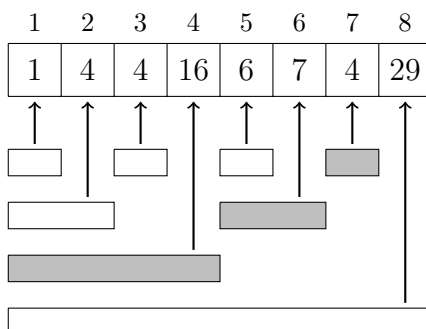
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

Hình sau cho thấy rõ hơn mỗi giá trị trong cây chỉ số nhị phân tương ứng với một đoạn trong mảng ban đầu như thế nào:



Dùng cây chỉ số nhị phân, bất kỳ giá trị nào của $\text{sum}_q(1, k)$ cũng có thể được tính trong thời gian $O(\log n)$, vì một đoạn $[1, k]$ luôn có thể được chia thành $O(\log n)$ đoạn mà tổng của chúng đã được lưu trong cây.

Ví dụ, đoạn $[1, 7]$ gồm các đoạn sau:



Do đó, ta có thể tính tổng tương ứng như sau:

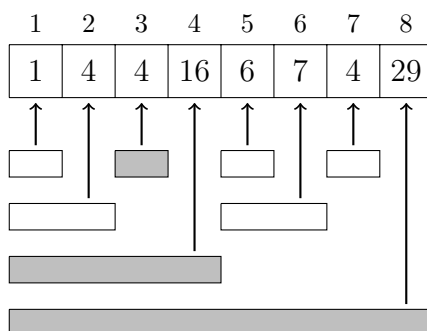
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

Để tính giá trị của $\text{sum}_q(a, b)$ với $a > 1$, ta có thể dùng cùng thủ thuật đã dùng với mảng tổng tiền tố:

$$\text{sum}_q(a, b) = \text{sum}_q(1, b) - \text{sum}_q(1, a - 1).$$

Vì ta có thể tính cả $\text{sum}_q(1, b)$ và $\text{sum}_q(1, a - 1)$ trong thời gian $O(\log n)$, độ phức tạp thời gian tổng cộng là $O(\log n)$.

Sau đó, khi cập nhật một giá trị trong mảng ban đầu, một số giá trị trong cây chỉ số nhị phân cũng phải được cập nhật. Ví dụ, nếu giá trị tại vị trí 3 thay đổi, tổng của các đoạn sau thay đổi:



Vì mỗi phần tử của mảng thuộc $O(\log n)$ đoạn trong cây chỉ số nhị phân, chỉ cần cập nhật $O(\log n)$ giá trị trong cây.

Cài đặt

Các thao tác của cây chỉ số nhị phân có thể được cài đặt hiệu quả bằng các phép toán bit. Sự kiện then chốt cần dùng là ta có thể tính bất kỳ giá trị nào của $p(k)$ bằng công thức

$$p(k) = k \& -k.$$

Hàm sau tính giá trị của $\text{sum}_q(1, k)$:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k & -k;
    }
    return s;
}
```

Hàm sau tăng giá trị của mảng tại vị trí k thêm x (x có thể dương hoặc âm):


```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

Độ phức tạp thời gian của cả hai hàm là $O(\log n)$, vì các hàm truy cập $O(\log n)$ giá trị trong cây chỉ số nhị phân, và mỗi bước chuyển sang vị trí tiếp theo mất thời gian $O(1)$.

9.3 Cây đoạn

Segment tree (cây đoạn)³ là một cấu trúc dữ liệu hỗ trợ hai thao tác: xử lý một truy vấn đoạn và cập nhật một giá trị của mảng. Cây đoạn có thể hỗ trợ truy vấn tổng, truy vấn nhỏ nhất và lớn nhất, cùng nhiều loại truy vấn khác sao cho cả hai thao tác đều chạy trong thời gian $O(\log n)$.

So với cây chỉ số nhị phân, ưu điểm của cây đoạn là nó là một cấu trúc dữ liệu tổng quát hơn. Trong khi cây chỉ số nhị phân chỉ hỗ trợ truy vấn tổng⁴, cây đoạn còn hỗ trợ các truy vấn khác. Mặt khác, cây đoạn cần nhiều bộ nhớ hơn và cài đặt khó hơn một chút.

Cấu trúc

Cây đoạn là một cây nhị phân sao cho các nút ở mức dưới cùng của cây tương ứng với các phần tử của mảng, còn các nút khác chứa thông tin cần thiết để xử lý truy vấn đoạn.

Trong phần này, ta giả sử kích thước của mảng là một lũy thừa của hai và dùng đánh chỉ số từ không, vì như vậy thuận tiện để xây dựng cây đoạn cho mảng đó. Nếu kích thước của mảng không phải là lũy thừa của hai, ta luôn có thể thêm các phần tử phụ vào cuối mảng.

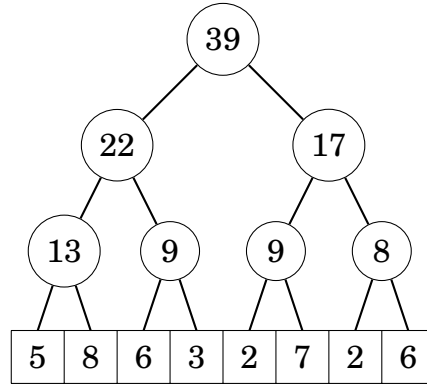
Trước hết, ta sẽ thảo luận về các cây đoạn hỗ trợ truy vấn tổng. Ví dụ, xét mảng sau:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

Cây đoạn tương ứng như sau:

³Cài đặt từ dưới lên trong chương này tương ứng với cài đặt trong [62]. Các cấu trúc tương tự đã được dùng vào cuối thập niên 1970 để giải các bài toán hình học [9].

⁴Thực ra, dùng hai cây chỉ số nhị phân thì có thể hỗ trợ truy vấn nhỏ nhất [16], nhưng cách này phức tạp hơn so với dùng cây đoạn.

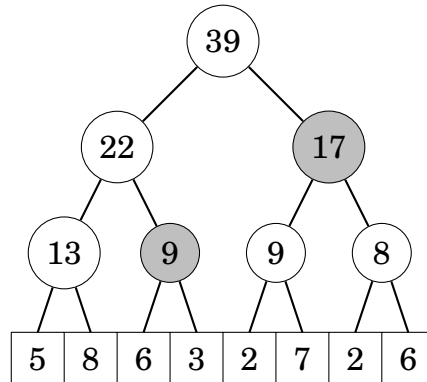


Mỗi nút trong của cây tương ứng với một đoạn của mảng có kích thước là một lũy thừa của hai. Trong cây trên, giá trị của mỗi nút trong là tổng các giá trị mảng tương ứng, và có thể được tính bằng tổng giá trị của nút con trái và nút con phải của nó.

Hóa ra bất kỳ đoạn $[a, b]$ nào cũng có thể được chia thành $O(\log n)$ đoạn mà giá trị của chúng được lưu trong các nút của cây. Ví dụ, xét đoạn $[2, 7]$:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

Ở đây $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. Trong trường hợp này, hai nút cây sau tương ứng với đoạn đó:

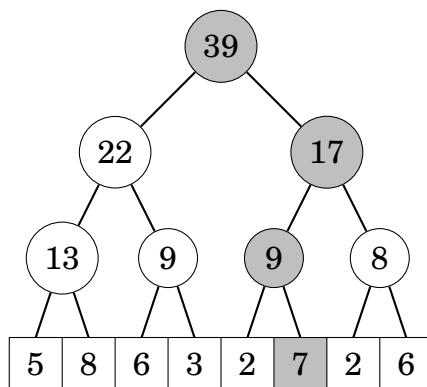


Do đó, một cách khác để tính tổng là $9 + 17 = 26$.

Khi tổng được tính bằng các nút nằm cao nhất có thể trong cây, mỗi mức của cây cần nhiều nhất hai nút. Vì vậy, tổng số nút là $O(\log n)$.

Sau một lần cập nhật mảng, ta cần cập nhật tất cả các nút có giá trị phụ thuộc vào giá trị đã được cập nhật. Việc này có thể được thực hiện bằng cách đi theo đường đi từ phần tử mảng vừa cập nhật lên nút gốc và cập nhật các nút trên đường đi đó.

Hình sau cho thấy những nút cây nào thay đổi nếu giá trị 7 trong mảng thay đổi:

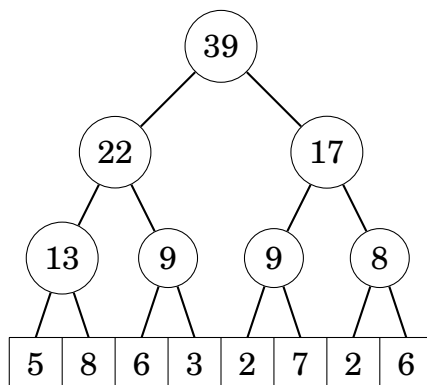


Đường đi từ dưới lên trên luôn gồm $O(\log n)$ nút, vì vậy mỗi lần cập nhật làm thay đổi $O(\log n)$ nút trong cây.

Cài đặt

Ta lưu cây đoạn dưới dạng một mảng gồm $2n$ phần tử, trong đó n là kích thước của mảng ban đầu và là một lũy thừa của hai. Các nút cây được lưu từ trên xuống dưới: $\text{tree}[1]$ là nút gốc, $\text{tree}[2]$ và $\text{tree}[3]$ là các nút con của nó, và cứ tiếp tục như vậy. Cuối cùng, các giá trị từ $\text{tree}[n]$ đến $\text{tree}[2n - 1]$ tương ứng với các giá trị của mảng ban đầu ở mức dưới cùng của cây.

Ví dụ, cây đoạn



được lưu như sau:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Dùng cách biểu diễn này, cha của $\text{tree}[k]$ là $\text{tree}[\lfloor k/2 \rfloor]$, và các con của nó là $\text{tree}[2k]$ và $\text{tree}[2k + 1]$. Lưu ý rằng điều này kéo theo vị trí của một nút là chẵn nếu nó là con trái và lẻ nếu nó là con phải.

Hàm sau tính giá trị của $\text{sum}_q(a, b)$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
```

```

while (a <= b) {
    if (a%2 == 1) s += tree[a++];
    if (b%2 == 0) s += tree[b--];
    a /= 2; b /= 2;
}
return s;
}

```

Hàm duy trì một đoạn ban đầu là $[a + n, b + n]$. Sau đó, ở mỗi bước, đoạn này được đưa lên một mức cao hơn trong cây, và trước đó, giá trị của các nút không thuộc đoạn ở mức cao hơn được cộng vào tổng.

Hàm sau tăng giá trị của mảng tại vị trí k thêm x :

```

void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}

```

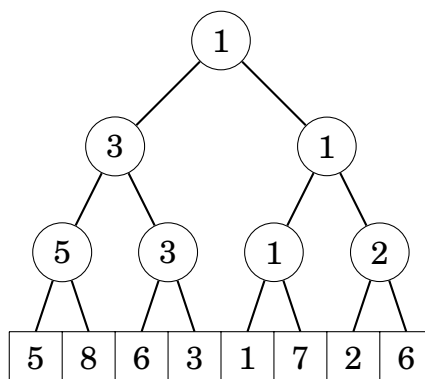
Trước hết, hàm cập nhật giá trị ở mức dưới cùng của cây. Sau đó, hàm cập nhật giá trị của tất cả các nút trong, cho đến khi chạm tới nút gốc của cây.

Cả hai hàm trên đều chạy trong thời gian $O(\log n)$, vì một cây đoạn của n phần tử gồm $O(\log n)$ mức, và ở mỗi bước các hàm đi lên một mức trong cây.

Các truy vấn khác

Cây đoạn có thể hỗ trợ mọi truy vấn đoạn mà ta có thể chia một đoạn thành hai phần, tính riêng câu trả lời cho từng phần rồi kết hợp các câu trả lời một cách hiệu quả. Ví dụ về các truy vấn như vậy là nhỏ nhất và lớn nhất, ước chung lớn nhất, và các phép toán bit and, or và xor.

Ví dụ, cây đoạn sau hỗ trợ các truy vấn nhỏ nhất:

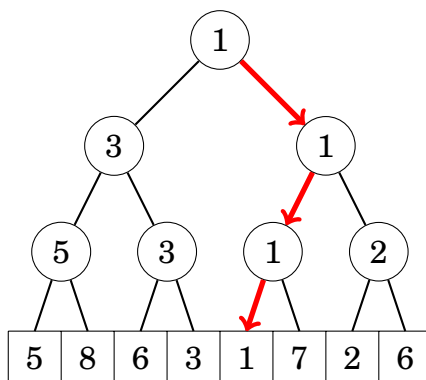


Trong trường hợp này, mỗi nút cây chứa giá trị nhỏ nhất trong đoạn mảng tương ứng. Nút gốc của cây chứa giá trị nhỏ nhất trong toàn bộ mảng.

Các thao tác có thể được cài đặt như trước, nhưng thay vì tính tổng, ta tính giá trị nhỏ nhất.

Cấu trúc của cây đoạn cũng cho phép ta dùng tìm kiếm nhị phân để định vị các phần tử của mảng. Ví dụ, nếu cây hỗ trợ truy vấn nhỏ nhất, ta có thể tìm vị trí của một phần tử có giá trị nhỏ nhất trong thời gian $O(\log n)$.

Ví dụ, trong cây trên, một phần tử có giá trị nhỏ nhất 1 có thể được tìm bằng cách đi theo một đường đi xuống từ nút gốc:



9.4 Các kỹ thuật bổ sung

Nén chỉ số

Một hạn chế của các cấu trúc dữ liệu được xây dựng dựa trên mảng là các phần tử được đánh chỉ số bằng các số nguyên liên tiếp. Khó khăn xuất hiện khi cần dùng các chỉ số lớn. Ví dụ, nếu ta muốn dùng chỉ số 10^9 , mảng sẽ phải chứa 10^9 phần tử, đòi hỏi quá nhiều bộ nhớ.

Tuy nhiên, ta thường có thể vượt qua hạn chế này bằng cách dùng **index compression** (nén chỉ số), trong đó các chỉ số ban đầu được thay thế bằng các chỉ số 1, 2, 3, v.v. Điều này có thể thực hiện được nếu ta biết trước tất cả các chỉ số cần dùng trong thuật toán.

Ý tưởng là thay mỗi chỉ số ban đầu x bằng $c(x)$, trong đó c là một hàm nén các chỉ số. Ta yêu cầu thứ tự của các chỉ số không thay đổi, nên nếu $a < b$, thì $c(a) < c(b)$. Điều này cho phép ta thực hiện truy vấn thuận tiện ngay cả khi các chỉ số đã được nén.

Ví dụ, nếu các chỉ số ban đầu là 555, 10^9 và 8, các chỉ số mới là:

$$\begin{aligned} c(8) &= 1 \\ c(555) &= 2 \\ c(10^9) &= 3 \end{aligned}$$

Cập nhật đoạn

Cho đến nay, ta đã cài đặt các cấu trúc dữ liệu hỗ trợ truy vấn đoạn và cập nhật các giá trị đơn lẻ. Bây giờ hãy xét tình huống ngược lại, trong đó ta cần

cập nhật các đoạn và lấy các giá trị đơn lẻ. Ta tập trung vào thao tác tăng tất cả các phần tử trong một đoạn $[a, b]$ thêm x .

Điều đáng ngạc nhiên là ta cũng có thể dùng các cấu trúc dữ liệu đã trình bày trong chương này cho tình huống đó. Để làm vậy, ta xây dựng một **difference array** (mảng hiệu) mà các giá trị của nó cho biết hiệu giữa các giá trị liên tiếp trong mảng ban đầu. Do đó, mảng ban đầu chính là mảng tổng tiền tố của mảng hiệu. Ví dụ, xét mảng sau:

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2

Mảng hiệu của mảng trên như sau:

0	1	2	3	4	5	6	7
3	0	-2	0	0	4	-3	0

Ví dụ, giá trị 2 tại vị trí 6 trong mảng ban đầu tương ứng với tổng $3 - 2 + 4 - 3 = 2$ trong mảng hiệu.

Ưu điểm của mảng hiệu là ta có thể cập nhật một đoạn trong mảng ban đầu bằng cách chỉ thay đổi hai phần tử trong mảng hiệu. Ví dụ, nếu ta muốn tăng các giá trị của mảng ban đầu từ vị trí 1 đến 4 thêm 5, chỉ cần tăng giá trị của mảng hiệu tại vị trí 1 thêm 5 và giảm giá trị tại vị trí 5 đi 5. Kết quả như sau:

0	1	2	3	4	5	6	7
3	5	-2	0	0	-1	-3	0

Tổng quát hơn, để tăng các giá trị trong đoạn $[a, b]$ thêm x , ta tăng giá trị tại vị trí a thêm x và giảm giá trị tại vị trí $b + 1$ đi x . Vì vậy, ta chỉ cần cập nhật các giá trị đơn lẻ và xử lý truy vấn tổng, nên có thể dùng cây chỉ số nhị phân hoặc cây đoạn.

Một bài toán khó hơn là hỗ trợ cả truy vấn đoạn lẫn cập nhật đoạn. Trong Chương 28, ta sẽ thấy rằng ngay cả điều này cũng có thể làm được.

Thao tác bit

10.1 Biểu diễn bit

Đây là biểu diễn bit của số int 43:

Các bit trong biểu diễn được đánh chỉ số từ phải sang trái. Để chuyển biểu diễn bit $b_k \cdots b_2 b_1 b_0$ thành một số, ta có thể dùng công thức

$$b_k 2^k + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

Biểu diễn bit của một số có thể là **signed** hoặc **unsigned**. Thông thường, ta dùng biểu diễn có dấu, nghĩa là có thể biểu diễn cả số âm và số dương. Một biến có dấu gồm n bit có thể chứa mọi số nguyên từ -2^{n-1} đến $2^{n-1} - 1$. Ví dụ, kiểu `int` trong C++ là một kiểu có dấu, nên một biến `int` có thể chứa mọi số nguyên từ -2^{31} đến $2^{31} - 1$.

Bit đầu tiên trong một biểu diễn có dấu là dấu của số (0 cho số không âm và 1 cho số âm), và $n - 1$ bit còn lại chứa độ lớn của số. **Two's complement** được sử dụng, nghĩa là số đối của một số được tính bằng cách trước hết đảo tất cả các bit trong số đó, rồi tăng số lên một.

Ví dụ, biểu diễn bit của số int -43 là

1111111111111111111111111111010101.

Trong biểu diễn không dấu, chỉ có thể dùng các số không âm, nhưng cận trên của giá trị lớn hơn. Một biến không dấu gồm n bit có thể chứa mọi số nguyên từ 0 đến $2^n - 1$. Ví dụ, trong C++, một biến unsigned int có thể chứa mọi số nguyên từ 0 đến $2^{32} - 1$.

Có một mối liên hệ giữa các biểu diễn: một số có dấu $-x$ bằng với một số không dấu $2^n - x$. Ví dụ, đoạn mã sau cho thấy số có dấu $x = -43$ bằng với số không dấu $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Nếu một số lớn hơn cận trên của biểu diễn bit, số đó sẽ bị tràn. Trong biểu diễn có dấu, số kế tiếp sau $2^{n-1} - 1$ là -2^{n-1} , và trong biểu diễn không dấu, số kế tiếp sau $2^n - 1$ là 0. Ví dụ, xét đoạn mã sau:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Ban đầu, giá trị của x là $2^{31} - 1$. Đây là giá trị lớn nhất có thể được lưu trong một biến int, nên số kế tiếp sau $2^{31} - 1$ là -2^{31} .

10.2 Các phép toán bit

Phép and

Phép toán **and** $x \& y$ tạo ra một số có các bit một ở những vị trí mà cả x và y đều có bit một. Ví dụ, $22 \& 26 = 18$, vì

$$\begin{array}{r} 10110 \quad (22) \\ \& 11010 \quad (26) \\ \hline = 10010 \quad (18) \end{array}$$

Dùng phép and, ta có thể kiểm tra một số x có chẵn hay không vì $x \& 1 = 0$ nếu x chẵn, và $x \& 1 = 1$ nếu x lẻ. Tổng quát hơn, x chia hết cho 2^k đúng khi và chỉ khi $x \& (2^k - 1) = 0$.

Phép or

Phép toán **or** $x \mid y$ tạo ra một số có các bit một ở những vị trí mà ít nhất một trong x và y có bit một. Ví dụ, $22 \mid 26 = 30$, vì

$$\begin{array}{r} 10110 \quad (22) \\ \mid 11010 \quad (26) \\ \hline = 11110 \quad (30) \end{array}$$

Phép xor

Phép toán **xor** $x \wedge y$ tạo ra một số có các bit một ở những vị trí mà đúng một trong x và y có bit một. Ví dụ, $22 \wedge 26 = 12$, vì

$$\begin{array}{r} 10110 \quad (22) \\ \wedge \quad 11010 \quad (26) \\ \hline = \quad 01100 \quad (12) \end{array}$$

Phép not

Phép toán **not** $\sim x$ tạo ra một số trong đó tất cả các bit của x đã bị đảo. Công thức $\sim x = -x - 1$ luôn đúng, ví dụ, $\sim 29 = -30$.

Kết quả của phép not ở mức bit phụ thuộc vào độ dài của biểu diễn bit, vì phép toán này đảo tất cả các bit. Ví dụ, nếu các số là các số int 32 bit, kết quả như sau:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000000011101 \\ \sim x & = & -30 \quad 111111111111111111111111111110010 \end{array}$$

Dịch bit

Phép dịch bit trái $x \ll k$ thêm k bit không vào cuối số, và phép dịch bit phải $x \gg k$ loại bỏ k bit cuối của số. Ví dụ, $14 \ll 2 = 56$, vì 14 và 56 tương ứng với 1110 và 111000. Tương tự, $49 \gg 3 = 6$, vì 49 và 6 tương ứng với 110001 và 110.

Lưu ý rằng $x \ll k$ tương ứng với việc nhân x với 2^k , và $x \gg k$ tương ứng với việc chia x cho 2^k rồi làm tròn xuống thành một số nguyên.

Ứng dụng

Một số có dạng $1 \ll k$ có một bit một ở vị trí k và tất cả các bit khác bằng không, nên ta có thể dùng những số như vậy để truy cập từng bit riêng lẻ của số. Cụ thể, bit thứ k của một số là một đúng khi và chỉ khi $x \& (1 \ll k)$ khác không. Đoạn mã sau in biểu diễn bit của một số int x :

```
for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}
```

Cũng có thể sửa đổi từng bit riêng lẻ của số bằng các ý tưởng tương tự. Ví dụ, công thức $x | (1 \ll k)$ đặt bit thứ k của x thành một, công thức $x \& \sim(1 \ll k)$ đặt bit thứ k của x thành không, và công thức $x \wedge (1 \ll k)$ đảo bit thứ k của x .

Công thức $x \& (x - 1)$ đặt bit một cuối cùng của x thành không, và công thức $x \& -x$ đặt tất cả các bit một thành không, ngoại trừ bit một cuối cùng.

Công thức $x \mid (x - 1)$ đảo tất cả các bit sau bit một cuối cùng. Ngoài ra, lưu ý rằng một số dương x là một lũy thừa của hai đúng khi và chỉ khi $x \& (x - 1) = 0$.

Các hàm bổ sung

Trình biên dịch `g++` cung cấp các hàm sau để đếm bit:

- `__builtin_clz(x)`: số lượng số không ở đầu số
- `__builtin_ctz(x)`: số lượng số không ở cuối số
- `__builtin_popcount(x)`: số lượng bit một trong số
- `__builtin_parity(x)`: tính chẵn lẻ (even or odd) của số lượng bit một

Các hàm có thể được sử dụng như sau:

```
int x = 5328; // 000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Trong khi các hàm trên chỉ hỗ trợ số `int`, cũng có các phiên bản `long` `long` của các hàm này với hậu tố `ll`.

10.3 Biểu diễn tập hợp

Mọi tập con của một tập $\{0, 1, 2, \dots, n - 1\}$ có thể được biểu diễn bằng một số nguyên n bit mà các bit một cho biết những phần tử nào thuộc về tập con. Đây là một cách hiệu quả để biểu diễn tập hợp, vì mỗi phần tử chỉ cần một bit bộ nhớ, và các phép toán trên tập hợp có thể được cài đặt bằng phép toán bit.

Ví dụ, vì `int` là một kiểu 32 bit, một số `int` có thể biểu diễn bất kỳ tập con nào của tập $\{0, 1, 2, \dots, 31\}$. Biểu diễn bit của tập $\{1, 3, 4, 8\}$ là

000000000000000000000000100011010,

tương ứng với số $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Cài đặt tập hợp

Đoạn mã sau khai báo một biến `int x` có thể chứa một tập con của $\{0, 1, 2, \dots, 31\}$. Sau đó, đoạn mã thêm các phần tử 1, 3, 4 và 8 vào tập hợp và in kích thước của tập hợp.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Tiếp theo, đoạn mã sau in tất cả các phần tử thuộc về tập hợp:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

Phép toán tập hợp

Các phép toán tập hợp có thể được cài đặt như sau bằng phép toán bit:

	cú pháp tập hợp	cú pháp bit
giao	$a \cap b$	$a \& b$
hợp	$a \cup b$	$a b$
bù	$\bar{a}$	$\sim a$
hiệu	$a \setminus b$	$a \& (\sim b)$

Ví dụ, đoạn mã sau trước hết xây dựng các tập $x = \{1, 3, 4, 8\}$ và $y = \{3, 6, 8, 9\}$, rồi xây dựng tập $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Duyệt các tập con

Đoạn mã sau duyệt qua các tập con của $\{0, 1, \dots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

Đoạn mã sau duyệt qua các tập con có đúng k phần tử:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

Đoạn mã sau duyệt qua các tập con của một tập x :

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

10.4 Tối ưu hóa bằng bit

Nhiều thuật toán có thể được tối ưu hóa bằng các phép toán bit. Những tối ưu hóa như vậy không thay đổi độ phức tạp thời gian của thuật toán, nhưng chúng có thể có ảnh hưởng lớn đến thời gian chạy thực tế của mã. Trong mục này, ta thảo luận một số ví dụ về các tình huống như vậy.

Khoảng cách Hamming

Hamming distance $\text{hamming}(a, b)$ giữa hai chuỗi a và b có cùng độ dài là số vị trí mà hai chuỗi khác nhau. Ví dụ,

$$\text{hamming}(01101, 11001) = 2.$$

Xét bài toán sau: Cho một danh sách gồm n chuỗi bit, mỗi chuỗi có độ dài k , tính khoảng cách Hamming nhỏ nhất giữa hai chuỗi trong danh sách. Ví dụ, đáp án cho $[00111, 01101, 11110]$ là 2, vì

- $\text{hamming}(00111, 01101) = 2$,
- $\text{hamming}(00111, 11110) = 3$, và
- $\text{hamming}(01101, 11110) = 3$.

Một cách trực tiếp để giải bài toán là duyệt qua tất cả các cặp chuỗi và tính khoảng cách Hamming của chúng, tạo ra một thuật toán thời gian $O(n^2k)$. Hàm sau có thể được dùng để tính khoảng cách:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

Tuy nhiên, nếu k nhỏ, ta có thể tối ưu hóa mã bằng cách lưu các chuỗi bit dưới dạng số nguyên và tính khoảng cách Hamming bằng phép toán bit. Cụ thể, nếu $k \leq 32$, ta chỉ cần lưu các chuỗi dưới dạng giá trị `int` và dùng hàm sau để tính khoảng cách:

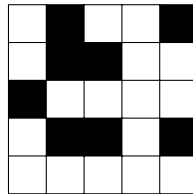
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

Trong hàm trên, phép xor xây dựng một xâu bit có các bit một ở những vị trí mà a và b khác nhau. Sau đó, số lượng bit được tính bằng hàm `__builtin_popcount`.

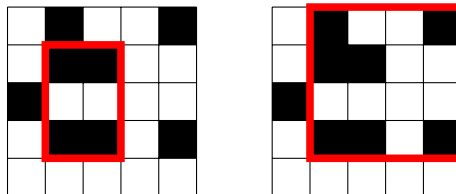
Để so sánh các cách cài đặt, chúng tôi sinh một danh sách gồm 10000 xâu bit ngẫu nhiên có độ dài 30. Với cách đầu tiên, việc tìm kiếm mất 13.5 giây, và sau khi tối ưu hóa bằng bit, nó chỉ mất 0.5 giây. Vì vậy, mã được tối ưu hóa bằng bit nhanh hơn gần 30 lần so với mã ban đầu.

Đếm lưới con

Là một ví dụ khác, xét bài toán sau: Cho một lưới $n \times n$ mà mỗi ô là đen (1) hoặc trắng (0), tính số lưới con có tất cả các góc đều màu đen. Ví dụ, lưới



chứa hai lưới con như vậy:



Có một thuật toán thời gian $O(n^3)$ để giải bài toán: duyệt qua tất cả $O(n^2)$ cặp hàng và với mỗi cặp (a, b) tính số cột có một ô đen ở cả hai hàng trong thời gian $O(n)$. Đoạn mã sau giả sử rằng `color[y][x]` biểu thị màu ở hàng y và cột x :

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Khi đó, các cột đó đóng góp $\text{count}(\text{count} - 1)/2$ lưới con có góc đen, vì ta có thể chọn bất kỳ hai cột nào trong chúng để tạo thành một lưới con.

Để tối ưu hóa thuật toán này, ta chia lưới thành các khối cột sao cho mỗi khối gồm N cột liên tiếp. Sau đó, mỗi hàng được lưu dưới dạng một danh sách các số N bit mô tả màu của các ô. Bây giờ ta có thể xử lý N cột cùng lúc bằng phép toán bit. Trong đoạn mã sau, `color[y][k]` biểu diễn một khối gồm N màu dưới dạng bit.

```

int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}

```

Thuật toán thu được chạy trong thời gian $O(n^3/N)$.

Chúng tôi sinh một lưới ngẫu nhiên kích thước 2500×2500 và so sánh cách cài đặt ban đầu với cách cài đặt được tối ưu hóa bằng bit. Trong khi mã ban đầu mất 29.6 giây, phiên bản tối ưu hóa bằng bit chỉ mất 3.1 giây với $N = 32$ (các số int) và 1.7 giây với $N = 64$ (các số long long).

10.5 Quy hoạch động

Các phép toán bit cung cấp một cách hiệu quả và tiện lợi để cài đặt các thuật toán quy hoạch động mà trạng thái chứa các tập con của các phần tử, vì những trạng thái như vậy có thể được lưu dưới dạng số nguyên. Tiếp theo, ta thảo luận các ví dụ về việc kết hợp phép toán bit và quy hoạch động.

Lựa chọn tối ưu

Là ví dụ đầu tiên, xét bài toán sau: Ta được cho giá của k sản phẩm trong n ngày, và muốn mua mỗi sản phẩm đúng một lần. Tuy nhiên, ta chỉ được phép mua nhiều nhất một sản phẩm trong một ngày. Tổng giá nhỏ nhất là bao nhiêu? Ví dụ, xét tình huống sau ($k = 3$ và $n = 8$):

	0	1	2	3	4	5	6	7
sản phẩm 0	6	9	5	2	8	9	1	6
sản phẩm 1	8	2	6	2	7	5	7	2
sản phẩm 2	5	3	9	7	3	5	1	4

Trong tình huống này, tổng giá nhỏ nhất là 5:

	0	1	2	3	4	5	6	7
sản phẩm 0	6	9	5	2	8	9	1	6
sản phẩm 1	8	2	6	2	7	5	7	2
sản phẩm 2	5	3	9	7	3	5	1	4

Gọi $\text{price}[x][d]$ là giá của sản phẩm x vào ngày d . Ví dụ, trong tình huống trên $\text{price}[2][3] = 7$. Tiếp theo, gọi $\text{total}(S, d)$ là tổng giá nhỏ nhất để mua một tập con S các sản phẩm trước hoặc trong ngày d . Dùng hàm này, lời giải của bài toán là $\text{total}(\{0 \dots k-1\}, n-1)$.

Trước hết, $\text{total}(\emptyset, d) = 0$, vì không tốn chi phí gì để mua một tập rỗng, và $\text{total}(\{x\}, 0) = \text{price}[x][0]$, vì có một cách để mua một sản phẩm vào ngày

đầu tiên. Sau đó, có thể dùng công thức truy hồi sau:

$$\text{total}(S, d) = \min(\text{total}(S, d - 1), \min_{x \in S} (\text{total}(S \setminus x, d - 1) + \text{price}[x][d]))$$

Điều này có nghĩa là ta hoặc không mua sản phẩm nào vào ngày d , hoặc mua một sản phẩm x thuộc S . Trong trường hợp sau, ta loại x khỏi S và cộng giá của x vào tổng giá.

Bước tiếp theo là tính các giá trị của hàm bằng quy hoạch động. Để lưu các giá trị của hàm, ta khai báo một mảng

```
int total[1<<K][N];
```

trong đó K và N là các hằng đủ lớn. Chiều đầu tiên của mảng tương ứng với một biểu diễn bit của một tập con.

Trước hết, các trường hợp $d = 0$ có thể được xử lý như sau:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Sau đó, công thức truy hồi được chuyển thành đoạn mã sau:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s & (1<<x)) {
                total[s][d] = min(total[s][d],
                                   total[s^(1<<x)][d-1] + price[x][d]);
            }
        }
    }
}
```

Độ phức tạp thời gian của thuật toán là $O(n2^k k)$.

Từ hoán vị đến tập con

Dùng quy hoạch động, thường có thể chuyển một vòng lặp trên các hoán vị thành một vòng lặp trên các tập con¹. Lợi ích của việc này là $n!$, số lượng hoán vị, lớn hơn rất nhiều so với 2^n , số lượng tập con. Ví dụ, nếu $n = 20$, thì $n! \approx 2.4 \cdot 10^{18}$ và $2^n \approx 10^6$. Do đó, với một số giá trị của n , ta có thể duyệt các tập con một cách hiệu quả nhưng không thể duyệt các hoán vị.

Là một ví dụ, xét bài toán sau: Có một thang máy với trọng lượng tối đa x , và n người với trọng lượng đã biết muốn đi từ tầng trệt lên tầng cao nhất.

¹Kỹ thuật này được giới thiệu năm 1962 bởi M. Held và R. M. Karp [34].

Cần ít nhất bao nhiêu lượt đi nếu mọi người vào thang máy theo một thứ tự tối ưu?

Ví dụ, giả sử $x = 10$, $n = 5$ và các trọng lượng như sau:

người	trọng lượng
0	2
1	3
2	3
3	5
4	6

Trong trường hợp này, số lượt đi nhỏ nhất là 2. Một thứ tự tối ưu là $\{0, 2, 3, 1, 4\}$, chia mọi người thành hai lượt đi: đầu tiên $\{0, 2, 3\}$ (tổng trọng lượng 10), và sau đó $\{1, 4\}$ (tổng trọng lượng 9).

Bài toán có thể được giải dễ dàng trong thời gian $O(n!n)$ bằng cách thử tất cả các hoán vị có thể của n người. Tuy nhiên, ta có thể dùng quy hoạch động để có một thuật toán hiệu quả hơn với thời gian $O(2^n n)$. Ý tưởng là tính cho mỗi tập con người hai giá trị: số lượt đi nhỏ nhất cần thiết và trọng lượng nhỏ nhất của những người đi trong nhóm cuối cùng.

Gọi $\text{weight}[p]$ là trọng lượng của người p . Ta định nghĩa hai hàm: $\text{rides}(S)$ là số lượt đi nhỏ nhất cho một tập con S , và $\text{last}(S)$ là trọng lượng nhỏ nhất của lượt đi cuối cùng. Ví dụ, trong tình huống trên

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{và} \quad \text{last}(\{1, 3, 4\}) = 5,$$

vì các lượt đi tối ưu là $\{1, 4\}$ và $\{3\}$, và lượt đi thứ hai có trọng lượng 5. Dĩ nhiên, mục tiêu cuối cùng của ta là tính giá trị của $\text{rides}(\{0 \dots n-1\})$.

Ta có thể tính các giá trị của các hàm một cách đệ quy rồi áp dụng quy hoạch động. Ý tưởng là duyệt qua tất cả những người thuộc S và chọn tối ưu người cuối cùng p bước vào thang máy. Mỗi lựa chọn như vậy tạo ra một bài toán con cho một tập người nhỏ hơn. Nếu $\text{last}(S \setminus p) + \text{weight}[p] \leq x$, ta có thể thêm p vào lượt đi cuối cùng. Ngược lại, ta phải dành một lượt đi mới mà ban đầu chỉ chứa p .

Để cài đặt quy hoạch động, ta khai báo một mảng

```
pair<int,int> best[1<<N];
```

chứa cho mỗi tập con S một cặp $(\text{rides}(S), \text{last}(S))$. Ta đặt giá trị cho nhóm rỗng như sau:

```
best[0] = {1,0};
```

Sau đó, ta có thể điền mảng như sau:

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
```



```

    auto option = best[s^(1<<p)];
    if (option.second+weight[p] <= x) {
        // add p to an existing ride
        option.second += weight[p];
    } else {
        // reserve a new ride for p
        option.first++;
        option.second = weight[p];
    }
    best[s] = min(best[s], option);
}
}
}

```

Lưu ý rằng vòng lặp trên đảm bảo rằng với bất kỳ hai tập con S_1 và S_2 sao cho $S_1 \subset S_2$, ta xử lý S_1 trước S_2 . Do đó, các giá trị quy hoạch động được tính theo đúng thứ tự.

Đếm tập con

Bài toán cuối cùng của chúng ta trong chương này như sau: Cho $X = \{0 \dots n-1\}$, và mỗi tập con $S \subset X$ được gán một số nguyên $\text{value}[S]$. Nhiệm vụ của ta là tính cho mỗi S

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

tức là tổng giá trị của các tập con của S .

Ví dụ, giả sử $n = 3$ và các giá trị như sau:

- $\text{value}[\emptyset] = 3$
- $\text{value}[\{0\}] = 1$
- $\text{value}[\{1\}] = 4$
- $\text{value}[\{0, 1\}] = 5$
- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0, 2\}] = 1$
- $\text{value}[\{1, 2\}] = 3$
- $\text{value}[\{0, 1, 2\}] = 3$

Trong trường hợp này, ví dụ,

$$\begin{aligned} \text{sum}(\{0, 2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

Vì có tổng cộng 2^n tập con, một lời giải khả dĩ là duyệt qua tất cả các cặp tập con trong thời gian $O(2^{2n})$. Tuy nhiên, bằng quy hoạch động, ta có thể giải bài toán trong thời gian $O(2^n n)$. Ý tưởng là tập trung vào các tổng mà các phần tử có thể được loại khỏi S bị giới hạn.

Gọi $\text{partial}(S, k)$ là tổng giá trị của các tập con của S với ràng buộc rằng chỉ các phần tử $0 \dots k$ có thể bị loại khỏi S . Ví dụ,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

vì ta chỉ có thể loại các phần tử $0 \dots 1$. Ta có thể tính các giá trị của sum bằng các giá trị của partial , vì

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Các trường hợp cơ sở của hàm là

$$\text{partial}(S, -1) = \text{value}[S],$$

vì trong trường hợp này không phần tử nào có thể bị loại khỏi S . Sau đó, trong trường hợp tổng quát, ta có thể dùng công thức truy hồi sau:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Ở đây ta tập trung vào phần tử k . Nếu $k \in S$, ta có hai lựa chọn: hoặc giữ k trong S hoặc loại nó khỏi S .

Có một cách đặc biệt khéo léo để cài đặt việc tính các tổng. Ta có thể khai báo một mảng

```
int sum[1<<N];
```

sẽ chứa tổng của mỗi tập con. Mảng được khởi tạo như sau:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Sau đó, ta có thể điền mảng như sau:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s & (1<<k)) sum[s] += sum[s ^ (1<<k)];
    }
}
```

Đoạn mã này tính các giá trị của $\text{partial}(S, k)$ với $k = 0 \dots n - 1$ vào mảng sum . Vì $\text{partial}(S, k)$ luôn dựa trên $\text{partial}(S, k - 1)$, ta có thể tái sử dụng mảng sum , tạo ra một cách cài đặt rất hiệu quả.

Phần II

Thuật toán đồ thị

Chương 11

Cơ bản về đồ thị

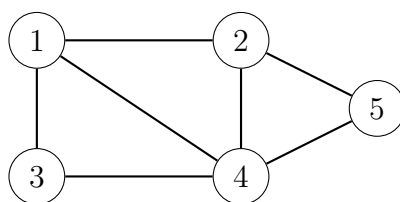
Nhiều bài toán lập trình có thể được giải bằng cách mô hình hóa bài toán thành một bài toán đồ thị và sử dụng một thuật toán đồ thị thích hợp. Một ví dụ điển hình của đồ thị là mạng lưới đường sá và các thành phố trong một quốc gia. Tuy vậy, đôi khi đồ thị bị ẩn trong bài toán và có thể khó nhận ra nó.

Phần này của cuốn sách bàn về các thuật toán đồ thị, đặc biệt tập trung vào những chủ đề quan trọng trong lập trình thi đấu. Trong chương này, chúng ta xét các khái niệm liên quan đến đồ thị, và nghiên cứu những cách khác nhau để biểu diễn đồ thị trong thuật toán.

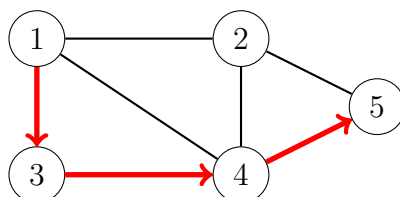
11.1 Thuật ngữ đồ thị

Một **đồ thị** (graph) gồm các **đỉnh** (node) và các **cạnh** (edge). Trong cuốn sách này, biến n biểu thị số đỉnh trong một đồ thị, và biến m biểu thị số cạnh. Các đỉnh được đánh số bằng các số nguyên $1, 2, \dots, n$.

Ví dụ, đồ thị sau gồm 5 đỉnh và 7 cạnh:



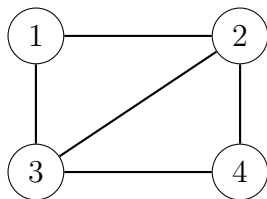
Một **đường đi** (path) dẫn từ đỉnh a đến đỉnh b qua các cạnh của đồ thị. **Độ dài** của một đường đi là số cạnh trong đường đi đó. Ví dụ, đồ thị trên chứa đường đi $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ có độ dài 3 từ đỉnh 1 đến đỉnh 5:



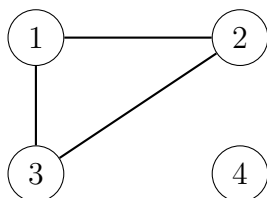
Một đường đi là một **chu trình** (cycle) nếu đỉnh đầu và đỉnh cuối là cùng một đỉnh. Ví dụ, đồ thị trên chứa chu trình $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. Một đường đi là **đơn** nếu mỗi đỉnh xuất hiện nhiều nhất một lần trong đường đi.

Tính liên thông

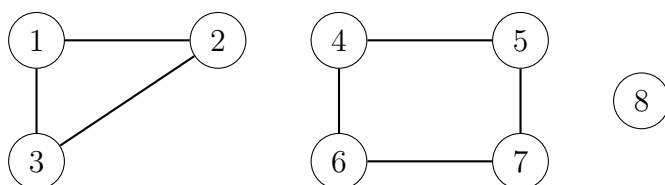
Một đồ thị là **liên thông** nếu có một đường đi giữa hai đỉnh bất kỳ. Ví dụ, đồ thị sau là liên thông:



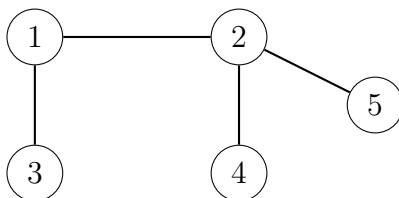
Đồ thị sau không liên thông, vì không thể đi từ đỉnh 4 đến bất kỳ đỉnh nào khác:



Các phần liên thông của một đồ thị được gọi là các **thành phần liên thông** (components) của nó. Ví dụ, đồ thị sau chứa ba thành phần liên thông: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ và $\{8\}$.

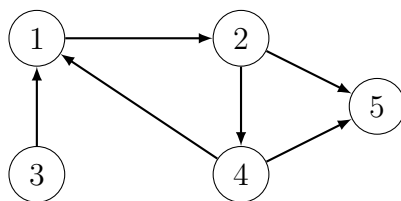


Một **cây** (tree) là một đồ thị liên thông gồm n đỉnh và $n - 1$ cạnh. Có đúng một đường đi giữa hai đỉnh bất kỳ của một cây. Ví dụ, đồ thị sau là một cây:



Hướng của cạnh

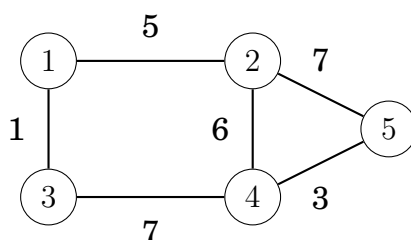
Một đồ thị là **có hướng** nếu các cạnh chỉ có thể được đi qua theo một chiều. Ví dụ, đồ thị sau là có hướng:



Đồ thị trên chứa đường đi $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ từ đỉnh 3 đến đỉnh 5, nhưng không có đường đi từ đỉnh 5 đến đỉnh 3.

Trọng số cạnh

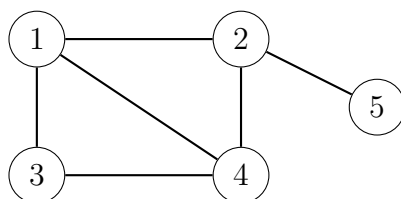
Trong một đồ thị **có trọng số**, mỗi cạnh được gán một **trọng số**. Các trọng số thường được hiểu là độ dài cạnh. Ví dụ, đồ thị sau là đồ thị có trọng số:



Độ dài của một đường đi trong đồ thị có trọng số là tổng trọng số của các cạnh trên đường đi. Ví dụ, trong đồ thị trên, độ dài của đường đi $1 \rightarrow 2 \rightarrow 5$ là 12, và độ dài của đường đi $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ là 11. Đường đi sau là đường đi **ngắn nhất** từ đỉnh 1 đến đỉnh 5.

Đỉnh kề và bậc

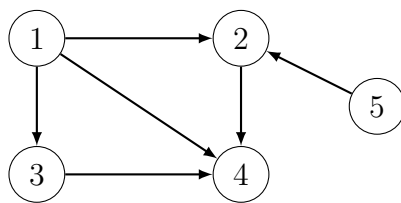
Hai đỉnh là **đỉnh kề** hoặc **kề nhau** nếu có một cạnh giữa chúng. **Bậc** của một đỉnh là số đỉnh kề của nó. Ví dụ, trong đồ thị sau, các đỉnh kề của đỉnh 2 là 1, 4 và 5, nên bậc của nó là 3.



Tổng các bậc trong một đồ thị luôn là $2m$, trong đó m là số cạnh, vì mỗi cạnh làm tăng bậc của đúng hai đỉnh thêm một. Vì lý do này, tổng các bậc luôn là số chẵn.

Một đồ thị là **chính quy** nếu bậc của mọi đỉnh là một hằng số d . Một đồ thị là **đầy đủ** nếu bậc của mọi đỉnh là $n - 1$, tức là đồ thị chứa tất cả các cạnh có thể có giữa các đỉnh.

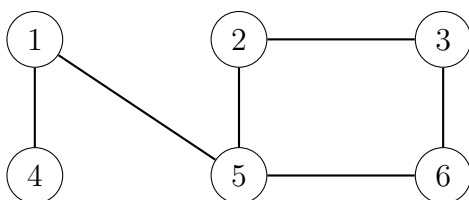
Trong một đồ thị có hướng, **bậc vào** của một đỉnh là số cạnh kết thúc tại đỉnh đó, và **bậc ra** của một đỉnh là số cạnh bắt đầu từ đỉnh đó. Ví dụ, trong đồ thị sau, bậc vào của đỉnh 2 là 2, và bậc ra của đỉnh 2 là 1.



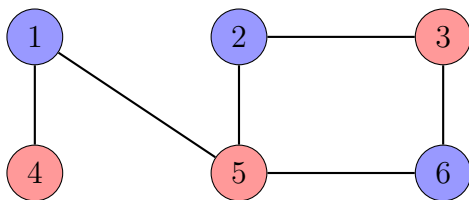
Tô màu

Trong một **cách tô màu** của một đồ thị, mỗi đỉnh được gán một màu sao cho không có hai đỉnh kề nhau nào có cùng màu.

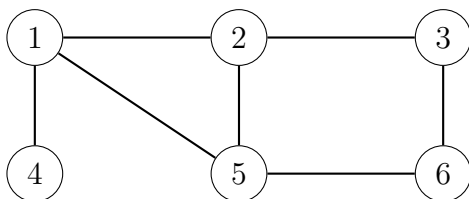
Một đồ thị là **hai phía** (bipartite) nếu có thể tô màu nó bằng hai màu. Hóa ra một đồ thị là hai phía khi và chỉ khi nó không chứa chu trình có số cạnh lẻ. Ví dụ, đồ thị



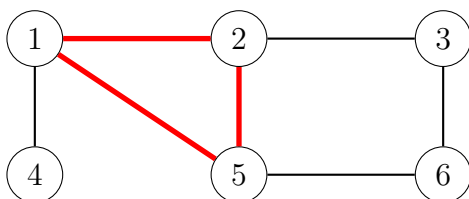
là hai phía, vì nó có thể được tô màu như sau:



Tuy nhiên, đồ thị

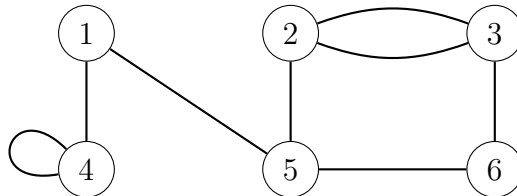


không phải là hai phía, vì không thể tô màu chu trình gồm ba đỉnh sau bằng hai màu:



Tính đơn

Một đồ thị là **đơn** nếu không có cạnh nào bắt đầu và kết thúc tại cùng một đỉnh, và không có nhiều cạnh giữa cùng hai đỉnh. Thông thường ta giả sử rằng các đồ thị là đơn. Ví dụ, đồ thị sau *không* đơn:



11.2 Biểu diễn đồ thị

Có nhiều cách để biểu diễn đồ thị trong thuật toán. Việc chọn cấu trúc dữ liệu phụ thuộc vào kích thước của đồ thị và cách thuật toán xử lý nó. Tiếp theo chúng ta sẽ xét ba cách biểu diễn phổ biến.

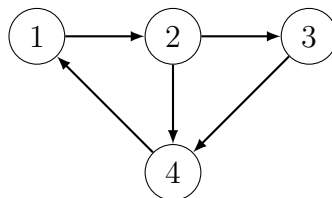
Biểu diễn bằng danh sách kề

Trong biểu diễn bằng danh sách kề, mỗi đỉnh x trong đồ thị được gán một **danh sách kề** gồm các đỉnh mà có cạnh đi từ x đến chúng. Danh sách kề là cách phổ biến nhất để biểu diễn đồ thị, và hầu hết các thuật toán có thể được cài đặt hiệu quả bằng cách dùng chúng.

Một cách thuận tiện để lưu các danh sách kề là khai báo một mảng các vector như sau:

```
vector<int> adj[N];
```

Hằng số N được chọn sao cho có thể lưu được tất cả các danh sách kề. Ví dụ, đồ thị



có thể được lưu như sau:

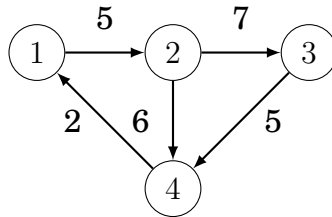
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Nếu đồ thị là vô hướng, nó có thể được lưu theo cách tương tự, nhưng mỗi cạnh được thêm vào theo cả hai hướng.

Với đồ thị có trọng số, cấu trúc có thể được mở rộng như sau:

```
vector<pair<int,int>> adj[N];
```

Trong trường hợp này, danh sách kề của đỉnh a chứa cặp (b, w) mỗi khi có một cạnh từ đỉnh a đến đỉnh b với trọng số w . Ví dụ, đồ thị



có thể được lưu như sau:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Lợi ích của việc dùng danh sách kề là ta có thể tìm hiệu quả các đỉnh mà ta có thể di chuyển tới từ một đỉnh cho trước qua một cạnh. Ví dụ, vòng lặp sau duyệt qua tất cả các đỉnh mà ta có thể di chuyển tới từ đỉnh s :

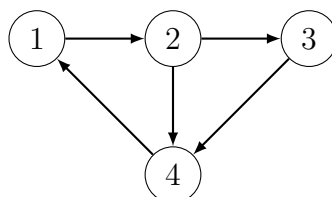
```
for (auto u : adj[s]) {
    // process node u
}
```

Biểu diễn bằng ma trận kề

Một **ma trận kề** là một mảng hai chiều cho biết đồ thị chứa những cạnh nào. Ta có thể kiểm tra hiệu quả từ ma trận kề xem có cạnh giữa hai đỉnh hay không. Ma trận có thể được lưu dưới dạng một mảng

```
int adj[N][N];
```

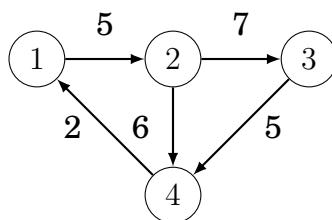
trong đó mỗi giá trị $adj[a][b]$ cho biết đồ thị có chứa một cạnh từ đỉnh a đến đỉnh b hay không. Nếu cạnh đó có trong đồ thị, thì $adj[a][b] = 1$, và ngược lại $adj[a][b] = 0$. Ví dụ, đồ thị



có thể được biểu diễn như sau:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Nếu đồ thị có trọng số, biểu diễn bằng ma trận kề có thể được mở rộng sao cho ma trận chứa trọng số của cạnh nếu cạnh đó tồn tại. Khi dùng cách biểu diễn này, đồ thị



tương ứng với ma trận sau:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Nhược điểm của biểu diễn bằng ma trận kề là ma trận chứa n^2 phần tử, và thường phần lớn trong số đó là 0. Vì lý do này, cách biểu diễn này không thể được dùng nếu đồ thị lớn.

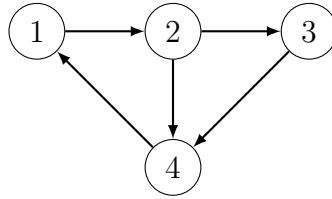
Biểu diễn bằng danh sách cạnh

Một **danh sách cạnh** chứa tất cả các cạnh của một đồ thị theo một thứ tự nào đó. Đây là một cách thuận tiện để biểu diễn đồ thị nếu thuật toán xử lý tất cả các cạnh của đồ thị và không cần tìm các cạnh bắt đầu từ một đỉnh cho trước.

Danh sách cạnh có thể được lưu trong một vector

```
vector<pair<int,int>> edges;
```

trong đó mỗi cặp (a, b) biểu thị rằng có một cạnh từ đỉnh a đến đỉnh b . Do đó, đồ thị



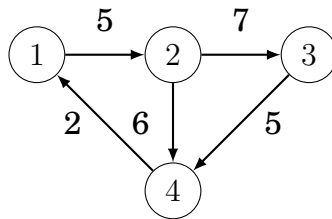
có thể được biểu diễn như sau:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Nếu đồ thị có trọng số, cấu trúc có thể được mở rộng như sau:

```
vector<tuple<int,int,int>> edges;
```

Mỗi phần tử trong danh sách này có dạng (a, b, w) , nghĩa là có một cạnh từ đỉnh a đến đỉnh b với trọng số w . Ví dụ, đồ thị



có thể được biểu diễn như sau¹:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

¹Trong một số trình biên dịch cũ hơn, phải dùng hàm `make_tuple` thay cho cặp ngoặc nhọn (ví dụ, `make_tuple(1,2,5)` thay cho `{1,2,5}`).

Chương 12

Duyệt đồ thị (Graph Traversal)

Chương này bàn về hai thuật toán đồ thị cơ bản: tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng. Cả hai thuật toán đều nhận một đỉnh xuất phát trong đồ thị và thăm tất cả các đỉnh có thể đi tới từ đỉnh đó. Điểm khác nhau giữa hai thuật toán là thứ tự mà chúng thăm các đỉnh.

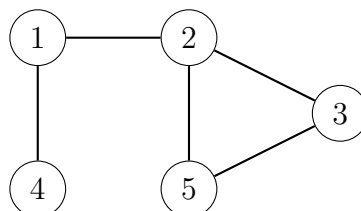
12.1 Tìm kiếm theo chiều sâu

Tìm kiếm theo chiều sâu (Depth-first search, DFS) là một kỹ thuật duyệt đồ thị đơn giản. Thuật toán bắt đầu từ một đỉnh xuất phát, rồi đi tới tất cả các đỉnh khác có thể đạt được từ đỉnh xuất phát bằng các cạnh của đồ thị.

Tìm kiếm theo chiều sâu luôn đi theo một đường đi duy nhất trong đồ thị chừng nào còn tìm thấy đỉnh mới. Sau đó, nó quay lại các đỉnh trước đó và bắt đầu khám phá những phần khác của đồ thị. Thuật toán ghi nhớ các đỉnh đã thăm, nhờ đó mỗi đỉnh chỉ được xử lý một lần.

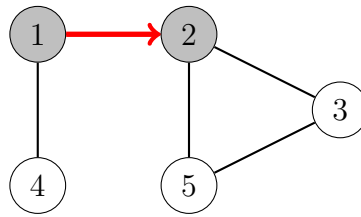
Ví dụ

Hãy xét cách tìm kiếm theo chiều sâu xử lý đồ thị sau:

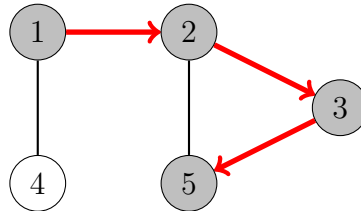


Ta có thể bắt đầu tìm kiếm ở bất kỳ đỉnh nào của đồ thị; ở đây ta bắt đầu từ đỉnh 1.

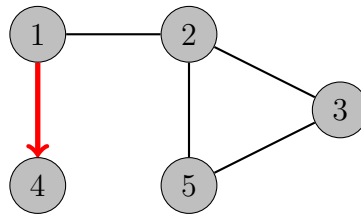
Trước tiên, tìm kiếm đi tới đỉnh 2:



Sau đó, các đỉnh 3 và 5 sẽ được thăm:



Các đỉnh kề của đỉnh 5 là 2 và 3, nhưng tìm kiếm đã thăm cả hai đỉnh này, vì vậy đã đến lúc quay lại các đỉnh trước đó. Các đỉnh kề của 3 và 2 cũng đã được thăm, nên tiếp theo ta đi từ đỉnh 1 tới đỉnh 4:



Sau bước này, tìm kiếm kết thúc vì nó đã thăm tất cả các đỉnh.

Độ phức tạp thời gian của tìm kiếm theo chiều sâu là $O(n + m)$, trong đó n là số đỉnh và m là số cạnh, vì thuật toán xử lý mỗi đỉnh và mỗi cạnh một lần.

Cài đặt

Tìm kiếm theo chiều sâu có thể được cài đặt thuận tiện bằng đệ quy. Hàm dfs sau bắt đầu một lượt tìm kiếm theo chiều sâu từ một đỉnh cho trước. Hàm giả sử đồ thị được lưu dưới dạng danh sách kề trong một mảng

```
vector<int> adj[N];
```

và cũng duy trì một mảng

```
bool visited[N];
```

dùng để ghi nhớ các đỉnh đã thăm. Ban đầu, mỗi giá trị trong mảng là false, và khi tìm kiếm tới đỉnh s , giá trị `visited[s]` trở thành true. Hàm có thể được cài đặt như sau:

```
void dfs(int s) {
    if (visited[s]) return;
```

```

visited[s] = true;
// process node s
for (auto u: adj[s]) {
    dfs(u);
}
}

```

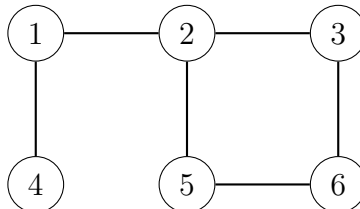
12.2 Tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng (Breadth-first search, BFS) thăm các đỉnh theo thứ tự tăng dần của khoảng cách từ đỉnh xuất phát. Do đó, ta có thể dùng tìm kiếm theo chiều rộng để tính khoảng cách từ đỉnh xuất phát tới tất cả các đỉnh khác. Tuy nhiên, tìm kiếm theo chiều rộng khó cài đặt hơn tìm kiếm theo chiều sâu.

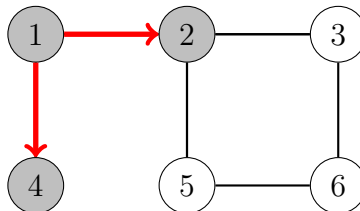
Tìm kiếm theo chiều rộng đi qua các đỉnh từng mức một. Trước tiên, tìm kiếm khám phá các đỉnh có khoảng cách tới đỉnh xuất phát là 1, sau đó là các đỉnh có khoảng cách là 2, v.v. Quá trình này tiếp tục cho tới khi tất cả các đỉnh đã được thăm.

Ví dụ

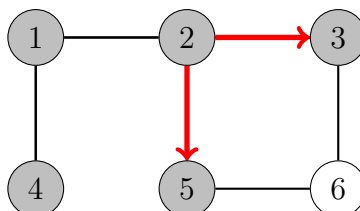
Hãy xét cách tìm kiếm theo chiều rộng xử lý đồ thị sau:



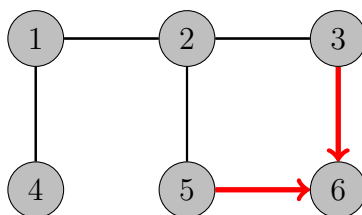
Giả sử tìm kiếm bắt đầu tại đỉnh 1. Trước tiên, ta xử lý tất cả các đỉnh có thể đi tới từ đỉnh 1 bằng đúng một cạnh:



Sau đó, ta đi tiếp tới các đỉnh 3 và 5:



Cuối cùng, ta thăm đỉnh 6:



Bây giờ ta đã tính được khoảng cách từ đỉnh xuất phát tới tất cả các đỉnh của đồ thị. Các khoảng cách như sau:

đỉnh	khoảng cách
1	0
2	1
3	2
4	1
5	2
6	3

Tương tự tìm kiếm theo chiều sâu, độ phức tạp thời gian của tìm kiếm theo chiều rộng là $O(n + m)$, trong đó n là số đỉnh và m là số cạnh.

Cài đặt

Tìm kiếm theo chiều rộng khó cài đặt hơn tìm kiếm theo chiều sâu, vì thuật toán thăm các đỉnh ở nhiều phần khác nhau của đồ thị. Cách cài đặt điển hình dựa trên một hàng đợi chứa các đỉnh. Ở mỗi bước, đỉnh tiếp theo trong hàng đợi sẽ được xử lý.

Đoạn mã sau giả sử đồ thị được lưu dưới dạng danh sách kề và duy trì các cấu trúc dữ liệu sau:

```
queue<int> q;  
bool visited [N];  
int distance[N];
```

Hàng đợi q chứa các đỉnh cần xử lý theo thứ tự tăng dần của khoảng cách. Các đỉnh mới luôn được thêm vào cuối hàng đợi, và đỉnh ở đầu hàng đợi là đỉnh tiếp theo được xử lý. Mảng `visited` cho biết những đỉnh nào đã được tìm kiếm thăm, còn mảng `distance` sẽ chứa khoảng cách từ đỉnh xuất phát tới tất cả các đỉnh của đồ thị.

Tìm kiếm có thể được cài đặt như sau, bắt đầu tại đỉnh x :

```
visited[x] = true;  
distance[x] = 0;  
q.push(x);  
while (!q.empty()) {  
    int s = q.front(); q.pop();
```



```

// process node s
for (auto u : adj[s]) {
    if (visited[u]) continue;
    visited[u] = true;
    distance[u] = distance[s]+1;
    q.push(u);
}
}

```

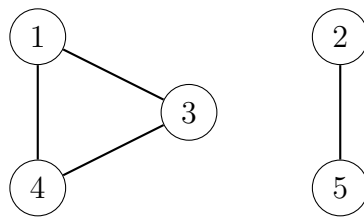
12.3 Ứng dụng

Bằng các thuật toán duyệt đồ thị, ta có thể kiểm tra nhiều tính chất của đồ thị. Thông thường, cả tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng đều có thể dùng được, nhưng trong thực tế, tìm kiếm theo chiều sâu thường là lựa chọn tốt hơn vì dễ cài đặt hơn. Trong các ứng dụng sau, ta giả sử đồ thị là vô hướng.

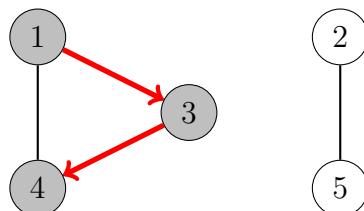
Kiểm tra tính liên thông

Một đồ thị là liên thông nếu tồn tại một đường đi giữa mọi cặp đỉnh của đồ thị. Vì vậy, ta có thể kiểm tra đồ thị có liên thông hay không bằng cách bắt đầu từ một đỉnh bất kỳ và xem ta có thể đi tới tất cả các đỉnh còn lại hay không.

Ví dụ, trong đồ thị



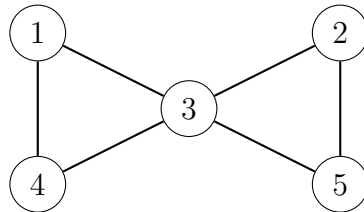
một lượt tìm kiếm theo chiều sâu từ đỉnh 1 thăm các đỉnh sau:



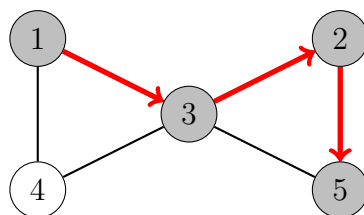
Vì tìm kiếm không thăm tất cả các đỉnh, ta có thể kết luận rằng đồ thị không liên thông. Tương tự, ta cũng có thể tìm tất cả các thành phần liên thông của một đồ thị bằng cách duyệt qua các đỉnh và luôn bắt đầu một lượt tìm kiếm theo chiều sâu mới nếu đỉnh hiện tại chưa thuộc thành phần nào.

Tìm chu trình

Một đồ thị chứa chu trình nếu trong khi duyệt đồ thị, ta tìm thấy một đỉnh có đỉnh kề (khác với đỉnh ngay trước đó trên đường đi hiện tại) đã được thăm. Ví dụ, đồ thị



chứa hai chu trình và ta có thể tìm một trong số đó như sau:



Sau khi đi từ đỉnh 2 tới đỉnh 5, ta nhận thấy đỉnh kề 3 của đỉnh 5 đã được thăm. Vì vậy, đồ thị chứa một chu trình đi qua đỉnh 3, chẳng hạn $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

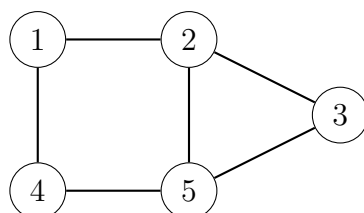
Một cách khác để kiểm tra đồ thị có chứa chu trình hay không là tính số đỉnh và số cạnh trong từng thành phần. Nếu một thành phần có c đỉnh và không có chu trình, nó phải có đúng $c - 1$ cạnh (tức là nó là một cây). Nếu có c cạnh trở lên, thành phần đó chắc chắn chứa một chu trình.

Kiểm tra tính hai phía

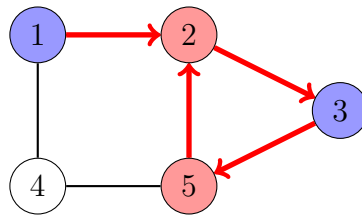
Một đồ thị là hai phía (bipartite) nếu các đỉnh của nó có thể được tô bằng hai màu sao cho không có hai đỉnh kề nhau cùng màu. Việc kiểm tra một đồ thị có hai phía hay không bằng các thuật toán duyệt đồ thị đơn giản một cách bất ngờ.

Ý tưởng là tô đỉnh xuất phát màu xanh, tất cả các đỉnh kề của nó màu đỏ, tất cả các đỉnh kề của chúng màu xanh, v.v. Nếu tại một thời điểm nào đó của quá trình tìm kiếm ta thấy hai đỉnh kề nhau có cùng màu, điều này nghĩa là đồ thị không hai phía. Ngược lại, đồ thị là hai phía và ta đã tìm được một cách tô màu.

Ví dụ, đồ thị



không hai phía, vì một lượt tìm kiếm từ đỉnh 1 diễn ra như sau:



Ta nhận thấy cả hai đỉnh 2 và 5 đều có màu đỏ, trong khi chúng là hai đỉnh kề nhau trong đồ thị. Vì vậy, đồ thị không hai phía.

Thuật toán này luôn đúng, vì khi chỉ có hai màu, màu của đỉnh xuất phát trong một thành phần quyết định màu của tất cả các đỉnh khác trong thành phần đó. Việc đỉnh xuất phát là đỏ hay xanh không tạo ra khác biệt.

Lưu ý rằng trong trường hợp tổng quát, việc xác định các đỉnh của một đồ thị có thể được tô bằng k màu sao cho không có hai đỉnh kề nhau cùng màu là khó. Ngay cả khi $k = 3$, chưa có thuật toán hiệu quả nào được biết, và bài toán là NP-khó.

Chương 13

Đường đi ngắn nhất

Tìm đường đi ngắn nhất giữa hai đỉnh của một đồ thị là một bài toán quan trọng với nhiều ứng dụng thực tế. Ví dụ, một bài toán tự nhiên liên quan đến mạng lưới đường bộ là tính độ dài nhỏ nhất có thể của một tuyến đường giữa hai thành phố, khi biết độ dài của các con đường.

Trong đồ thị không trọng số, độ dài của một đường đi bằng số cạnh của nó, và ta có thể chỉ cần dùng tìm kiếm theo chiều rộng để tìm đường đi ngắn nhất. Tuy nhiên, trong chương này ta tập trung vào đồ thị có trọng số, nơi cần các thuật toán tinh vi hơn để tìm đường đi ngắn nhất.

13.1 Thuật toán Bellman–Ford

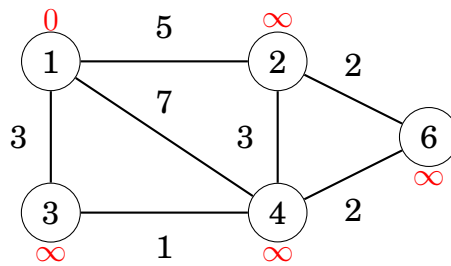
Thuật toán Bellman–Ford¹ tìm các đường đi ngắn nhất từ một đỉnh xuất phát tới tất cả các đỉnh của đồ thị. Thuật toán có thể xử lý mọi loại đồ thị, miễn là đồ thị không chứa chu trình có độ dài âm. Nếu đồ thị chứa chu trình âm, thuật toán có thể phát hiện điều đó.

Thuật toán theo dõi khoảng cách từ đỉnh xuất phát tới tất cả các đỉnh của đồ thị. Ban đầu, khoảng cách tới đỉnh xuất phát là 0 và khoảng cách tới mọi đỉnh khác là vô cực. Thuật toán giảm dần các khoảng cách bằng cách tìm những cạnh làm ngắn đường đi, cho tới khi không thể giảm thêm khoảng cách nào.

Ví dụ

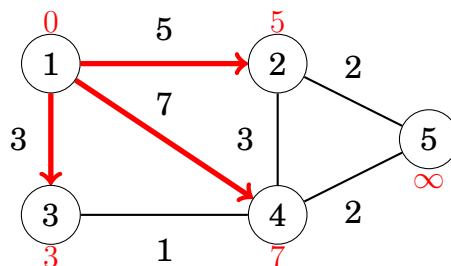
Hãy xét cách thuật toán Bellman–Ford hoạt động trên đồ thị sau:

¹Thuật toán được đặt theo tên R. E. Bellman và L. R. Ford, những người công bố nó độc lập lần lượt vào các năm 1958 và 1956 [5, 24].

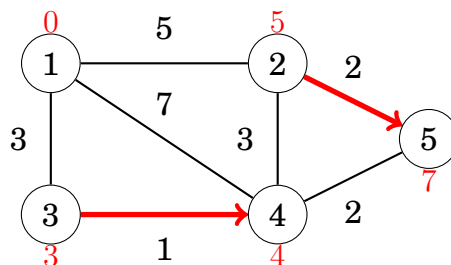


Mỗi đỉnh của đồ thị được gán một khoảng cách. Ban đầu, khoảng cách tới đỉnh xuất phát là 0, và khoảng cách tới mọi đỉnh khác là vô cực.

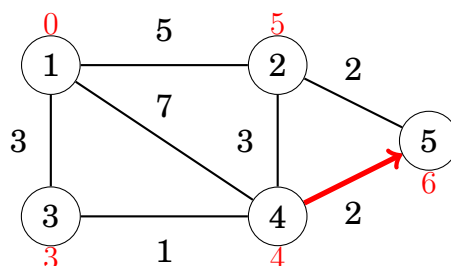
Thuật toán tìm các cạnh làm giảm khoảng cách. Đầu tiên, tất cả các cạnh xuất phát từ đỉnh 1 đều làm giảm khoảng cách:



Sau đó, các cạnh $2 \rightarrow 5$ và $3 \rightarrow 4$ làm giảm khoảng cách:

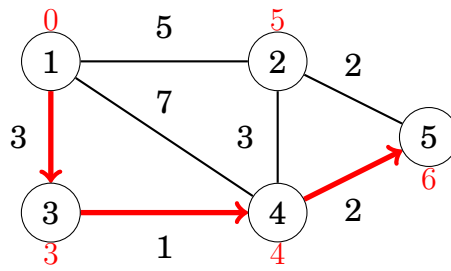


Cuối cùng, còn một thay đổi nữa:



Sau bước này, không cạnh nào có thể làm giảm khoảng cách. Điều đó nghĩa là các khoảng cách đã là cuối cùng, và ta đã tính thành công các khoảng cách ngắn nhất từ đỉnh xuất phát tới tất cả các đỉnh của đồ thị.

Ví dụ, khoảng cách ngắn nhất 3 từ đỉnh 1 tới đỉnh 5 tương ứng với đường đi sau:



Cài đặt

Cài đặt sau của thuật toán Bellman–Ford xác định các khoảng cách ngắn nhất từ một đỉnh x tới tất cả các đỉnh của đồ thị. Đoạn mã giả sử đồ thị được lưu dưới dạng danh sách cạnh edges, gồm các bộ có dạng (a, b, w) , nghĩa là có một cạnh từ đỉnh a tới đỉnh b với trọng số w .

Thuật toán gồm $n - 1$ vòng, và ở mỗi vòng thuật toán duyệt qua tất cả các cạnh của đồ thị rồi cố gắng giảm các khoảng cách. Thuật toán xây dựng mảng distance chứa khoảng cách từ x tới tất cả các đỉnh của đồ thị. Hằng INF biểu diễn khoảng cách vô cực.

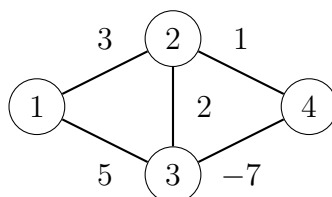
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Độ phức tạp thời gian của thuật toán là $O(nm)$, vì thuật toán gồm $n - 1$ vòng và duyệt qua tất cả m cạnh trong mỗi vòng. Nếu đồ thị không có chu trình âm, tất cả khoảng cách sẽ là cuối cùng sau $n - 1$ vòng, vì mỗi đường đi ngắn nhất có thể chứa nhiều nhất $n - 1$ cạnh.

Trong thực tế, các khoảng cách cuối cùng thường được tìm thấy nhanh hơn $n - 1$ vòng. Do đó, một cách để làm thuật toán hiệu quả hơn là dừng thuật toán nếu trong một vòng không có khoảng cách nào có thể được giảm.

Chu trình âm

Thuật toán Bellman–Ford cũng có thể được dùng để kiểm tra đồ thị có chứa chu trình độ dài âm hay không. Ví dụ, đồ thị



chứa một chu trình âm $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ có độ dài -4 .

Nếu đồ thị chứa chu trình âm, ta có thể làm ngắn vô hạn lần bất kỳ đường đi nào chứa chu trình đó bằng cách lặp lại chu trình nhiều lần. Vì vậy, khái niệm đường đi ngắn nhất không còn có ý nghĩa trong tình huống này.

Có thể phát hiện chu trình âm bằng thuật toán Bellman–Ford bằng cách chạy thuật toán trong n vòng. Nếu vòng cuối cùng làm giảm bất kỳ khoảng cách nào, đồ thị chứa một chu trình âm. Lưu ý rằng thuật toán này có thể được dùng để tìm chu trình âm trong toàn bộ đồ thị, bất kể đỉnh xuất phát là gì.

Thuật toán SPFA

Thuật toán SPFA ("Shortest Path Faster Algorithm") [20] là một biến thể của thuật toán Bellman–Ford, thường hiệu quả hơn thuật toán gốc. SPFA không duyệt qua tất cả các cạnh ở mỗi vòng, mà chọn các cạnh cần xét một cách thông minh hơn.

Thuật toán duy trì một hàng đợi các đỉnh có thể được dùng để giảm khoảng cách. Đầu tiên, thuật toán thêm đỉnh xuất phát x vào hàng đợi. Sau đó, thuật toán luôn xử lý đỉnh đầu tiên trong hàng đợi, và khi một cạnh $a \rightarrow b$ làm giảm một khoảng cách, đỉnh b được thêm vào hàng đợi.

Hiệu quả của thuật toán SPFA phụ thuộc vào cấu trúc của đồ thị: thuật toán thường chạy nhanh, nhưng độ phức tạp thời gian trong trường hợp xấu nhất vẫn là $O(nm)$, và có thể tạo các bộ dữ liệu khiến thuật toán chậm như Bellman–Ford gốc.

13.2 Thuật toán Dijkstra

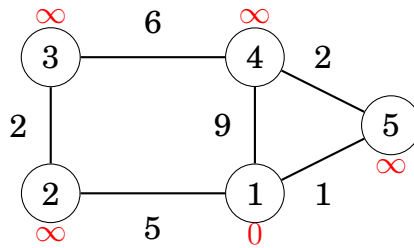
Thuật toán Dijkstra² tìm các đường đi ngắn nhất từ đỉnh xuất phát tới tất cả các đỉnh của đồ thị, tương tự thuật toán Bellman–Ford. Ưu điểm của thuật toán Dijkstra là nó hiệu quả hơn và có thể dùng để xử lý các đồ thị lớn. Tuy nhiên, thuật toán yêu cầu đồ thị không có cạnh trọng số âm.

Giống thuật toán Bellman–Ford, thuật toán Dijkstra duy trì khoảng cách tới các đỉnh và giảm chúng trong quá trình tìm kiếm. Thuật toán Dijkstra hiệu quả vì nó chỉ xử lý mỗi cạnh trong đồ thị một lần, dựa trên việc không có cạnh âm.

Ví dụ

Hãy xét cách thuật toán Dijkstra hoạt động trên đồ thị sau khi đỉnh xuất phát là đỉnh 1:

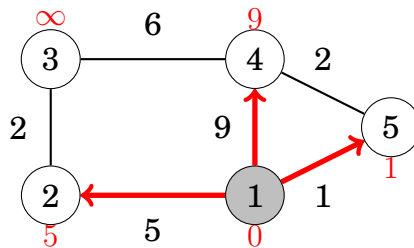
²E. W. Dijkstra công bố thuật toán này vào năm 1959 [14]; tuy nhiên, bài báo gốc của ông không nói cách cài đặt thuật toán hiệu quả.



Giống thuật toán Bellman–Ford, ban đầu khoảng cách tới đỉnh xuất phát là 0 và khoảng cách tới mọi đỉnh khác là vô cực.

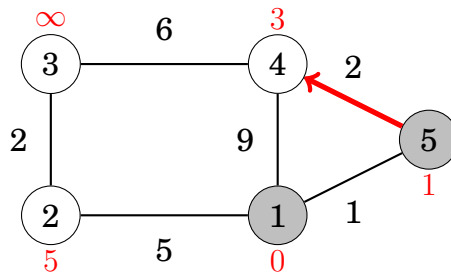
Ở mỗi bước, thuật toán Dijkstra chọn một đỉnh chưa được xử lý và có khoảng cách nhỏ nhất có thể. Đỉnh đầu tiên như vậy là đỉnh 1 với khoảng cách 0.

Khi một đỉnh được chọn, thuật toán duyệt qua tất cả các cạnh xuất phát từ đỉnh đó và dùng chúng để giảm các khoảng cách:

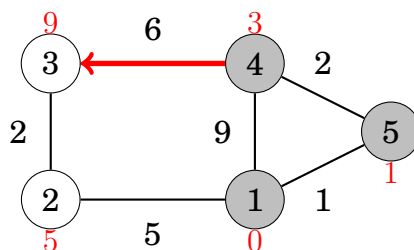


Trong trường hợp này, các cạnh từ đỉnh 1 làm giảm khoảng cách của các đỉnh 2, 4 và 5; khoảng cách của chúng hiện là 5, 9 và 1.

Đỉnh tiếp theo được xử lý là đỉnh 5 với khoảng cách 1. Điều này làm giảm khoảng cách tới đỉnh 4 từ 9 xuống 3:



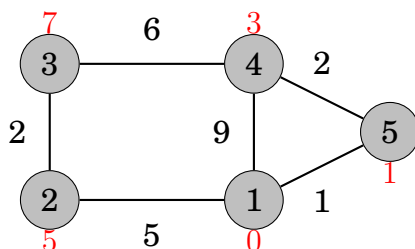
Sau đó, đỉnh tiếp theo là đỉnh 4, làm giảm khoảng cách tới đỉnh 3 xuống 9:



Một tính chất đáng chú ý của thuật toán Dijkstra là mỗi khi một đỉnh được chọn, khoảng cách của nó đã là cuối cùng. Ví dụ, tại thời điểm này của

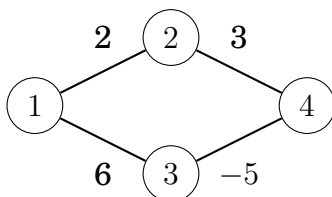
thuật toán, các khoảng cách 0, 1 và 3 là khoảng cách cuối cùng tới các đỉnh 1, 5 và 4.

Sau đó, thuật toán xử lý hai đỉnh còn lại, và các khoảng cách cuối cùng như sau:



Cạnh âm

Tính hiệu quả của thuật toán Dijkstra dựa trên giả thiết rằng đồ thị không chứa cạnh âm. Nếu có cạnh âm, thuật toán có thể cho kết quả sai. Ví dụ, hãy xét đồ thị sau:



Đường đi ngắn nhất từ đỉnh 1 tới đỉnh 4 là $1 \rightarrow 3 \rightarrow 4$ và có độ dài 1. Tuy nhiên, thuật toán Dijkstra tìm đường đi $1 \rightarrow 2 \rightarrow 4$ bằng cách đi theo các cạnh có trọng số nhỏ nhất. Thuật toán không xét đến việc trên đường đi kia, trọng số -5 bù lại trọng số lớn 6 trước đó.

Cài đặt

Cài đặt sau của thuật toán Dijkstra tính các khoảng cách nhỏ nhất từ một đỉnh x tới các đỉnh khác của đồ thị. Đồ thị được lưu dưới dạng danh sách kề, sao cho $\text{adj}[a]$ chứa một cặp (b, w) mỗi khi có một cạnh từ đỉnh a tới đỉnh b với trọng số w .

Một cài đặt hiệu quả của thuật toán Dijkstra cần có khả năng tìm nhanh đỉnh chưa xử lý có khoảng cách nhỏ nhất. Cấu trúc dữ liệu phù hợp cho việc này là hàng đợi ưu tiên chứa các đỉnh được sắp theo khoảng cách của chúng. Dùng hàng đợi ưu tiên, ta có thể lấy đỉnh tiếp theo cần xử lý trong thời gian logarit.

Trong đoạn mã sau, hàng đợi ưu tiên q chứa các cặp dạng $(-d, x)$, nghĩa là khoảng cách hiện tại tới đỉnh x là d . Mảng `distance` chứa khoảng cách tới từng đỉnh, và mảng `processed` cho biết một đỉnh đã được xử lý hay chưa. Ban đầu, khoảng cách tới x là 0 và tới mọi đỉnh khác là ∞ .

```

for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}

```

Lưu ý rằng hàng đợi ưu tiên chứa các khoảng cách *âm* tới các đỉnh. Lý do là phiên bản mặc định của hàng đợi ưu tiên trong C++ tìm phần tử lớn nhất, trong khi ta muốn tìm phần tử nhỏ nhất. Bằng cách dùng khoảng cách âm, ta có thể dùng trực tiếp hàng đợi ưu tiên mặc định³. Cũng lưu ý rằng có thể có nhiều bản sao của cùng một đỉnh trong hàng đợi ưu tiên; tuy nhiên, chỉ bản sao có khoảng cách nhỏ nhất mới được xử lý.

Độ phức tạp thời gian của cài đặt trên là $O(n + m \log m)$, vì thuật toán duyệt qua tất cả các đỉnh của đồ thị và với mỗi cạnh, thêm nhiều nhất một khoảng cách vào hàng đợi ưu tiên.

13.3 Thuật toán Floyd–Warshall

Thuật toán Floyd–Warshall⁴ cung cấp một cách tiếp cận khác cho bài toán tìm đường đi ngắn nhất. Khác với các thuật toán khác trong chương này, nó tìm mọi đường đi ngắn nhất giữa các đỉnh chỉ trong một lần chạy.

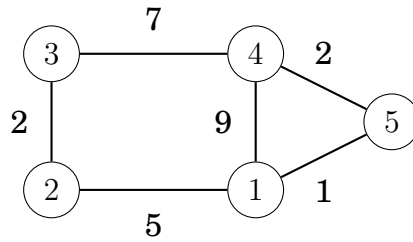
Thuật toán duy trì một mảng hai chiều chứa khoảng cách giữa các đỉnh. Đầu tiên, các khoảng cách chỉ được tính bằng các cạnh trực tiếp giữa các đỉnh, sau đó thuật toán giảm khoảng cách bằng cách dùng các đỉnh trung gian trên đường đi.

Ví dụ

Hãy xét cách thuật toán Floyd–Warshall hoạt động trên đồ thị sau:

³Tất nhiên, ta cũng có thể khai báo hàng đợi ưu tiên như trong Chương 4.5 và dùng khoảng cách dương, nhưng cài đặt sẽ dài hơn một chút.

⁴Thuật toán được đặt theo tên R. W. Floyd và S. Warshall, những người công bố nó độc lập vào năm 1962 [23, 70].



Ban đầu, khoảng cách từ mỗi đỉnh tới chính nó là 0, và khoảng cách giữa hai đỉnh a và b là ∞ nếu có một cạnh giữa a và b với trọng số x . Mọi khoảng cách khác là vô cực.

Trong đồ thị này, mảng ban đầu như sau:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Thuật toán gồm các vòng liên tiếp. Ở mỗi vòng, thuật toán chọn một đỉnh mới có thể đóng vai trò đỉnh trung gian trên các đường đi từ lúc này, và các khoảng cách được giảm bằng đỉnh đó.

Ở vòng đầu tiên, đỉnh 1 là đỉnh trung gian mới. Có một đường đi mới giữa các đỉnh 2 và 4 có độ dài 14, vì đỉnh 1 nối chúng lại. Cũng có một đường đi mới giữa các đỉnh 2 và 5 có độ dài 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

Ở vòng thứ hai, đỉnh 2 là đỉnh trung gian mới. Điều này tạo ra các đường đi mới giữa các đỉnh 1 và 3 và giữa các đỉnh 3 và 5:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

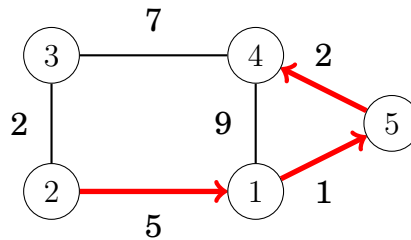
Ở vòng thứ ba, đỉnh 3 là đỉnh trung gian mới. Có một đường đi mới giữa các đỉnh 2 và 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

Thuật toán tiếp tục như vậy cho tới khi tất cả các đỉnh đã được chọn làm đỉnh trung gian. Sau khi thuật toán kết thúc, mảng chứa khoảng cách nhỏ nhất giữa mọi cặp đỉnh:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

Ví dụ, mảng cho biết rằng khoảng cách ngắn nhất giữa các đỉnh 2 và 4 là 8. Điều này tương ứng với đường đi sau:



Cài đặt

Ưu điểm của thuật toán Floyd–Warshall là nó dễ cài đặt. Đoạn mã sau xây dựng một ma trận khoảng cách, trong đó $\text{distance}[a][b]$ là khoảng cách ngắn nhất giữa hai đỉnh a và b . Đầu tiên, thuật toán khởi tạo distance bằng ma trận kề adj của đồ thị:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}
```

Sau đó, các khoảng cách ngắn nhất có thể được tìm như sau:

```
for (int k = 1; k <= n; k++) {
```

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        distance[i][j] = min(distance[i][j],
                               distance[i][k]+distance[k][j]);
    }
}

```

Độ phức tạp thời gian của thuật toán là $O(n^3)$, vì nó chứa ba vòng lặp lồng nhau duyệt qua các đỉnh của đồ thị.

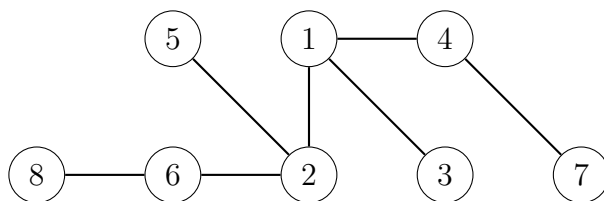
Vì cài đặt thuật toán Floyd–Warshall đơn giản, thuật toán có thể là một lựa chọn tốt ngay cả khi ta chỉ cần tìm một đường đi ngắn nhất duy nhất trong đồ thị. Tuy nhiên, thuật toán chỉ có thể dùng khi đồ thị đủ nhỏ để độ phức tạp bậc ba chạy đủ nhanh.

Chương 14

Thuật toán trên cây

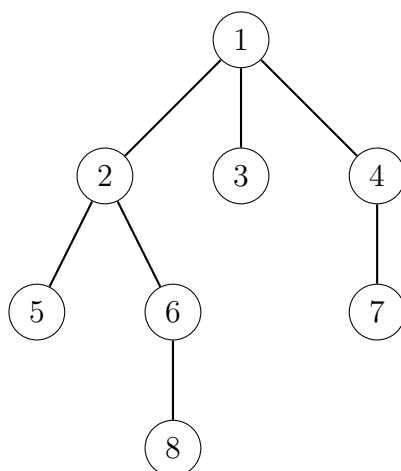
Một **cây** là một đồ thị liên thông, không chu trình gồm n đỉnh và $n - 1$ cạnh. Xóa bất kỳ cạnh nào khỏi một cây sẽ chia nó thành hai thành phần, và thêm bất kỳ cạnh nào vào một cây sẽ tạo ra một chu trình. Hơn nữa, luôn có đúng một đường đi giữa hai đỉnh bất kỳ của một cây.

Ví dụ, cây sau gồm 8 đỉnh và 7 cạnh:



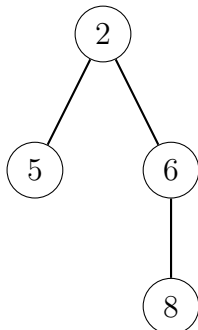
Các **lá** của một cây là những đỉnh có bậc 1, tức là chỉ có một đỉnh kề. Ví dụ, các lá của cây trên là các đỉnh 3, 5, 7 và 8.

Trong một cây **có gốc**, một trong các đỉnh được chọn làm **gốc** của cây, và tất cả các đỉnh còn lại được đặt bên dưới gốc. Ví dụ, trong cây sau, đỉnh 1 là đỉnh gốc.



Trong cây có gốc, các **con** của một đỉnh là những đỉnh kề nằm bên dưới nó, còn **cha** của một đỉnh là đỉnh kề nằm bên trên nó. Mỗi đỉnh có đúng một cha, ngoại trừ gốc không có cha. Ví dụ, trong cây trên, các con của đỉnh 2 là các đỉnh 5 và 6, và cha của nó là đỉnh 1.

Cấu trúc của cây có gốc mang tính *đệ quy*: mỗi đỉnh của cây đóng vai trò gốc của một **cây con** chứa chính đỉnh đó và tất cả các đỉnh nằm trong các cây con của các con của nó. Ví dụ, trong cây trên, cây con của đỉnh 2 gồm các đỉnh 2, 5, 6 và 8:



14.1 Duyệt cây

Các thuật toán duyệt đồ thị tổng quát có thể được dùng để duyệt các đỉnh của một cây. Tuy nhiên, việc duyệt cây dễ cài đặt hơn duyệt đồ thị tổng quát, vì trong cây không có chu trình và không thể đi tới một đỉnh từ nhiều hướng khác nhau.

Cách thông thường để duyệt một cây là bắt đầu một lượt tìm kiếm theo chiều sâu từ một đỉnh tùy ý. Ta có thể dùng hàm đệ quy sau:

```
void dfs(int s, int e) {  
    // process node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Hàm nhận hai tham số: đỉnh hiện tại s và đỉnh trước đó e . Mục đích của tham số e là bảo đảm tìm kiếm chỉ đi tới những đỉnh chưa được thăm.

Lời gọi hàm sau bắt đầu tìm kiếm tại đỉnh x :

```
dfs(x, 0);
```

Trong lời gọi đầu tiên $e = 0$, vì không có đỉnh trước đó, và ta được phép đi theo bất kỳ hướng nào trong cây.

Quy hoạch động

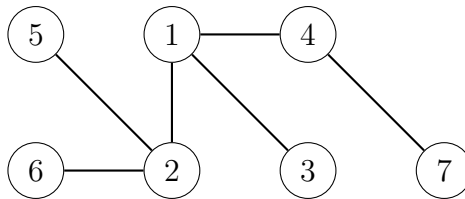
Quy hoạch động có thể được dùng để tính một số thông tin trong khi duyệt cây. Bằng quy hoạch động, chẳng hạn ta có thể tính trong thời gian $O(n)$ cho mỗi đỉnh của một cây có gốc số đỉnh trong cây con của nó, hoặc độ dài đường đi dài nhất từ đỉnh đó tới một lá.

Ví dụ, hãy tính cho mỗi đỉnh s một giá trị $\text{count}[s]$: số đỉnh trong cây con của nó. Cây con chứa chính đỉnh đó và tất cả các đỉnh trong các cây con của các con của nó, nên ta có thể tính số đỉnh một cách đệ quy bằng đoạn mã sau:

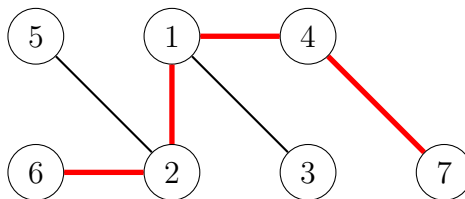
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Đường kính

Đường kính của một cây là độ dài lớn nhất của một đường đi giữa hai đỉnh. Ví dụ, hãy xét cây sau:



Đường kính của cây này là 4, tương ứng với đường đi sau:



Lưu ý rằng có thể có nhiều đường đi có độ dài lớn nhất. Trong đường đi trên, ta có thể thay đỉnh 6 bằng đỉnh 5 để thu được một đường đi khác có độ dài 4.

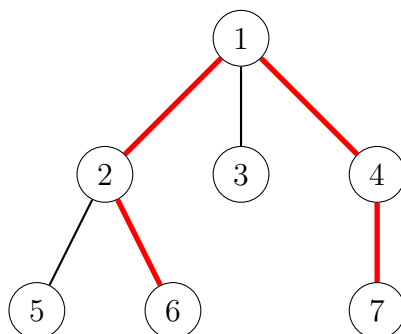
Tiếp theo, ta sẽ bàn về hai thuật toán thời gian $O(n)$ để tính đường kính của một cây. Thuật toán thứ nhất dựa trên quy hoạch động, và thuật toán thứ hai dùng hai lượt tìm kiếm theo chiều sâu.

Thuật toán 1

Một cách tiếp cận tổng quát cho nhiều bài toán trên cây là trước hết chọn gốc cho cây một cách tùy ý. Sau đó, ta có thể cố gắng giải bài toán riêng cho từng cây con. Thuật toán đầu tiên của ta để tính đường kính dựa trên ý tưởng này.

Một nhận xét quan trọng là mọi đường đi trong cây có gốc đều có một *điểm cao nhất*: đỉnh cao nhất thuộc đường đi đó. Vì vậy, ta có thể tính cho mỗi đỉnh độ dài của đường đi dài nhất có điểm cao nhất là đỉnh đó. Một trong các đường đi này tương ứng với đường kính của cây.

Ví dụ, trong cây sau, đỉnh 1 là điểm cao nhất trên đường đi tương ứng với đường kính:



Ta tính cho mỗi đỉnh x hai giá trị:

- $\text{toLeaf}(x)$: độ dài lớn nhất của một đường đi từ x tới một lá bất kỳ
- $\text{maxLength}(x)$: độ dài lớn nhất của một đường đi có điểm cao nhất là x

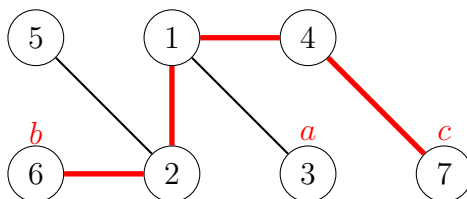
Ví dụ, trong cây trên, $\text{toLeaf}(1) = 2$, vì có đường đi $1 \rightarrow 2 \rightarrow 6$, và $\text{maxLength}(1) = 4$, vì có đường đi $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$. Trong trường hợp này, $\text{maxLength}(1)$ bằng đường kính.

Có thể dùng quy hoạch động để tính các giá trị trên cho mọi đỉnh trong thời gian $O(n)$. Đầu tiên, để tính $\text{toLeaf}(x)$, ta duyệt qua các con của x , chọn một con c có $\text{toLeaf}(c)$ lớn nhất và cộng thêm một vào giá trị này. Sau đó, để tính $\text{maxLength}(x)$, ta chọn hai con phân biệt a và b sao cho tổng $\text{toLeaf}(a) + \text{toLeaf}(b)$ lớn nhất rồi cộng thêm hai vào tổng đó.

Thuật toán 2

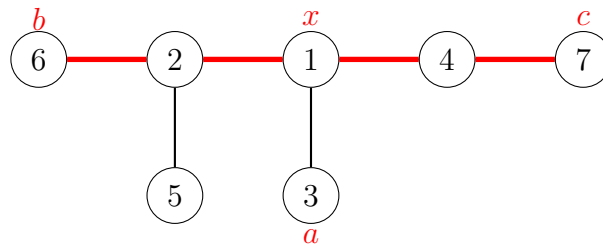
Một cách hiệu quả khác để tính đường kính của cây dựa trên hai lượt tìm kiếm theo chiều sâu. Đầu tiên, ta chọn một đỉnh tùy ý a trong cây và tìm đỉnh b xa nhất từ a . Sau đó, ta tìm đỉnh c xa nhất từ b . Đường kính của cây là khoảng cách giữa b và c .

Trong đồ thị sau, a , b và c có thể là:



Đây là một phương pháp tinh gọn, nhưng vì sao nó đúng?

Sẽ hữu ích nếu vẽ cây theo cách khác, sao cho đường đi tương ứng với đường kính nằm ngang, và mọi đỉnh khác treo từ đường đi đó:

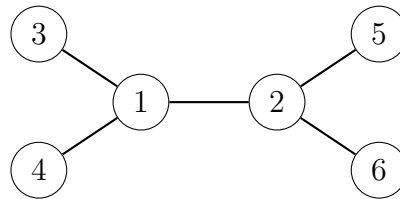


Đỉnh x chỉ nơi đường đi từ đỉnh a nhập vào đường đi tương ứng với đường kính. Đỉnh xa nhất từ a là đỉnh b , đỉnh c , hoặc một đỉnh khác cách đỉnh x ít nhất xa như vậy. Do đó, đỉnh này luôn là một lựa chọn hợp lệ cho một đầu mút của đường đi tương ứng với đường kính.

14.3 Mọi đường đi dài nhất

Bài toán tiếp theo là tính cho mỗi đỉnh trong cây độ dài lớn nhất của một đường đi bắt đầu tại đỉnh đó. Có thể xem đây là một tổng quát hóa của bài toán đường kính cây, vì giá trị lớn nhất trong các độ dài này bằng đường kính của cây. Bài toán này cũng có thể được giải trong thời gian $O(n)$.

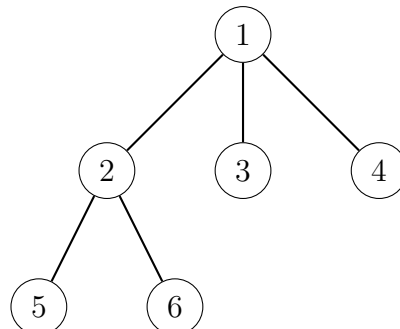
Ví dụ, hãy xét cây sau:



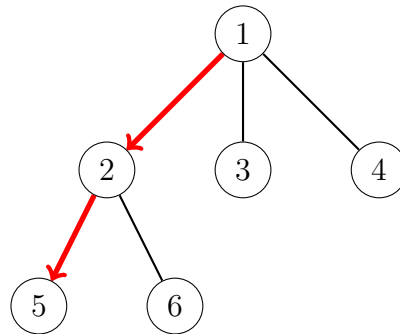
Gọi $\text{maxLength}(x)$ là độ dài lớn nhất của một đường đi bắt đầu tại đỉnh x . Ví dụ, trong cây trên, $\text{maxLength}(4) = 3$, vì có đường đi $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Bảng đầy đủ các giá trị là:

đỉnh x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Trong bài toán này, một điểm khởi đầu tốt cũng là chọn gốc cho cây một cách tùy ý:

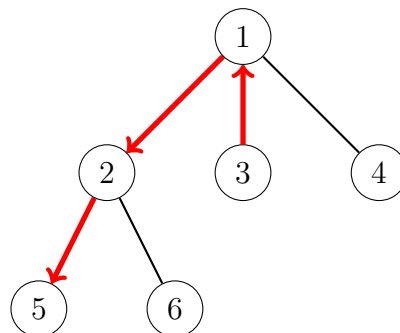


Phần đầu tiên của bài toán là tính cho mỗi đỉnh x độ dài lớn nhất của một đường đi đi qua một con của x . Ví dụ, đường đi dài nhất từ đỉnh 1 đi qua con 2 của nó:

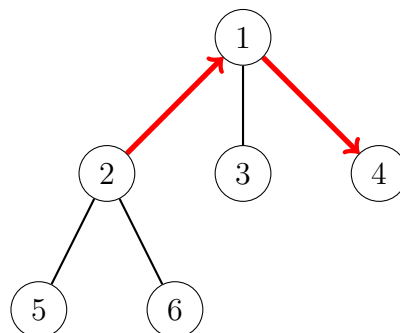


Phần này dễ giải trong thời gian $O(n)$, vì ta có thể dùng quy hoạch động như đã làm trước đó.

Sau đó, phần thứ hai của bài toán là tính cho mỗi đỉnh x độ dài lớn nhất của một đường đi đi qua cha p của nó. Ví dụ, đường đi dài nhất từ đỉnh 3 đi qua cha 1 của nó:



Thoạt nhìn, có vẻ ta nên chọn đường đi dài nhất từ p . Tuy nhiên, cách này *không* phải lúc nào cũng đúng, vì đường đi dài nhất từ p có thể đi qua x . Đây là một ví dụ cho tình huống đó:



Tuy vậy, ta vẫn có thể giải phần thứ hai trong thời gian $O(n)$ bằng cách lưu *hai* độ dài lớn nhất cho mỗi đỉnh x :

- $\text{maxLength}_1(x)$: độ dài lớn nhất của một đường đi từ x
- $\text{maxLength}_2(x)$ độ dài lớn nhất của một đường đi từ x theo hướng khác với đường đi thứ nhất

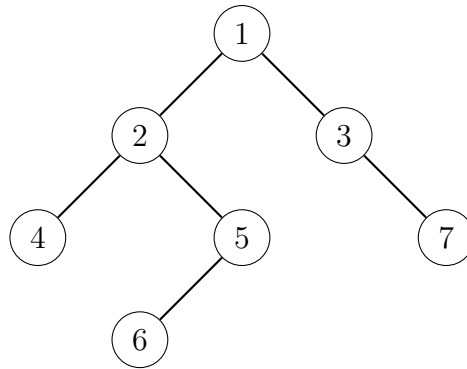
Ví dụ, trong đồ thị trên, $\text{maxLength}_1(1) = 2$ khi dùng đường đi $1 \rightarrow 2 \rightarrow 5$, và $\text{maxLength}_2(1) = 1$ khi dùng đường đi $1 \rightarrow 3$.

Cuối cùng, nếu đường đi tương ứng với $\text{maxLength}_1(p)$ đi qua x , ta kết luận rằng độ dài lớn nhất là $\text{maxLength}_2(p) + 1$, còn nếu không thì độ dài lớn nhất là $\text{maxLength}_1(p) + 1$.

14.4 Cây nhị phân

Một **cây nhị phân** (binary tree) là một cây có gốc trong đó mỗi đỉnh có một cây con trái và một cây con phải. Cây con của một đỉnh có thể rỗng. Vì vậy, mỗi đỉnh trong cây nhị phân có không, một hoặc hai con.

Ví dụ, cây sau là một cây nhị phân:



Các đỉnh của một cây nhị phân có ba thứ tự tự nhiên tương ứng với các cách khác nhau để duyệt cây một cách đệ quy:

- **pre-order**: xử lý gốc trước, rồi duyệt cây con trái, rồi duyệt cây con phải
- **in-order**: duyệt cây con trái trước, rồi xử lý gốc, rồi duyệt cây con phải
- **post-order**: duyệt cây con trái trước, rồi duyệt cây con phải, rồi xử lý gốc

Với cây trên, các đỉnh theo thứ tự pre-order là $[1, 2, 4, 5, 6, 3, 7]$, theo in-order là $[4, 2, 6, 5, 1, 3, 7]$ và theo post-order là $[4, 6, 5, 2, 7, 3, 1]$.

Nếu ta biết pre-order và in-order của một cây, ta có thể dựng lại chính xác cấu trúc của cây. Ví dụ, cây trên là cây duy nhất có thể có pre-order $[1, 2, 4, 5, 6, 3, 7]$ và in-order $[4, 2, 6, 5, 1, 3, 7]$. Tương tự, post-order và in-order cũng xác định cấu trúc của một cây.

Tuy nhiên, tình huống sẽ khác nếu ta chỉ biết pre-order và post-order của một cây. Trong trường hợp này, có thể có nhiều hơn một cây khớp với các thứ tự đó. Ví dụ, trong cả hai cây



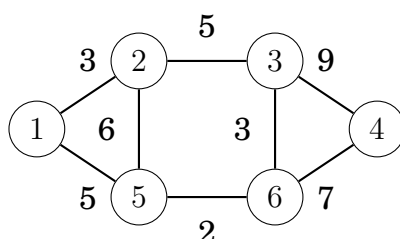
pre-order là $[1, 2]$ và post-order là $[2, 1]$, nhưng cấu trúc của hai cây khác nhau.

Chương 15

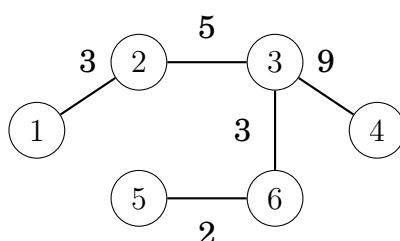
Cây khung

Một **cây khung** (spanning tree) của một đồ thị gồm tất cả các đỉnh của đồ thị và một số cạnh của đồ thị sao cho có đường đi giữa mọi cặp đỉnh. Giống như cây nói chung, cây khung là liên thông và không có chu trình. Thông thường có nhiều cách để xây dựng một cây khung.

Ví dụ, hãy xét đồ thị sau:

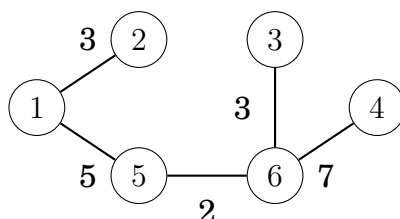


Một cây khung của đồ thị là:

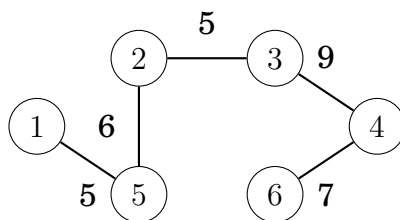


Trọng số của một cây khung là tổng trọng số các cạnh của nó. Ví dụ, trọng số của cây khung trên là $3 + 5 + 9 + 3 + 2 = 22$.

Một **cây khung nhỏ nhất** (minimum spanning tree) là cây khung có trọng số nhỏ nhất có thể. Trọng số của một cây khung nhỏ nhất trong đồ thị ví dụ là 20, và một cây như vậy có thể được xây dựng như sau:



Tương tự, một **cây khung lớn nhất** là cây khung có trọng số lớn nhất có thể. Trọng số của một cây khung lớn nhất trong đồ thị ví dụ là 32:



Lưu ý rằng một đồ thị có thể có nhiều cây khung nhỏ nhất và lớn nhất, nên các cây này không nhất thiết là duy nhất.

Hóa ra có thể dùng một số phương pháp tham lam để xây dựng cây khung nhỏ nhất và cây khung lớn nhất. Trong chương này, ta thảo luận hai thuật toán xử lý các cạnh của đồ thị theo thứ tự trọng số. Ta tập trung vào việc tìm cây khung nhỏ nhất, nhưng các thuật toán tương tự có thể tìm cây khung lớn nhất bằng cách xử lý các cạnh theo thứ tự ngược lại.

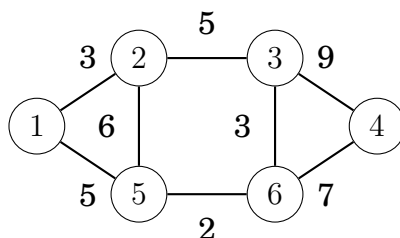
15.1 Thuật toán Kruskal

Trong **thuật toán Kruskal**¹, cây khung ban đầu chỉ chứa các đỉnh của đồ thị và chưa chứa cạnh nào. Sau đó thuật toán duyệt qua các cạnh theo thứ tự trọng số, và luôn thêm một cạnh vào cây nếu cạnh đó không tạo ra chu trình.

Thuật toán duy trì các thành phần của cây. Ban đầu, mỗi đỉnh của đồ thị thuộc một thành phần riêng. Mỗi khi một cạnh được thêm vào cây, hai thành phần được hợp nhất. Cuối cùng, tất cả các đỉnh thuộc cùng một thành phần, và ta đã tìm được một cây khung nhỏ nhất.

Ví dụ

Hãy xét cách thuật toán Kruskal xử lý đồ thị sau:



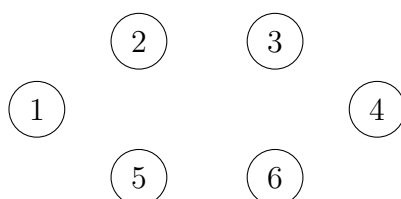
Bước đầu tiên của thuật toán là sắp xếp các cạnh theo thứ tự tăng dần của trọng số. Kết quả là danh sách sau:

¹Thuật toán được J. B. Kruskal công bố vào năm 1956 [48].

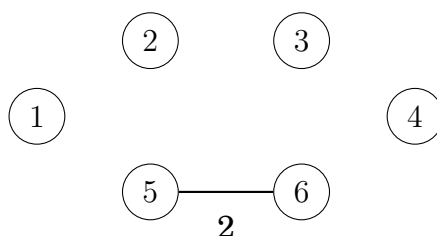
cạnh	trọng số
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Sau đó, thuật toán duyệt qua danh sách và thêm mỗi cạnh vào cây nếu nó nối hai thành phần riêng biệt.

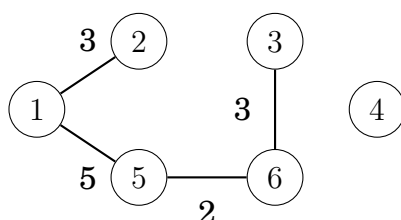
Ban đầu, mỗi đỉnh nằm trong thành phần riêng của nó:



Cạnh đầu tiên được thêm vào cây là cạnh 5-6, tạo thành phần $\{5, 6\}$ bằng cách hợp nhất hai thành phần $\{5\}$ và $\{6\}$:



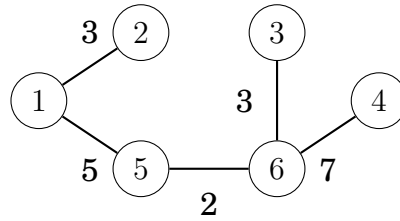
Sau đó, các cạnh 1-2, 3-6 và 1-5 được thêm vào theo cách tương tự:



Sau các bước đó, hầu hết các thành phần đã được hợp nhất và trong cây còn hai thành phần: $\{1, 2, 3, 5, 6\}$ và $\{4\}$.

Cạnh tiếp theo trong danh sách là cạnh 2-3, nhưng nó sẽ không được đưa vào cây, vì các đỉnh 2 và 3 đã nằm trong cùng một thành phần. Vì lý do tương tự, cạnh 2-5 cũng sẽ không được đưa vào cây.

Cuối cùng, cạnh 4–6 sẽ được đưa vào cây:

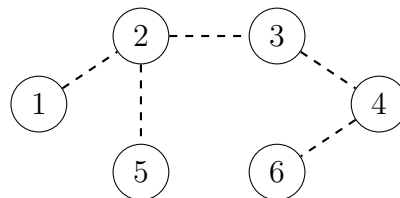


Sau bước này, thuật toán sẽ không thêm cạnh mới nào nữa, vì đồ thị đã liên thông và có đường đi giữa mọi cặp đỉnh. Đồ thị thu được là một cây khung nhỏ nhất có trọng số $2 + 3 + 3 + 5 + 7 = 20$.

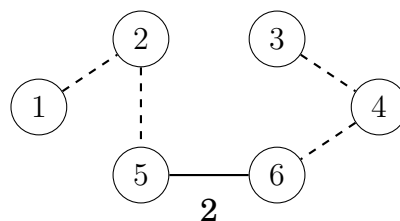
Vì sao thuật toán đúng?

Một câu hỏi tự nhiên là vì sao thuật toán Kruskal đúng. Vì sao chiến lược tham lam bảo đảm rằng ta sẽ tìm được một cây khung nhỏ nhất?

Hãy xem điều gì xảy ra nếu cạnh có trọng số nhỏ nhất của đồ thị *không* nằm trong cây khung. Ví dụ, giả sử một cây khung của đồ thị trước không chứa cạnh trọng số nhỏ nhất 5–6. Ta không biết cấu trúc chính xác của cây khung như vậy, nhưng trong mọi trường hợp nó phải chứa một số cạnh. Giả sử cây đó như sau:



Tuy nhiên, cây trên không thể là cây khung nhỏ nhất của đồ thị. Lý do là ta có thể xóa một cạnh khỏi cây và thay bằng cạnh có trọng số nhỏ nhất 5–6. Việc này tạo ra một cây khung có trọng số *nhỏ hơn*:



Vì lý do này, việc đưa cạnh có trọng số nhỏ nhất vào cây luôn là tối ưu để tạo ra một cây khung nhỏ nhất. Dùng lập luận tương tự, ta có thể chứng minh rằng việc thêm cạnh tiếp theo theo thứ tự trọng số vào cây cũng là tối ưu, và cứ tiếp tục như vậy. Do đó, thuật toán Kruskal hoạt động đúng và luôn tạo ra một cây khung nhỏ nhất.

Cài đặt

Khi cài đặt thuật toán Kruskal, việc dùng biểu diễn đồ thị bằng danh sách cạnh là thuận tiện. Giai đoạn đầu tiên của thuật toán sắp xếp các cạnh trong danh sách trong thời gian $O(m \log m)$. Sau đó, giai đoạn thứ hai của thuật toán xây dựng cây khung nhỏ nhất như sau:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

Vòng lặp duyệt qua các cạnh trong danh sách và luôn xử lý một cạnh $a-b$, trong đó a và b là hai đỉnh. Cần hai hàm: hàm `same` xác định a và b có nằm trong cùng một thành phần hay không, và hàm `unite` hợp nhất các thành phần chứa a và b .

Bài toán là cài đặt hiệu quả hai hàm `same` và `unite`. Một khả năng là cài đặt hàm `same` như một lượt duyệt đồ thị và kiểm tra xem ta có thể đi từ đỉnh a tới đỉnh b hay không. Tuy nhiên, độ phức tạp thời gian của hàm như vậy sẽ là $O(n + m)$ và thuật toán thu được sẽ chậm, vì hàm `same` sẽ được gọi cho mỗi cạnh trong đồ thị.

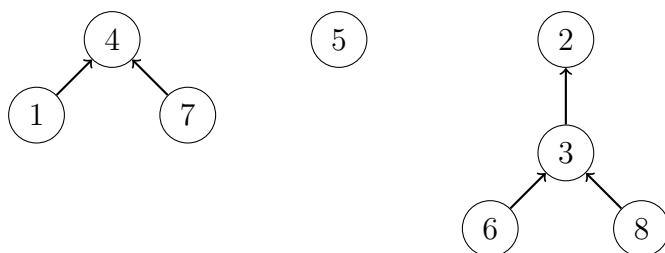
Ta sẽ giải quyết bài toán bằng cấu trúc `union-find`, cài đặt cả hai hàm trong thời gian $O(\log n)$. Do đó, sau khi sắp xếp danh sách cạnh, độ phức tạp thời gian của thuật toán Kruskal sẽ là $O(m \log n)$.

15.2 Cấu trúc union-find

Một **cấu trúc union-find** duy trì một tập hợp các tập. Các tập này rời nhau, nên không phần tử nào thuộc nhiều hơn một tập. Cấu trúc hỗ trợ hai thao tác thời gian $O(\log n)$: thao tác `unite` hợp nhất hai tập, và thao tác `find` tìm đại diện của tập chứa một phần tử cho trước².

Cấu trúc

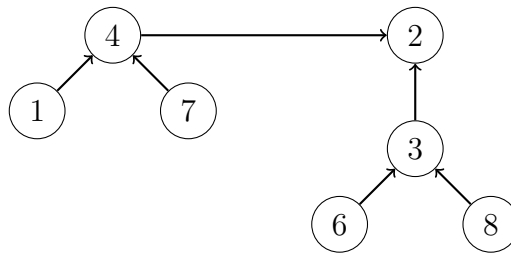
Trong cấu trúc `union-find`, một phần tử trong mỗi tập là đại diện của tập đó, và có một chuỗi từ mọi phần tử khác của tập tới đại diện. Ví dụ, giả sử các tập là $\{1, 4, 7\}$, $\{5\}$ và $\{2, 3, 6, 8\}$:



²Cấu trúc được trình bày ở đây được J. D. Hopcroft và J. D. Ullman giới thiệu vào năm 1971 [38]. Sau đó, vào năm 1975, R. E. Tarjan nghiên cứu một biến thể tinh vi hơn của cấu trúc này [64], hiện được bàn trong nhiều sách giáo khoa thuật toán.

Trong trường hợp này, các đại diện của các tập là 4, 5 và 2. Ta có thể tìm đại diện của bất kỳ phần tử nào bằng cách đi theo chuỗi bắt đầu từ phần tử đó. Ví dụ, phần tử 2 là đại diện cho phần tử 6, vì ta đi theo chuỗi $6 \rightarrow 3 \rightarrow 2$. Hai phần tử thuộc cùng một tập khi và chỉ khi đại diện của chúng giống nhau.

Hai tập có thể được hợp nhất bằng cách nối đại diện của một tập tới đại diện của tập kia. Ví dụ, các tập $\{1, 4, 7\}$ và $\{2, 3, 6, 8\}$ có thể được hợp nhất như sau:



Tập thu được chứa các phần tử $\{1, 2, 3, 4, 6, 7, 8\}$. Từ đây, phần tử 2 là đại diện cho toàn bộ tập và đại diện cũ 4 trở về phần tử 2.

Hiệu quả của cấu trúc union-find phụ thuộc vào cách các tập được hợp nhất. Hóa ra ta có thể theo một chiến lược đơn giản: luôn nối đại diện của tập *nhỏ hơn* tới đại diện của tập *lớn hơn* (hoặc nếu hai tập có cùng kích thước, ta có thể chọn tùy ý). Dùng chiến lược này, độ dài của mọi chuỗi sẽ là $O(\log n)$, nên ta có thể tìm đại diện của bất kỳ phần tử nào một cách hiệu quả bằng cách đi theo chuỗi tương ứng.

Cài đặt

Cấu trúc union-find có thể được cài đặt bằng các mảng. Trong cài đặt sau, mảng `link` chứa với mỗi phần tử phần tử tiếp theo trong chuỗi, hoặc chính phần tử đó nếu nó là đại diện, và mảng `size` cho biết với mỗi đại diện kích thước của tập tương ứng.

Ban đầu, mỗi phần tử thuộc một tập riêng:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

Hàm `find` trả về đại diện của một phần tử x . Đại diện có thể được tìm bằng cách đi theo chuỗi bắt đầu tại x .

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

Hàm `same` kiểm tra hai phần tử a và b có thuộc cùng một tập hay không. Điều này có thể làm dễ dàng bằng cách dùng hàm `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

```
}
```

Hàm `unite` hợp nhất các tập chứa hai phần tử a và b (hai phần tử phải nằm trong các tập khác nhau). Hàm trước hết tìm đại diện của các tập rồi nối tập nhỏ hơn vào tập lớn hơn.

```
void unite(int a, int b) {  
    a = find(a);  
    b = find(b);  
    if (size[a] < size[b]) swap(a,b);  
    size[a] += size[b];  
    link[b] = a;  
}
```

Độ phức tạp thời gian của hàm `find` là $O(\log n)$ nếu độ dài của mỗi chuỗi là $O(\log n)$. Khi đó, các hàm `same` và `unite` cũng chạy trong thời gian $O(\log n)$. Hàm `unite` bảo đảm độ dài của mỗi chuỗi là $O(\log n)$ bằng cách nối tập nhỏ hơn vào tập lớn hơn.

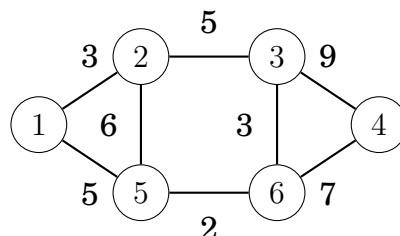
15.3 Thuật toán Prim

Thuật toán Prim³ là một phương pháp khác để tìm cây khung nhỏ nhất. Thuật toán trước hết thêm một đỉnh tùy ý vào cây. Sau đó, thuật toán luôn chọn một cạnh có trọng số nhỏ nhất thêm được một đỉnh mới vào cây. Cuối cùng, tất cả các đỉnh đã được thêm vào cây và ta đã tìm được một cây khung nhỏ nhất.

Thuật toán Prim giống thuật toán Dijkstra. Điểm khác biệt là thuật toán Dijkstra luôn chọn một cạnh có khoảng cách từ đỉnh xuất phát là nhỏ nhất, còn thuật toán Prim chỉ đơn giản chọn cạnh trọng số nhỏ nhất thêm được một đỉnh mới vào cây.

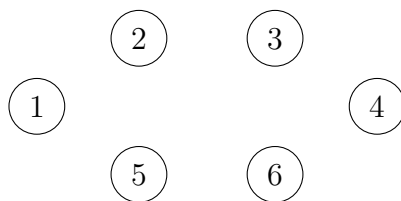
Ví dụ

Hãy xét cách thuật toán Prim hoạt động trên đồ thị sau:

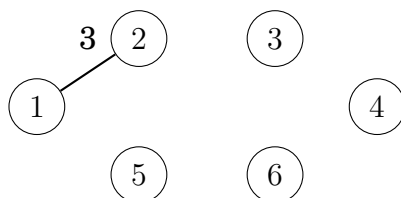


Ban đầu, chưa có cạnh nào giữa các đỉnh:

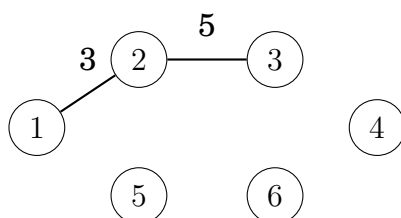
³Thuật toán được đặt theo tên R. C. Prim, người công bố nó vào năm 1957 [54]. Tuy nhiên, cùng thuật toán này đã được V. Jarník phát hiện từ năm 1930.



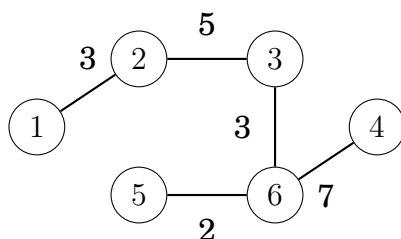
Một đỉnh tùy ý có thể làm đỉnh xuất phát, nên ta chọn đỉnh 1. Trước tiên, ta thêm đỉnh 2, được nối bằng một cạnh có trọng số 3:



Sau đó, có hai cạnh trọng số 5, nên ta có thể thêm đỉnh 3 hoặc đỉnh 5 vào cây. Hãy thêm đỉnh 3 trước:



Quá trình tiếp tục cho tới khi tất cả các đỉnh đã được đưa vào cây:



Cài đặt

Giống thuật toán Dijkstra, thuật toán Prim có thể được cài đặt hiệu quả bằng hàng đợi ưu tiên. Hàng đợi ưu tiên nên chứa tất cả các đỉnh có thể được nối với thành phần hiện tại bằng một cạnh, theo thứ tự tăng dần của trọng số của các cạnh tương ứng.

Độ phức tạp thời gian của thuật toán Prim là $O(n + m \log m)$, bằng với độ phức tạp thời gian của thuật toán Dijkstra. Trong thực tế, cả thuật toán Prim và Kruskal đều hiệu quả, và việc chọn thuật toán nào là vấn đề sở thích. Dù vậy, hầu hết lập trình viên thi đấu sử dụng thuật toán Kruskal.

Chương 16

Đồ thị có hướng

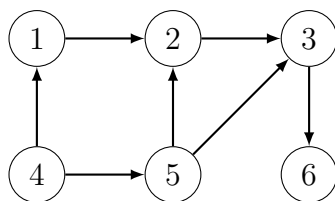
Trong chương này, ta tập trung vào hai lớp đồ thị có hướng:

- **Đồ thị không chu trình:** Trong đồ thị không có chu trình, vì vậy không có đường đi từ bất kỳ đỉnh nào về lại chính nó¹.
- **Đồ thị kế tiếp:** Bậc ra của mỗi đỉnh là 1, vì vậy mỗi đỉnh có đúng một đỉnh kế tiếp.

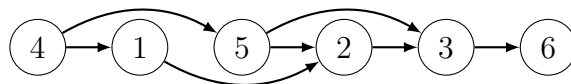
Hóa ra trong cả hai trường hợp, ta có thể thiết kế các thuật toán hiệu quả dựa trên những tính chất đặc biệt của các đồ thị này.

16.1 Sắp xếp topo

Một **thứ tự topo** (topological sort) là một cách sắp xếp các đỉnh của một đồ thị có hướng sao cho nếu có đường đi từ đỉnh a đến đỉnh b , thì đỉnh a xuất hiện trước đỉnh b trong thứ tự đó. Ví dụ, với đồ thị



một thứ tự topo là $[4, 1, 5, 2, 3, 6]$:



Một đồ thị không chu trình luôn có thứ tự topo. Tuy nhiên, nếu đồ thị chứa một chu trình, ta không thể tạo một thứ tự topo, vì không có đỉnh nào của chu trình có thể xuất hiện trước các đỉnh khác của chu trình trong thứ tự đó. Hóa ra tìm kiếm theo chiều sâu có thể được dùng vừa để kiểm tra một đồ thị có hướng có chứa chu trình hay không, vừa để xây dựng thứ tự topo nếu đồ thị không chứa chu trình.

¹Đồ thị có hướng không chu trình đôi khi được gọi là DAG.

Thuật toán

Ý tưởng là duyệt qua các đỉnh của đồ thị và luôn bắt đầu một tìm kiếm theo chiều sâu tại đỉnh hiện tại nếu nó chưa được xử lý. Trong quá trình tìm kiếm, các đỉnh có ba trạng thái có thể:

- trạng thái 0: đỉnh chưa được xử lý (trắng)
- trạng thái 1: đỉnh đang được xử lý (xám nhạt)
- trạng thái 2: đỉnh đã được xử lý (xám đậm)

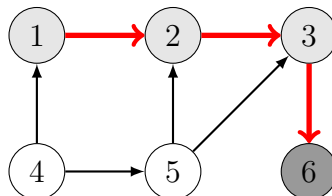
Ban đầu, trạng thái của mỗi đỉnh là 0. Khi một lần tìm kiếm chạm tới một đỉnh lần đầu tiên, trạng thái của nó trở thành 1. Cuối cùng, sau khi mọi đỉnh kế tiếp của đỉnh đó đã được xử lý, trạng thái của nó trở thành 2.

Nếu đồ thị chứa một chu trình, ta sẽ phát hiện điều này trong quá trình tìm kiếm, vì sớm hay muộn ta sẽ đi tới một đỉnh có trạng thái 1. Trong trường hợp này, không thể xây dựng thứ tự topo.

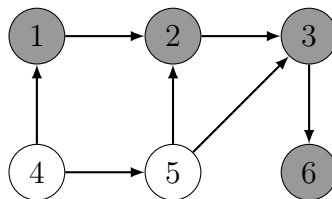
Nếu đồ thị không chứa chu trình, ta có thể xây dựng một thứ tự topo bằng cách thêm mỗi đỉnh vào một danh sách khi trạng thái của đỉnh đó trở thành 2. Danh sách này theo thứ tự ngược lại là một thứ tự topo.

Ví dụ 1

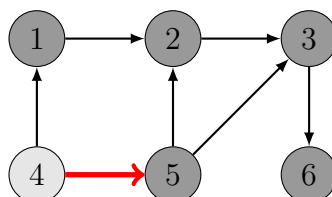
Trong đồ thị ví dụ, lần tìm kiếm đầu tiên đi từ đỉnh 1 đến đỉnh 6:



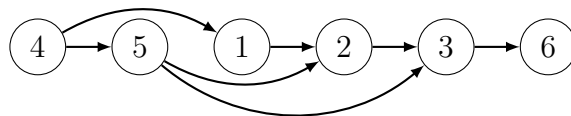
Bây giờ đỉnh 6 đã được xử lý, nên nó được thêm vào danh sách. Sau đó, các đỉnh 3, 2 và 1 cũng được thêm vào danh sách:



Tại thời điểm này, danh sách là [6, 3, 2, 1]. Lần tìm kiếm tiếp theo bắt đầu tại đỉnh 4:



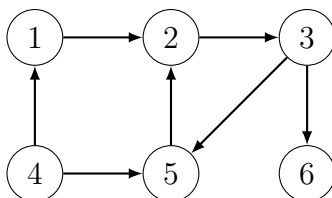
Do đó, danh sách cuối cùng là $[6, 3, 2, 1, 5, 4]$. Ta đã xử lý mọi đỉnh, nên đã tìm được một thứ tự topo. Thứ tự topo là danh sách đảo ngược $[4, 5, 1, 2, 3, 6]$:



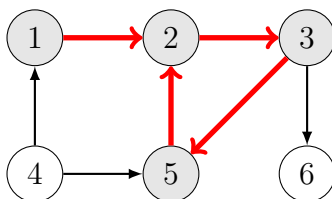
Lưu ý rằng thứ tự topo không duy nhất, và một đồ thị có thể có nhiều thứ tự topo.

Ví dụ 2

Bây giờ hãy xét một đồ thị mà ta không thể xây dựng thứ tự topo, vì đồ thị chứa một chu trình:



Lần tìm kiếm diễn ra như sau:



Lần tìm kiếm chạm tới đỉnh 2 có trạng thái 1, điều này có nghĩa là đồ thị chứa một chu trình. Trong ví dụ này, có chu trình $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

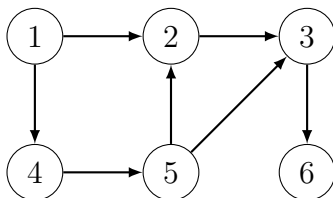
16.2 Quy hoạch động

Nếu một đồ thị có hướng không chu trình, ta có thể áp dụng quy hoạch động cho nó. Ví dụ, ta có thể giải hiệu quả các bài toán sau về đường đi từ một đỉnh bắt đầu đến một đỉnh kết thúc:

- có bao nhiêu đường đi khác nhau?
- đường đi ngắn nhất/dài nhất là gì?
- số cạnh nhỏ nhất/lớn nhất trong một đường đi là bao nhiêu?
- những đỉnh nào chắc chắn xuất hiện trong mọi đường đi?

Đếm số đường đi

Ví dụ, hãy tính số đường đi từ đỉnh 1 đến đỉnh 6 trong đồ thị sau:



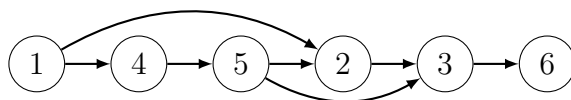
Có tổng cộng ba đường đi như vậy:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

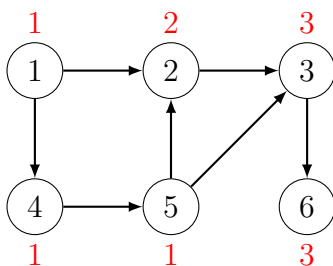
Gọi $\text{paths}(x)$ là số đường đi từ đỉnh 1 đến đỉnh x . Trường hợp cơ sở là $\text{paths}(1) = 1$. Sau đó, để tính các giá trị khác của $\text{paths}(x)$, ta có thể dùng công thức truy hồi

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

trong đó $a_1, a_2, \dots, a_k$ là các đỉnh có cạnh đi tới x . Vì đồ thị không chu trình, các giá trị $\text{paths}(x)$ có thể được tính theo thứ tự topo. Một thứ tự topo cho đồ thị trên như sau:



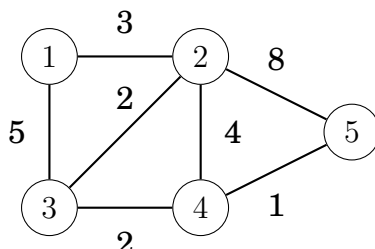
Do đó, số đường đi như sau:



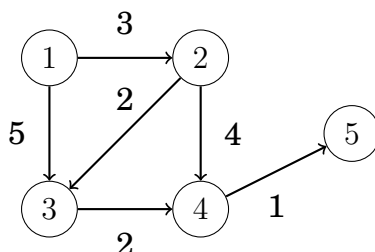
Ví dụ, để tính giá trị $\text{paths}(3)$, ta có thể dùng công thức $\text{paths}(2) + \text{paths}(5)$, vì có các cạnh từ đỉnh 2 và 5 đến đỉnh 3. Vì $\text{paths}(2) = 2$ và $\text{paths}(5) = 1$, ta kết luận rằng $\text{paths}(3) = 3$.

Mở rộng thuật toán Dijkstra

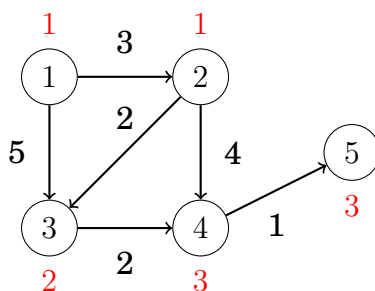
Một sản phẩm phụ của thuật toán Dijkstra là một đồ thị có hướng không chu trình, cho biết với mỗi đỉnh của đồ thị ban đầu các cách có thể để đi tới đỉnh đó bằng một đường đi ngắn nhất từ đỉnh bắt đầu. Ta có thể áp dụng quy hoạch động lên đồ thị này. Ví dụ, trong đồ thị



các đường đi ngắn nhất từ đỉnh 1 có thể dùng các cạnh sau:



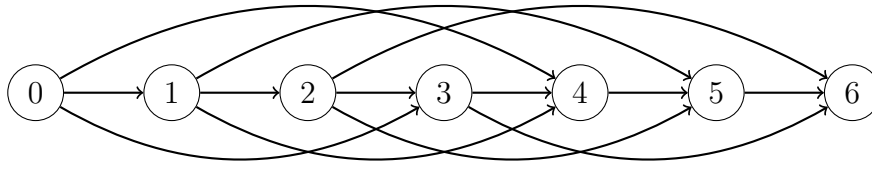
Bây giờ, chẳng hạn, ta có thể tính số đường đi ngắn nhất từ đỉnh 1 đến đỉnh 5 bằng quy hoạch động:



Biểu diễn bài toán bằng đồ thị

Thực ra, mọi bài toán quy hoạch động đều có thể được biểu diễn dưới dạng một đồ thị có hướng không chu trình. Trong đồ thị như vậy, mỗi đỉnh tương ứng với một trạng thái quy hoạch động và các cạnh cho biết các trạng thái phụ thuộc vào nhau như thế nào.

Ví dụ, xét bài toán tạo ra tổng tiền n bằng các đồng xu $\{c_1, c_2, \dots, c_k\}$. Trong bài toán này, ta có thể xây dựng một đồ thị trong đó mỗi đỉnh tương ứng với một tổng tiền, và các cạnh cho biết các đồng xu có thể được chọn như thế nào. Ví dụ, với các đồng xu $\{1, 3, 4\}$ và $n = 6$, đồ thị như sau:



Với biểu diễn này, đường đi ngắn nhất từ đỉnh 0 đến đỉnh n tương ứng với một lời giải dùng số đồng xu ít nhất, và tổng số đường đi từ đỉnh 0 đến đỉnh n bằng tổng số lời giải.

16.3 Đường đi kế tiếp

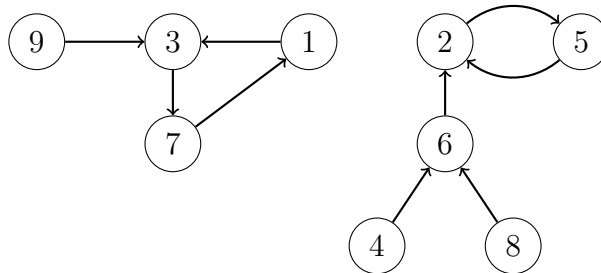
Trong phần còn lại của chương, ta sẽ tập trung vào **đồ thị kế tiếp**. Trong các đồ thị này, bậc ra của mỗi đỉnh là 1, tức là có đúng một cạnh xuất phát từ mỗi đỉnh. Một đồ thị kế tiếp gồm một hoặc nhiều thành phần, mỗi thành phần chứa một chu trình và một số đường đi dẫn vào chu trình đó.

Đồ thị kế tiếp đôi khi được gọi là **đồ thị hàm** (functional graph). Lý do là mọi đồ thị kế tiếp tương ứng với một hàm định nghĩa các cạnh của đồ thị. Tham số của hàm là một đỉnh của đồ thị, và hàm trả về đỉnh kế tiếp của đỉnh đó.

Ví dụ, hàm

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

định nghĩa đồ thị sau:



Vì mỗi đỉnh của đồ thị kế tiếp có đúng một đỉnh kế tiếp, ta cũng có thể định nghĩa hàm $\text{succ}(x, k)$ trả về đỉnh mà ta sẽ tới nếu bắt đầu ở đỉnh x và đi k bước về phía trước. Ví dụ, trong đồ thị trên $\text{succ}(4, 6) = 2$, vì ta sẽ tới đỉnh 2 sau khi đi 6 bước từ đỉnh 4:



Cách trực tiếp để tính một giá trị $\text{succ}(x, k)$ là bắt đầu tại đỉnh x và đi k bước, mất thời gian $O(k)$. Tuy nhiên, bằng tiền xử lý, mọi giá trị $\text{succ}(x, k)$ có thể được tính chỉ trong thời gian $O(\log k)$.

Ý tưởng là tính trước mọi giá trị $\text{succ}(x, k)$ trong đó k là lũy thừa của hai và không vượt quá u , với u là số bước lớn nhất mà ta sẽ từng đi. Điều này có thể thực hiện hiệu quả, vì ta có thể dùng công thức truy hồi sau:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Việc tính trước các giá trị mất thời gian $O(n \log u)$, vì $O(\log u)$ giá trị được tính cho mỗi đỉnh. Trong đồ thị trên, các giá trị đầu tiên như sau:

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x, 1)$	3	5	7	6	2	2	1	6	3
$\text{succ}(x, 2)$	7	2	1	2	5	5	3	2	7
$\text{succ}(x, 4)$	3	2	7	2	5	5	1	2	3
$\text{succ}(x, 8)$	7	2	1	2	5	5	3	2	7
...									

Sau đó, mọi giá trị $\text{succ}(x, k)$ có thể được tính bằng cách biểu diễn số bước k thành tổng các lũy thừa của hai. Ví dụ, nếu ta muốn tính giá trị $\text{succ}(x, 11)$, trước hết ta lập biểu diễn $11 = 8 + 2 + 1$. Dùng biểu diễn đó,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Ví dụ, trong đồ thị trước đó

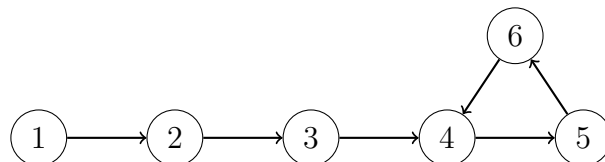
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Một biểu diễn như vậy luôn gồm $O(\log k)$ phần, nên việc tính một giá trị $\text{succ}(x, k)$ mất thời gian $O(\log k)$.

16.4 Phát hiện chu trình

Xét một đồ thị kế tiếp chỉ chứa một đường đi kết thúc bằng một chu trình. Ta có thể đặt các câu hỏi sau: nếu bắt đầu đi từ đỉnh đầu, đỉnh đầu tiên trong chu trình là đỉnh nào và chu trình chứa bao nhiêu đỉnh?

Ví dụ, trong đồ thị



ta bắt đầu đi từ đỉnh 1, đỉnh đầu tiên thuộc chu trình là đỉnh 4, và chu trình gồm ba đỉnh (4, 5 và 6).

Một cách đơn giản để phát hiện chu trình là đi trong đồ thị và theo dõi mọi đỉnh đã được thăm. Khi một đỉnh được thăm lần thứ hai, ta có thể kết

luận rằng đỉnh đó là đỉnh đầu tiên trong chu trình. Phương pháp này chạy trong thời gian $O(n)$ và cũng dùng bộ nhớ $O(n)$.

Tuy nhiên, có các thuật toán tốt hơn để phát hiện chu trình. Độ phức tạp thời gian của các thuật toán như vậy vẫn là $O(n)$, nhưng chúng chỉ dùng bộ nhớ $O(1)$. Đây là cải tiến quan trọng nếu n lớn. Tiếp theo ta sẽ thảo luận thuật toán Floyd, thuật toán đạt được các tính chất này.

Thuật toán Floyd

Thuật toán Floyd (Floyd's algorithm)² đi về phía trước trong đồ thị bằng hai con trỏ a và b . Cả hai con trỏ bắt đầu tại một đỉnh x , là đỉnh bắt đầu của đồ thị. Sau đó, ở mỗi lượt, con trỏ a đi một bước, còn con trỏ b đi hai bước. Quá trình tiếp tục cho đến khi hai con trỏ gặp nhau:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

Tại thời điểm này, con trỏ a đã đi k bước và con trỏ b đã đi $2k$ bước, vì vậy độ dài chu trình chia hết k . Do đó, đỉnh đầu tiên thuộc chu trình có thể được tìm bằng cách đưa con trỏ a về đỉnh x và cho hai con trỏ tiến từng bước cho đến khi chúng gặp lại nhau.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

Sau đó, độ dài của chu trình có thể được tính như sau:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

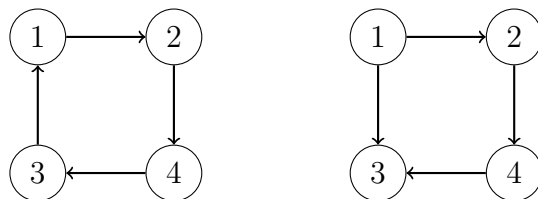
²Ý tưởng của thuật toán được nhắc tới trong [46] và được gán cho R. W. Floyd; tuy nhiên, không rõ Floyd có thực sự phát hiện ra thuật toán hay không.

Chương 17

Liên thông mạnh

Trong đồ thị có hướng, các cạnh chỉ có thể được đi theo một chiều, vì vậy ngay cả khi đồ thị liên thông, điều đó cũng không đảm bảo rằng sẽ có đường đi từ một đỉnh đến một đỉnh khác. Vì lý do này, ta cần định nghĩa một khái niệm mới mạnh hơn tính liên thông.

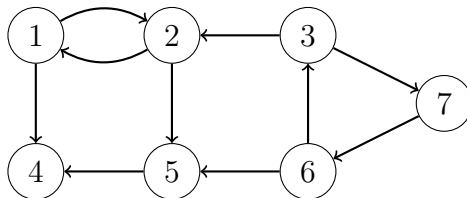
Một đồ thị là **liên thông mạnh** nếu có đường đi từ bất kỳ đỉnh nào đến mọi đỉnh khác trong đồ thị. Ví dụ, trong hình sau, đồ thị bên trái là liên thông mạnh còn đồ thị bên phải thì không.



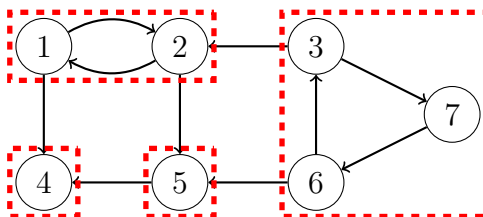
Đồ thị bên phải không liên thông mạnh vì, chẳng hạn, không có đường đi từ đỉnh 2 đến đỉnh 1.

Các **thành phần liên thông mạnh** (strongly connected components) của một đồ thị chia đồ thị thành các phần liên thông mạnh lớn nhất có thể. Các thành phần liên thông mạnh tạo thành một **đồ thị thành phần** có hướng không chu trình, biểu diễn cấu trúc sâu hơn của đồ thị ban đầu.

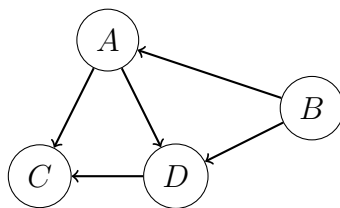
Ví dụ, với đồ thị



các thành phần liên thông mạnh như sau:



Đồ thị thành phần tương ứng như sau:



Các thành phần là $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ và $D = \{5\}$.

Đồ thị thành phần là một đồ thị có hướng không chu trình, vì vậy nó dễ xử lý hơn đồ thị ban đầu. Vì đồ thị không chứa chu trình, ta luôn có thể xây dựng một thứ tự topo và dùng các kỹ thuật quy hoạch động như những kỹ thuật đã trình bày trong Chương 16.

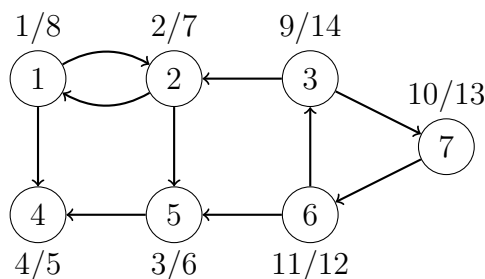
17.1 Thuật toán Kosaraju

Thuật toán Kosaraju (Kosaraju's algorithm)¹ là một phương pháp hiệu quả để tìm các thành phần liên thông mạnh của một đồ thị có hướng. Thuật toán thực hiện hai lần tìm kiếm theo chiều sâu: lần tìm kiếm thứ nhất xây dựng một danh sách các đỉnh theo cấu trúc của đồ thị, và lần tìm kiếm thứ hai tạo ra các thành phần liên thông mạnh.

Tìm kiếm 1

Pha thứ nhất của thuật toán Kosaraju xây dựng một danh sách các đỉnh theo thứ tự mà một tìm kiếm theo chiều sâu xử lý chúng. Thuật toán duyệt qua các đỉnh và bắt đầu một tìm kiếm theo chiều sâu tại mỗi đỉnh chưa được xử lý. Mỗi đỉnh sẽ được thêm vào danh sách sau khi nó đã được xử lý.

Trong đồ thị ví dụ, các đỉnh được xử lý theo thứ tự sau:



Ký hiệu x/y có nghĩa là việc xử lý đỉnh bắt đầu tại thời điểm x và kết thúc tại thời điểm y . Do đó, danh sách tương ứng như sau:

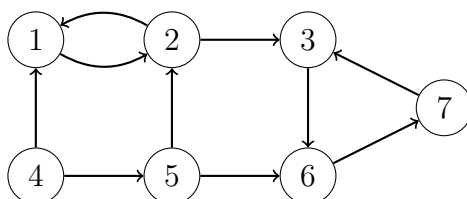
¹Theo [1], S. R. Kosaraju phát minh thuật toán này vào năm 1978 nhưng không công bố. Năm 1981, cùng thuật toán được M. Sharir phát hiện lại và công bố [57].

đỉnh	thời điểm xử lý
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Tìm kiếm 2

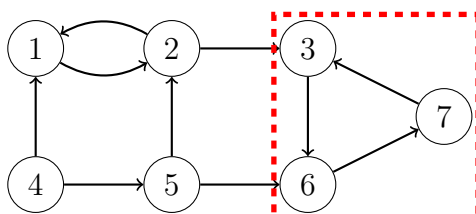
Pha thứ hai của thuật toán tạo ra các thành phần liên thông mạnh của đồ thị. Trước hết, thuật toán đảo chiều mọi cạnh trong đồ thị. Điều này đảm bảo rằng trong lần tìm kiếm thứ hai, ta luôn tìm được các thành phần liên thông mạnh không chứa thêm đỉnh thừa.

Sau khi đảo chiều các cạnh, đồ thị ví dụ như sau:



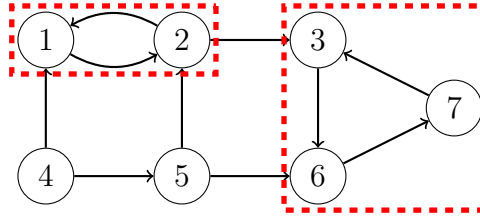
Sau đó, thuật toán duyệt qua danh sách các đỉnh được tạo bởi lần tìm kiếm thứ nhất, theo thứ tự *ngược*. Nếu một đỉnh chưa thuộc thành phần nào, thuật toán tạo một thành phần mới và bắt đầu một tìm kiếm theo chiều sâu để thêm mọi đỉnh mới tìm được trong lần tìm kiếm vào thành phần mới đó.

Trong đồ thị ví dụ, thành phần đầu tiên bắt đầu tại đỉnh 3:

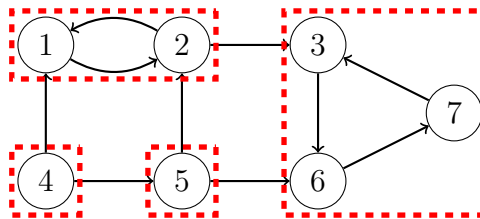


Lưu ý rằng vì mọi cạnh đã bị đảo chiều, thành phần này không "rò" sang các phần khác của đồ thị.

Các đỉnh tiếp theo trong danh sách là 7 và 6, nhưng chúng đã thuộc một thành phần, vì vậy thành phần mới tiếp theo bắt đầu tại đỉnh 1:



Cuối cùng, thuật toán xử lý các đỉnh 5 và 4, tạo ra các thành phần liên thông mạnh còn lại:



Độ phức tạp thời gian của thuật toán là $O(n + m)$, vì thuật toán thực hiện hai lần tìm kiếm theo chiều sâu.

17.2 Bài toán 2SAT

Tính liên thông mạnh cũng liên quan đến **bài toán 2SAT**². Trong bài toán này, ta được cho một công thức logic

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

trong đó mỗi a_i và b_i hoặc là một biến logic ($x_1, x_2, \dots, x_n$) hoặc là phủ định của một biến logic ($\neg x_1, \neg x_2, \dots, \neg x_n$). Các ký hiệu " $\wedge$ " và " $\vee$ " biểu diễn các toán tử logic "và" và "hoặc". Nhiệm vụ của ta là gán cho mỗi biến một giá trị sao cho công thức đúng, hoặc kết luận rằng điều này là không thể.

Ví dụ, công thức

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

đúng khi các biến được gán như sau:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

²Thuật toán được trình bày ở đây được giới thiệu trong [4]. Cũng có một thuật toán tuyến tính nổi tiếng khác [19] dựa trên quay lui.

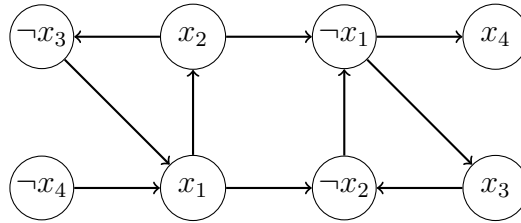
Tuy nhiên, công thức

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

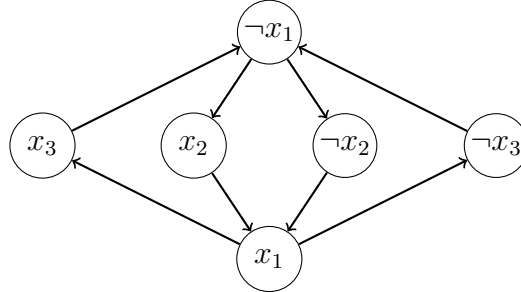
luôn sai, bất kể ta gán các giá trị như thế nào. Lý do là ta không thể chọn một giá trị cho x_1 mà không tạo ra mâu thuẫn. Nếu x_1 sai, cả x_2 và $\neg x_2$ đều phải đúng, điều này là không thể, còn nếu x_1 đúng, cả x_3 và $\neg x_3$ đều phải đúng, điều này cũng không thể.

Bài toán 2SAT có thể được biểu diễn dưới dạng một đồ thị có các đỉnh tương ứng với các biến x_i và các phủ định $\neg x_i$, còn các cạnh xác định các liên hệ giữa các biến. Mỗi cặp $(a_i \vee b_i)$ sinh ra hai cạnh: $\neg a_i \rightarrow b_i$ và $\neg b_i \rightarrow a_i$. Điều này có nghĩa là nếu a_i không đúng, b_i phải đúng, và ngược lại.

Đồ thị cho công thức L_1 là:



Và đồ thị cho công thức L_2 là:



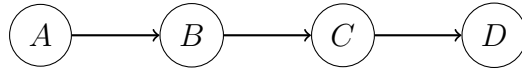
Cấu trúc của đồ thị cho ta biết liệu có thể gán giá trị cho các biến sao cho công thức đúng hay không. Hóa ra điều này thực hiện được khi và chỉ khi không có cặp đỉnh x_i và $\neg x_i$ nào mà cả hai cùng thuộc một thành phần liên thông mạnh. Nếu có các đỉnh như vậy, đồ thị chứa một đường đi từ x_i đến $\neg x_i$ và cũng có một đường đi từ $\neg x_i$ đến x_i , vì vậy cả x_i và $\neg x_i$ đều phải đúng, điều này là không thể.

Trong đồ thị của công thức L_1 , không có cặp đỉnh x_i và $\neg x_i$ nào cùng thuộc một thành phần liên thông mạnh, vì vậy tồn tại lời giải. Trong đồ thị của công thức L_2 , mọi đỉnh đều thuộc cùng một thành phần liên thông mạnh, vì vậy không tồn tại lời giải.

Nếu tồn tại lời giải, có thể tìm giá trị cho các biến bằng cách duyệt qua các đỉnh của đồ thị thành phần theo thứ tự topo ngược. Ở mỗi bước, ta xử lý một thành phần không có cạnh dẫn đến một thành phần chưa xử lý. Nếu các biến trong thành phần đó chưa được gán giá trị, giá trị của chúng sẽ được xác định theo các giá trị trong thành phần, còn nếu chúng đã có giá

trị, chúng được giữ nguyên. Quá trình tiếp tục cho đến khi mỗi biến đã được gán một giá trị.

Đồ thị thành phần cho công thức L_1 như sau:



Các thành phần là $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ và $D = \{x_4\}$. Khi xây dựng lời giải, trước tiên ta xử lý thành phần D , nơi x_4 trở thành đúng. Sau đó, ta xử lý thành phần C , nơi x_1 và x_2 trở thành sai và x_3 trở thành đúng. Mọi biến đã được gán giá trị, vì vậy các thành phần còn lại A và B không làm thay đổi các biến.

Lưu ý rằng phương pháp này hoạt động vì đồ thị có một cấu trúc đặc biệt: nếu có đường đi từ đỉnh x_i đến đỉnh x_j và từ đỉnh x_j đến đỉnh $\neg x_j$, thì đỉnh x_i sẽ không bao giờ trở thành đúng. Lý do là cũng có một đường đi từ đỉnh $\neg x_j$ đến đỉnh $\neg x_i$, và cả x_i lẫn x_j đều trở thành sai.

Một bài toán khó hơn là **bài toán 3SAT**, trong đó mỗi phần của công thức có dạng $(a_i \vee b_i \vee c_i)$. Bài toán này là NP-khó, vì vậy hiện chưa biết thuật toán hiệu quả nào để giải nó.

Chương 18

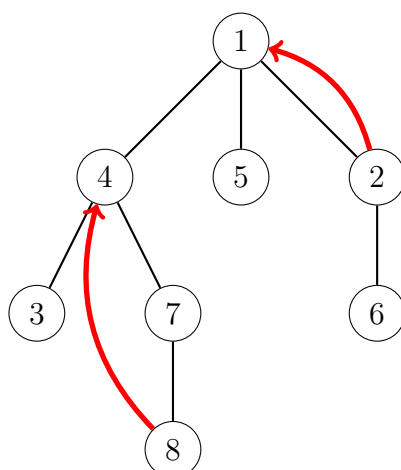
Truy vấn trên cây

Chương này thảo luận các kỹ thuật để xử lý truy vấn trên cây con và đường đi của một cây có gốc. Ví dụ, các truy vấn như vậy là:

- tổ tiên thứ k của một đỉnh là gì?
- tổng các giá trị trong cây con của một đỉnh là bao nhiêu?
- tổng các giá trị trên đường đi giữa hai đỉnh là bao nhiêu?
- tổ tiên chung thấp nhất của hai đỉnh là gì?

18.1 Tìm tổ tiên

Tổ tiên **thứ** k của một đỉnh x trong cây có gốc là đỉnh mà ta sẽ tới nếu đi lên k mức từ x . Gọi $\text{ancestor}(x, k)$ là tổ tiên thứ k của đỉnh x (hoặc 0 nếu không có tổ tiên như vậy). Ví dụ, trong cây sau, $\text{ancestor}(2, 1) = 1$ và $\text{ancestor}(8, 2) = 4$.



Cách đơn giản để tính bất kỳ giá trị $\text{ancestor}(x, k)$ nào là thực hiện một chuỗi k bước đi trong cây. Tuy nhiên, độ phức tạp thời gian của phương pháp này là $O(k)$, có thể chậm, vì một cây có n đỉnh có thể chứa một chuỗi gồm n đỉnh.

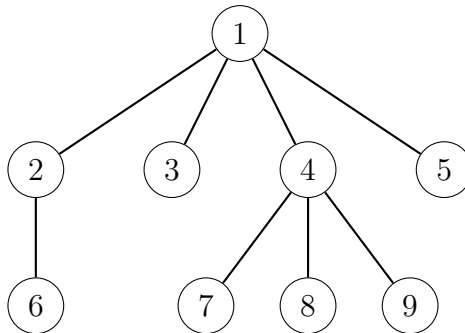
May mắn là bằng một kỹ thuật tương tự kỹ thuật đã dùng trong Chương 16.3, mọi giá trị $\text{ancestor}(x, k)$ có thể được tính hiệu quả trong thời gian $O(\log k)$ sau khi tiền xử lý. Ý tưởng là tính trước mọi giá trị $\text{ancestor}(x, k)$ trong đó $k \leq n$ là lũy thừa của hai. Ví dụ, các giá trị cho cây trên như sau:

x	1	2	3	4	5	6	7	8
$\text{ancestor}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestor}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestor}(x, 4)$	0	0	0	0	0	0	0	0
...								

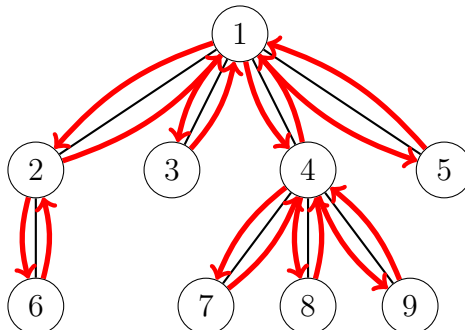
Việc tiền xử lý mất thời gian $O(n \log n)$, vì $O(\log n)$ giá trị được tính cho mỗi đỉnh. Sau đó, mọi giá trị $\text{ancestor}(x, k)$ có thể được tính trong thời gian $O(\log k)$ bằng cách biểu diễn k thành một tổng mà mỗi hạng tử là lũy thừa của hai.

18.2 Cây con và đường đi

Một **mảng duyệt cây** (tree traversal array) chứa các đỉnh của một cây có gốc theo thứ tự mà tìm kiếm theo chiều sâu từ đỉnh gốc thăm chúng. Ví dụ, trong cây



một tìm kiếm theo chiều sâu diễn ra như sau:



Do đó, mảng duyệt cây tương ứng như sau:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Truy vấn cây con

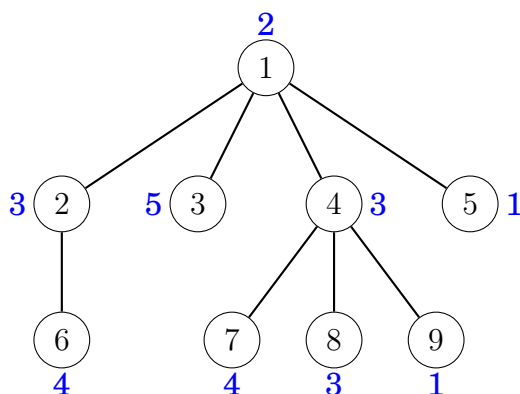
Mỗi cây con của một cây tương ứng với một mảng con của mảng duyệt cây sao cho phần tử đầu tiên của mảng con là đỉnh gốc của cây con. Ví dụ, mảng con sau chứa các đỉnh thuộc cây con của đỉnh 4:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Nhờ tính chất này, ta có thể xử lý hiệu quả các truy vấn liên quan đến cây con của một cây. Ví dụ, xét một bài toán trong đó mỗi đỉnh được gán một giá trị, và nhiệm vụ của ta là hỗ trợ các truy vấn sau:

- cập nhật giá trị của một đỉnh
- tính tổng các giá trị trong cây con của một đỉnh

Xét cây sau, trong đó các số màu xanh là giá trị của các đỉnh. Ví dụ, tổng của cây con tại đỉnh 4 là $3 + 4 + 3 + 1 = 11$.



Ý tưởng là xây dựng một mảng duyệt cây chứa ba giá trị cho mỗi đỉnh: mã định danh của đỉnh, kích thước cây con, và giá trị của đỉnh. Ví dụ, mảng cho cây trên như sau:

id đỉnh	1	2	6	3	4	7	8	9	5
cỡ cây con	9	2	1	1	4	1	1	1	1
giá trị đỉnh	2	3	4	5	3	4	3	1	1

Với mảng này, ta có thể tính tổng các giá trị trong bất kỳ cây con nào bằng cách trước hết tìm kích thước cây con rồi lấy các giá trị của những đỉnh tương ứng. Ví dụ, các giá trị trong cây con của đỉnh 4 có thể được tìm như sau:

id đỉnh	1	2	6	3	4	7	8	9	5
cỡ cây con	9	2	1	1	4	1	1	1	1
giá trị đỉnh	2	3	4	5	3	4	3	1	1

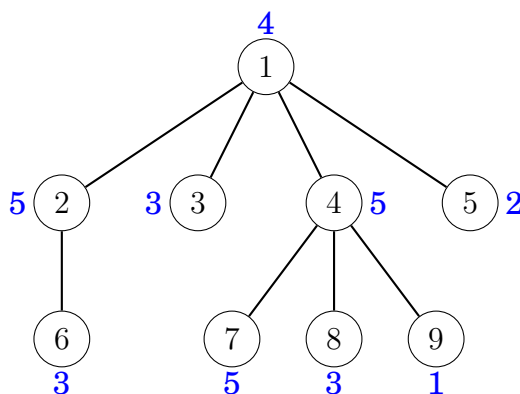
Để trả lời các truy vấn hiệu quả, chỉ cần lưu các giá trị của các đỉnh trong cây Fenwick (Binary Indexed Tree) hoặc cây phân đoạn. Sau đó, ta có thể vừa cập nhật một giá trị vừa tính tổng các giá trị trong thời gian $O(\log n)$.

Truy vấn đường đi

Với mảng duyệt cây, ta cũng có thể tính hiệu quả tổng các giá trị trên đường đi từ đỉnh gốc đến bất kỳ đỉnh nào của cây. Xét một bài toán trong đó nhiệm vụ của ta là hỗ trợ các truy vấn sau:

- thay đổi giá trị của một đỉnh
- tính tổng các giá trị trên đường đi từ gốc đến một đỉnh

Ví dụ, trong cây sau, tổng các giá trị từ đỉnh gốc đến đỉnh 7 là $4+5+5 = 14$:



Ta có thể giải bài toán này như trước, nhưng bây giờ mỗi giá trị ở hàng cuối của mảng là tổng các giá trị trên đường đi từ gốc đến đỉnh đó. Ví dụ, mảng sau tương ứng với cây trên:

id đỉnh	1	2	6	3	4	7	8	9	5
cỗ cây con	9	2	1	1	4	1	1	1	1
tổng đường đi	4	9	12	7	9	14	12	10	6

Khi giá trị của một đỉnh tăng thêm x , tổng của mọi đỉnh trong cây con của nó tăng thêm x . Ví dụ, nếu giá trị của đỉnh 4 tăng thêm 1, mảng thay đổi như sau:

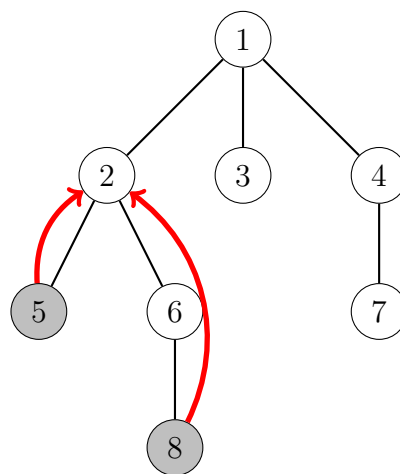
id đỉnh	1	2	6	3	4	7	8	9	5
cỗ cây con	9	2	1	1	4	1	1	1	1
tổng đường đi	4	9	12	7	10	15	13	11	6

Do đó, để hỗ trợ cả hai thao tác, ta cần có khả năng tăng mọi giá trị trong một đoạn và truy xuất một giá trị đơn lẻ. Điều này có thể được thực hiện trong thời gian $O(\log n)$ bằng cây Fenwick hoặc cây phân đoạn (xem Chương 9.4).

18.3 Tổ tiên chung thấp nhất

Tổ tiên chung thấp nhất (lowest common ancestor) của hai đỉnh trong một cây có gốc là đỉnh thấp nhất mà cây con của nó chứa cả hai đỉnh đó. Một bài toán điển hình là xử lý hiệu quả các truy vấn yêu cầu tìm tổ tiên chung thấp nhất của hai đỉnh.

Ví dụ, trong cây sau, tổ tiên chung thấp nhất của các đỉnh 5 và 8 là đỉnh 2:



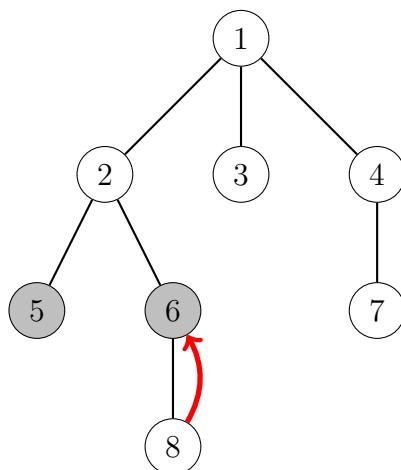
Tiếp theo ta sẽ thảo luận hai kỹ thuật hiệu quả để tìm tổ tiên chung thấp nhất của hai đỉnh.

Phương pháp 1

Một cách giải bài toán là dùng tính chất rằng ta có thể tìm hiệu quả tổ tiên thứ k của bất kỳ đỉnh nào trong cây. Nhờ đó, ta có thể chia bài toán tìm tổ tiên chung thấp nhất thành hai phần.

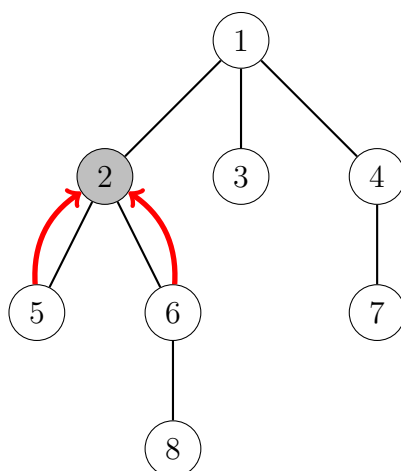
Ta dùng hai con trỏ ban đầu trỏ tới hai đỉnh cần tìm tổ tiên chung thấp nhất. Trước hết, ta di chuyển một trong hai con trỏ lên trên sao cho cả hai con trỏ trỏ tới các đỉnh cùng mức.

Trong tình huống ví dụ, ta di chuyển con trỏ thứ hai lên một mức để nó trỏ tới đỉnh 6, cùng mức với đỉnh 5:



Sau đó, ta xác định số bước nhỏ nhất cần để di chuyển cả hai con trở lên trên sao cho chúng trở tới cùng một đỉnh. Đỉnh mà các con trở tới sau bước này là tổ tiên chung thấp nhất.

Trong tình huống ví dụ, chỉ cần di chuyển cả hai con trở lên một bước tới đỉnh 2, chính là tổ tiên chung thấp nhất:

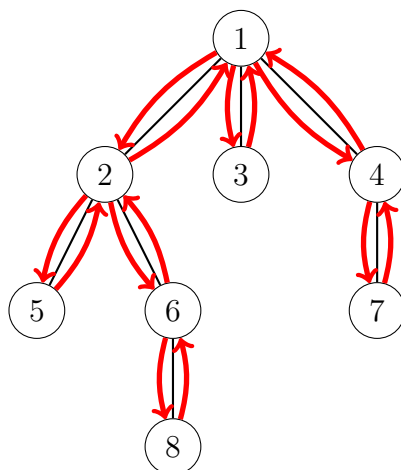


Vì cả hai phần của thuật toán có thể được thực hiện trong thời gian $O(\log n)$ bằng thông tin đã tiền xử lý, ta có thể tìm tổ tiên chung thấp nhất của bất kỳ hai đỉnh nào trong thời gian $O(\log n)$.

Phương pháp 2

Một cách khác để giải bài toán dựa trên mảng duyệt cây¹. Một lần nữa, ý tưởng là duyệt các đỉnh bằng tìm kiếm theo chiều sâu:

¹Thuật toán tổ tiên chung thấp nhất này được trình bày trong [7]. Kỹ thuật này đôi khi được gọi là **kỹ thuật Euler tour** (Euler tour technique) [66].



Tuy nhiên, ta dùng một mảng duyệt cây khác với trước: ta thêm mỗi đỉnh vào mảng *mỗi khi* tìm kiếm theo chiều sâu đi qua đỉnh đó, không chỉ ở lần thăm đầu tiên. Do đó, một đỉnh có k con sẽ xuất hiện $k + 1$ lần trong mảng và tổng cộng có $2n - 1$ đỉnh trong mảng.

Ta lưu hai giá trị trong mảng: mã định danh của đỉnh và độ sâu của đỉnh trong cây. Mảng sau tương ứng với cây trên:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id đỉnh	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
độ sâu	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Bây giờ ta có thể tìm tổ tiên chung thấp nhất của các đỉnh a và b bằng cách tìm đỉnh có độ sâu *nhỏ nhất* giữa các lần xuất hiện của a và b trong mảng. Ví dụ, tổ tiên chung thấp nhất của các đỉnh 5 và 8 có thể được tìm như sau:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id đỉnh	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
độ sâu	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

Đỉnh 5 ở vị trí 2, đỉnh 8 ở vị trí 5, và đỉnh có độ sâu nhỏ nhất giữa các vị trí 2...5 là đỉnh 2 ở vị trí 3 với độ sâu 2. Do đó, tổ tiên chung thấp nhất của các đỉnh 5 và 8 là đỉnh 2.

Vì vậy, để tìm tổ tiên chung thấp nhất của hai đỉnh, chỉ cần xử lý một truy vấn cực tiểu trên đoạn. Vì mảng là tĩnh, ta có thể xử lý các truy vấn như vậy trong thời gian $O(1)$ sau khi tiền xử lý $O(n \log n)$.

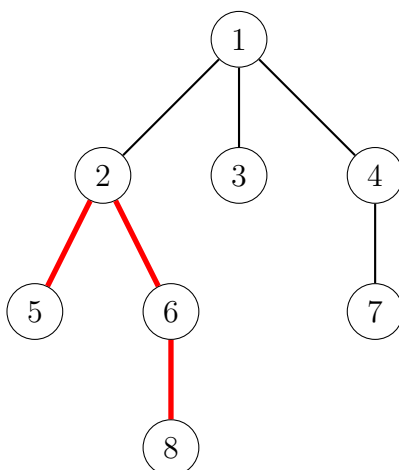
Khoảng cách giữa các đỉnh

Khoảng cách giữa các đỉnh a và b bằng độ dài đường đi từ a đến b . Hóa ra bài toán tính khoảng cách giữa các đỉnh có thể quy về việc tìm tổ tiên chung thấp nhất của chúng.

Trước hết, ta chọn gốc cho cây một cách tùy ý. Sau đó, khoảng cách giữa các đỉnh a và b có thể được tính bằng công thức

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c),$$

trong đó c là tổ tiên chung thấp nhất của a và b và $\text{depth}(s)$ là độ sâu của đỉnh s . Ví dụ, xét khoảng cách giữa các đỉnh 5 và 8:



Tổ tiên chung thấp nhất của các đỉnh 5 và 8 là đỉnh 2. Độ sâu của các đỉnh là $\text{depth}(5) = 3$, $\text{depth}(8) = 4$ và $\text{depth}(2) = 2$, vì vậy khoảng cách giữa các đỉnh 5 và 8 là $3 + 4 - 2 \cdot 2 = 3$.

18.4 Thuật toán offline

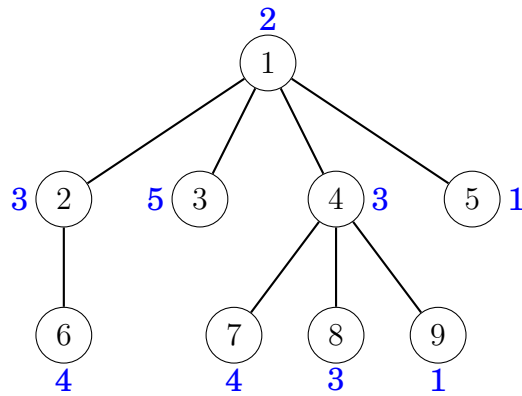
Cho đến nay, ta đã thảo luận các thuật toán *online* cho truy vấn trên cây. Những thuật toán đó có thể xử lý các truy vấn lần lượt sao cho mỗi truy vấn được trả lời trước khi nhận truy vấn tiếp theo.

Tuy nhiên, trong nhiều bài toán, tính chất *online* không cần thiết. Trong phần này, ta tập trung vào các thuật toán *offline*. Các thuật toán này được cho một tập truy vấn có thể được trả lời theo bất kỳ thứ tự nào. Thiết kế một thuật toán *offline* thường dễ hơn so với thuật toán *online*.

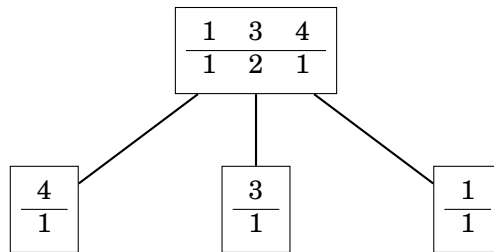
Gộp cấu trúc dữ liệu

Một phương pháp để xây dựng thuật toán *offline* là thực hiện duyệt cây theo chiều sâu và duy trì các cấu trúc dữ liệu tại các đỉnh. Tại mỗi đỉnh s , ta tạo một cấu trúc dữ liệu $d[s]$ dựa trên các cấu trúc dữ liệu của các con của s . Sau đó, dùng cấu trúc dữ liệu này, ta xử lý mọi truy vấn liên quan đến s .

Ví dụ, xét bài toán sau: Ta được cho một cây mà mỗi đỉnh có một giá trị. Nhiệm vụ là xử lý các truy vấn dạng "tính số đỉnh có giá trị x trong cây con của đỉnh s ". Ví dụ, trong cây sau, cây con của đỉnh 4 chứa hai đỉnh có giá trị 3.



Trong bài toán này, ta có thể dùng các cấu trúc map để trả lời truy vấn. Ví dụ, các map cho đỉnh 4 và các con của nó như sau:



Nếu ta tạo một cấu trúc dữ liệu như vậy cho mỗi đỉnh, ta có thể dễ dàng xử lý mọi truy vấn đã cho, vì có thể xử lý mọi truy vấn liên quan đến một đỉnh ngay sau khi tạo cấu trúc dữ liệu của nó. Ví dụ, cấu trúc map ở trên cho đỉnh 4 cho ta biết rằng cây con của nó chứa hai đỉnh có giá trị 3.

Tuy nhiên, việc tạo tất cả cấu trúc dữ liệu từ đầu sẽ quá chậm. Thay vào đó, tại mỗi đỉnh s , ta tạo một cấu trúc dữ liệu ban đầu $d[s]$ chỉ chứa giá trị của s . Sau đó, ta duyệt qua các con của s và *gộp* $d[s]$ với tất cả cấu trúc dữ liệu $d[u]$ trong đó u là một con của s .

Ví dụ, trong cây trên, map cho đỉnh 4 được tạo bằng cách gộp các map sau:



Ở đây map đầu tiên là cấu trúc dữ liệu ban đầu cho đỉnh 4, và ba map còn lại tương ứng với các đỉnh 7, 8 và 9.

Việc gộp tại đỉnh s có thể được thực hiện như sau: Ta duyệt qua các con của s và tại mỗi con u gộp $d[s]$ với $d[u]$. Ta luôn sao chép nội dung từ $d[u]$ sang $d[s]$. Tuy nhiên, trước đó, ta *hoán đổi* nội dung của $d[s]$ và $d[u]$ nếu $d[s]$ nhỏ hơn $d[u]$. Bằng cách này, mỗi giá trị chỉ bị sao chép $O(\log n)$ lần trong quá trình duyệt cây, đảm bảo thuật toán hiệu quả.

Để hoán đổi nội dung của hai cấu trúc dữ liệu a và b một cách hiệu quả, ta chỉ cần dùng đoạn mã sau:

```
swap(a,b);
```

Đoạn mã trên được đảm bảo chạy trong thời gian hằng số khi a và b là các cấu trúc dữ liệu của thư viện chuẩn C++.

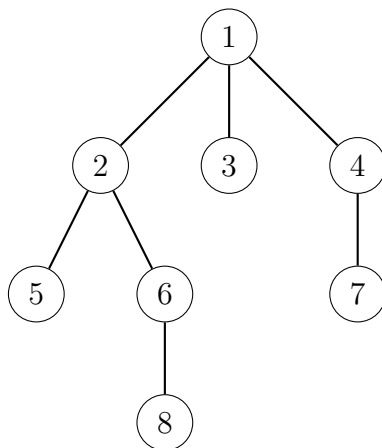
Tổ tiên chung thấp nhất

Cũng có một thuật toán offline để xử lý một tập truy vấn tổ tiên chung thấp nhất². Thuật toán dựa trên cấu trúc dữ liệu union-find (xem Chương 15.2), và lợi ích của nó là dễ cài đặt hơn các thuật toán đã thảo luận trước đó trong chương này.

Đầu vào của thuật toán là một tập các cặp đỉnh, và với mỗi cặp như vậy, thuật toán xác định tổ tiên chung thấp nhất của hai đỉnh. Thuật toán thực hiện duyệt cây theo chiều sâu và duy trì các tập rời nhau của các đỉnh. Ban đầu, mỗi đỉnh thuộc một tập riêng. Với mỗi tập, ta cũng lưu đỉnh cao nhất trong cây thuộc tập đó.

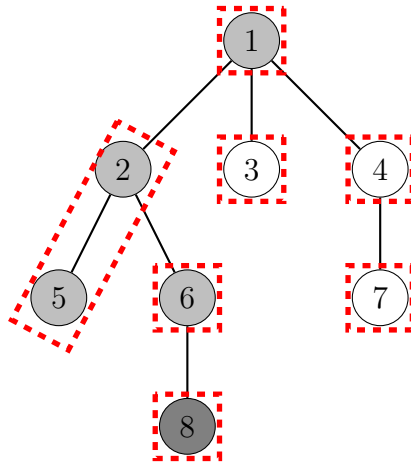
Khi thuật toán thăm một đỉnh x , nó duyệt qua mọi đỉnh y sao cho cần tìm tổ tiên chung thấp nhất của x và y . Nếu y đã được thăm, thuật toán báo rằng tổ tiên chung thấp nhất của x và y là đỉnh cao nhất trong tập của y . Sau đó, sau khi xử lý đỉnh x , thuật toán hợp nhất các tập của x và cha của nó.

Ví dụ, giả sử ta muốn tìm các tổ tiên chung thấp nhất của các cặp đỉnh $(5, 8)$ và $(2, 7)$ trong cây sau:

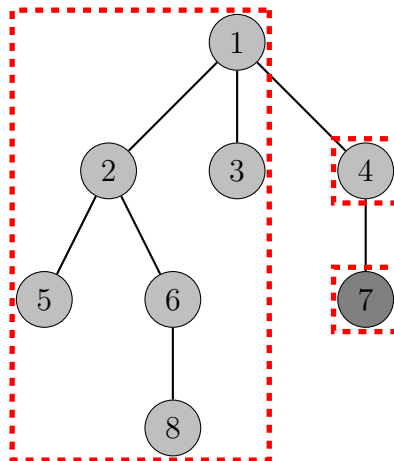


Trong các cây sau, các đỉnh màu xám biểu thị những đỉnh đã thăm và các nhóm nét đứt gồm các đỉnh thuộc cùng một tập. Khi thuật toán thăm đỉnh 8, nó nhận thấy rằng đỉnh 5 đã được thăm và đỉnh cao nhất trong tập của nó là 2. Do đó, tổ tiên chung thấp nhất của các đỉnh 5 và 8 là 2:

²Thuật toán này được R. E. Tarjan công bố năm 1979 [65].



Sau đó, khi thăm đỉnh 7, thuật toán xác định rằng tổ tiên chung thấp nhất của các đỉnh 2 và 7 là 1:



Chương 19

Đường đi và chu trình

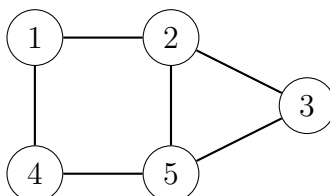
Chương này tập trung vào hai loại đường đi trong đồ thị:

- Một **đường đi Euler** (Eulerian path) là đường đi đi qua mỗi cạnh đúng một lần.
- Một **đường đi Hamilton** (Hamiltonian path) là đường đi thăm mỗi đỉnh đúng một lần.

Mặc dù thoát nhìn đường đi Euler và đường đi Hamilton có vẻ là các khái niệm tương tự, những bài toán tính toán liên quan đến chúng lại rất khác nhau. Hóa ra có một quy tắc đơn giản để xác định một đồ thị có chứa đường đi Euler hay không, và cũng có một thuật toán hiệu quả để tìm đường đi như vậy nếu nó tồn tại. Ngược lại, kiểm tra sự tồn tại của một đường đi Hamilton là một bài toán NP-khó, và hiện chưa biết thuật toán hiệu quả nào để giải bài toán này.

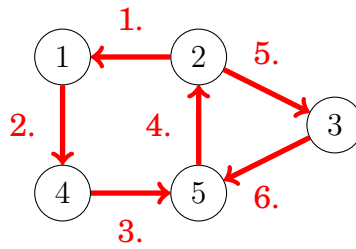
19.1 Đường đi Euler

Một **đường đi Euler**¹ là một đường đi đi qua mỗi cạnh của đồ thị đúng một lần. Ví dụ, đồ thị

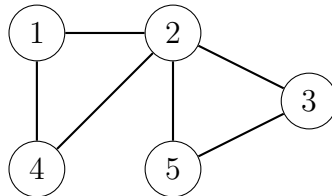


có một đường đi Euler từ đỉnh 2 đến đỉnh 5:

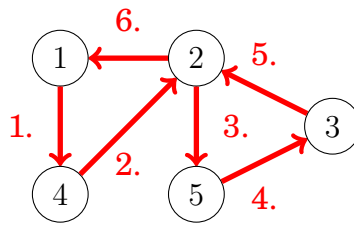
¹L. Euler nghiên cứu các đường đi như vậy vào năm 1736 khi ông giải bài toán bảy cây cầu Königsberg nổi tiếng. Đây là sự ra đời của lý thuyết đồ thị.



Một **chu trình Euler** (Eulerian circuit) là một đường đi Euler bắt đầu và kết thúc tại cùng một đỉnh. Ví dụ, đồ thị



có một chu trình Euler bắt đầu và kết thúc tại đỉnh 1:



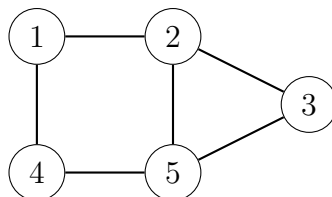
Sự tồn tại

Sự tồn tại của đường đi và chu trình Euler phụ thuộc vào bậc của các đỉnh. Trước hết, một đồ thị vô hướng có đường đi Euler khi và chỉ khi mọi cạnh thuộc cùng một thành phần liên thông và

- bậc của mỗi đỉnh là chẵn *hoặc*
- đúng hai đỉnh có bậc lẻ, và bậc của mọi đỉnh còn lại là chẵn.

Trong trường hợp thứ nhất, mọi đường đi Euler cũng là chu trình Euler. Trong trường hợp thứ hai, hai đỉnh bậc lẻ là đỉnh bắt đầu và đỉnh kết thúc của một đường đi Euler không phải là chu trình Euler.

Ví dụ, trong đồ thị



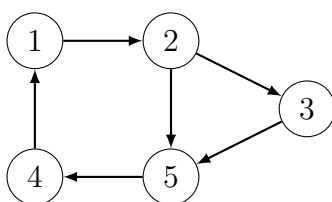
các đỉnh 1, 3 và 4 có bậc 2, còn các đỉnh 2 và 5 có bậc 3. Có đúng hai đỉnh có bậc lẻ, vì vậy tồn tại một đường đi Euler giữa các đỉnh 2 và 5, nhưng đồ thị không chứa chu trình Euler.

Trong đồ thị có hướng, ta xét bậc vào và bậc ra của các đỉnh. Một đồ thị có hướng chứa đường đi Euler khi và chỉ khi mọi cạnh thuộc cùng một thành phần liên thông và

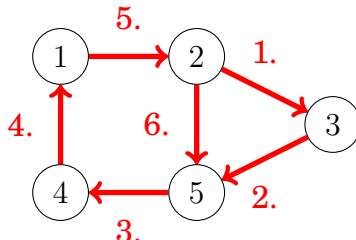
- tại mỗi đỉnh, bậc vào bằng bậc ra, *hoặc*
- tại một đỉnh, bậc vào lớn hơn bậc ra đúng một đơn vị, tại một đỉnh khác, bậc ra lớn hơn bậc vào đúng một đơn vị, và tại mọi đỉnh còn lại, bậc vào bằng bậc ra.

Trong trường hợp thứ nhất, mọi đường đi Euler cũng là chu trình Euler, còn trong trường hợp thứ hai, đồ thị chứa một đường đi Euler bắt đầu tại đỉnh có bậc ra lớn hơn và kết thúc tại đỉnh có bậc vào lớn hơn.

Ví dụ, trong đồ thị



các đỉnh 1, 3 và 4 đều có bậc vào 1 và bậc ra 1, đỉnh 2 có bậc vào 1 và bậc ra 2, còn đỉnh 5 có bậc vào 2 và bậc ra 1. Do đó, đồ thị chứa một đường đi Euler từ đỉnh 2 đến đỉnh 5:



Thuật toán Hierholzer

Thuật toán Hierholzer (Hierholzer's algorithm)² là một phương pháp hiệu quả để xây dựng một chu trình Euler. Thuật toán gồm nhiều vòng, mỗi vòng thêm các cạnh mới vào chu trình. Dĩ nhiên, ta giả sử đồ thị chứa một chu trình Euler; nếu không, thuật toán Hierholzer không thể tìm được nó.

Đầu tiên, thuật toán xây dựng một chu trình chứa một số cạnh (không nhất thiết là tất cả) của đồ thị. Sau đó, thuật toán mở rộng chu trình từng bước bằng cách thêm các chu trình con vào nó. Quá trình tiếp tục cho đến khi mọi cạnh đã được thêm vào chu trình.

Thuật toán mở rộng chu trình bằng cách luôn tìm một đỉnh x thuộc chu trình nhưng còn một cạnh đi ra chưa nằm trong chu trình. Thuật toán xây dựng một đường đi mới từ đỉnh x chỉ gồm các cạnh chưa có trong chu trình. Sớm hay muộn, đường đi sẽ quay lại đỉnh x , tạo thành một chu trình con.

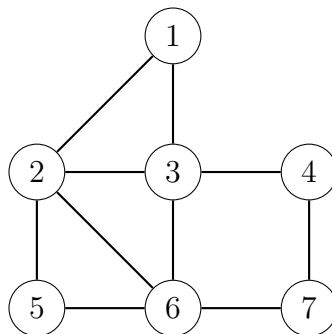
²Thuật toán được công bố năm 1873 sau khi Hierholzer qua đời [35].

Nếu đồ thị chỉ chứa một đường đi Euler, ta vẫn có thể dùng thuật toán Hierholzer để tìm nó bằng cách thêm một cạnh phụ vào đồ thị và xóa cạnh đó sau khi chu trình đã được xây dựng. Ví dụ, trong đồ thị vô hướng, ta thêm cạnh phụ giữa hai đỉnh có bậc lẻ.

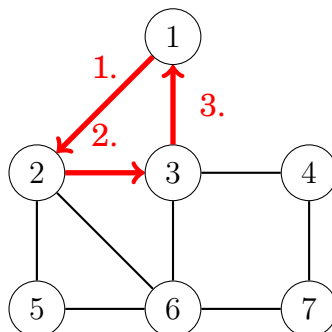
Tiếp theo ta sẽ xem thuật toán Hierholzer xây dựng một chu trình Euler cho đồ thị vô hướng như thế nào.

Ví dụ

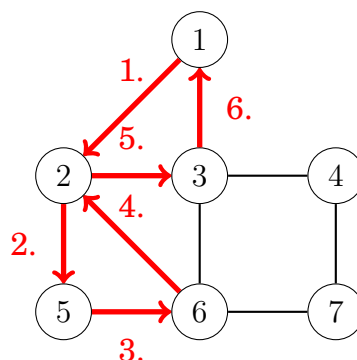
Hãy xét đồ thị sau:



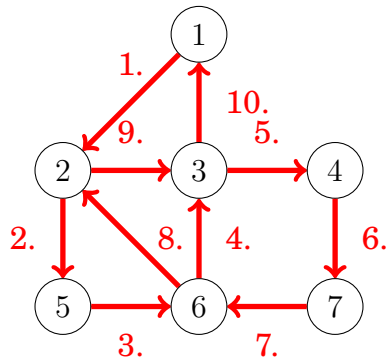
Giả sử thuật toán trước tiên tạo một chu trình bắt đầu tại đỉnh 1. Một chu trình có thể là $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



Sau đó, thuật toán thêm chu trình con $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ vào chu trình:



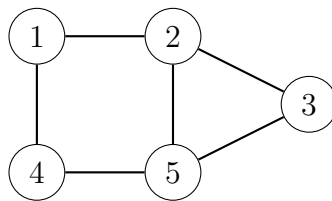
Cuối cùng, thuật toán thêm chu trình con $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ vào chu trình:



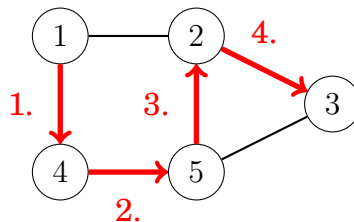
Bây giờ mọi cạnh đều đã nằm trong chu trình, vì vậy ta đã xây dựng thành công một chu trình Euler.

19.2 Đường đi Hamilton

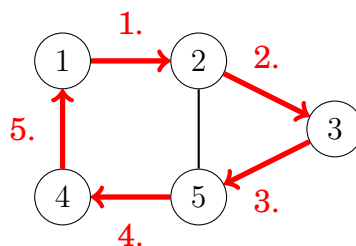
Một **đường đi Hamilton** là một đường đi thăm mỗi đỉnh của đồ thị đúng một lần. Ví dụ, đồ thị



chứa một đường đi Hamilton từ đỉnh 1 đến đỉnh 3:



Nếu một đường đi Hamilton bắt đầu và kết thúc tại cùng một đỉnh, nó được gọi là **chu trình Hamilton** (Hamiltonian circuit). Đồ thị trên cũng có một chu trình Hamilton bắt đầu và kết thúc tại đỉnh 1:



Sự tồn tại

Hiện chưa biết phương pháp hiệu quả nào để kiểm tra một đồ thị có chứa đường đi Hamilton hay không, và bài toán này là NP-khó. Tuy vậy, trong một số trường hợp đặc biệt, ta có thể chắc chắn rằng đồ thị chứa một đường đi Hamilton.

Một nhận xét đơn giản là nếu đồ thị đầy đủ, tức là có cạnh giữa mọi cặp đỉnh, thì nó cũng chứa một đường đi Hamilton. Ngoài ra còn có các kết quả mạnh hơn:

- **Định lý Dirac** (Dirac's theorem): Nếu bậc của mỗi đỉnh ít nhất là $n/2$, đồ thị chứa một đường đi Hamilton.
- **Định lý Ore** (Ore's theorem): Nếu tổng bậc của mỗi cặp đỉnh không kề nhau ít nhất là n , đồ thị chứa một đường đi Hamilton.

Một tính chất chung của các định lý này và các kết quả khác là chúng đảm bảo sự tồn tại của một đường đi Hamilton nếu đồ thị có *rất nhiều* cạnh. Điều này hợp lý, vì đồ thị càng có nhiều cạnh, càng có nhiều khả năng để xây dựng một đường đi Hamilton.

Xây dựng

Vì không có cách hiệu quả để kiểm tra liệu một đường đi Hamilton có tồn tại hay không, rõ ràng cũng không có phương pháp xây dựng đường đi đó một cách hiệu quả, bởi nếu có ta chỉ cần thử xây dựng đường đi và xem nó có tồn tại hay không.

Một cách đơn giản để tìm đường đi Hamilton là dùng thuật toán quay lui (backtracking) duyệt qua mọi cách có thể để xây dựng đường đi. Độ phức tạp thời gian của thuật toán như vậy ít nhất là $O(n!)$, vì có $n!$ cách khác nhau để chọn thứ tự của n đỉnh.

Một lời giải hiệu quả hơn dựa trên quy hoạch động (xem Chương 10.5). Ý tưởng là tính các giá trị của hàm $\text{possible}(S, x)$, trong đó S là một tập con các đỉnh và x là một trong các đỉnh. Hàm này cho biết có tồn tại một đường đi Hamilton thăm các đỉnh của S và kết thúc tại đỉnh x hay không. Có thể cài đặt lời giải này trong thời gian $O(2^n n^2)$.

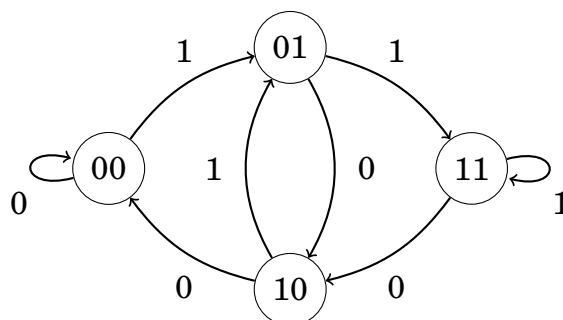
19.3 Dãy De Bruijn

Một **dãy De Bruijn** (De Bruijn sequence) là một xâu chứa mọi xâu độ dài n đúng một lần dưới dạng xâu con, với một bảng chữ cái cố định gồm k ký tự. Độ dài của xâu như vậy là $k^n + n - 1$ ký tự. Ví dụ, khi $n = 3$ và $k = 2$, một ví dụ về dãy De Bruijn là

0001011100.

Các xâu con của xâu này là tất cả các tổ hợp của ba bit: 000, 001, 010, 011, 100, 101, 110 và 111.

Hóa ra mỗi dãy De Bruijn tương ứng với một đường đi Euler trong một đồ thị. Ý tưởng là xây dựng một đồ thị trong đó mỗi đỉnh chứa một xâu gồm $n - 1$ ký tự và mỗi cạnh thêm một ký tự vào xâu. Đồ thị sau tương ứng với tình huống trên:



Một đường đi Euler trong đồ thị này tương ứng với một xâu chứa mọi xâu độ dài n . Xâu đó gồm các ký tự của đỉnh bắt đầu và tất cả ký tự trên các cạnh. Đỉnh bắt đầu có $n - 1$ ký tự và có k^n ký tự trên các cạnh, vì vậy độ dài của xâu là $k^n + n - 1$.

19.4 Hành trình quân mã

Một **hành trình quân mã** (knight's tour) là một chuỗi nước đi của quân mã trên bàn cờ $n \times n$ theo luật cờ vua sao cho quân mã thăm mỗi ô đúng một lần. Một hành trình quân mã được gọi là hành trình *đóng* nếu cuối cùng quân mã quay lại ô xuất phát, và ngược lại được gọi là hành trình *mở*.

Ví dụ, sau đây là một hành trình quân mã mở trên bàn 5×5 :

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Một hành trình quân mã tương ứng với một đường đi Hamilton trong đồ thị có các đỉnh biểu diễn các ô của bàn cờ, và hai đỉnh được nối bằng một cạnh nếu quân mã có thể di chuyển giữa hai ô đó theo luật cờ vua.

Một cách tự nhiên để xây dựng hành trình quân mã là dùng quay lui. Việc tìm kiếm có thể hiệu quả hơn nhờ dùng *heuristic* cố gắng dẫn quân mã sao cho một hành trình hoàn chỉnh được tìm thấy nhanh chóng.

Quy tắc Warnsdorf

Quy tắc Warnsdorf (Warnsdorf's rule) là một heuristic đơn giản và hiệu quả để tìm hành trình quân mã³. Với quy tắc này, có thể xây dựng hiệu quả một hành trình ngay cả trên bàn cờ lớn. Ý tưởng là luôn di chuyển quân mã sao cho nó đi đến một ô mà số nước đi tiếp theo có thể là *nhỏ nhất*.

Ví dụ, trong tình huống sau, có năm ô mà quân mã có thể di chuyển tới (các ô $a \dots e$):

1				a
		2		
b				e
	c		d	

Trong tình huống này, quy tắc Warnsdorf di chuyển quân mã tới ô a , vì sau lựa chọn này chỉ còn một nước đi có thể. Các lựa chọn khác sẽ đưa quân mã tới những ô mà sẽ còn ba nước đi khả dụng.

³Heuristic này được đề xuất trong sách của Warnsdorf [69] năm 1823. Cũng có các thuật toán đa thức để tìm hành trình quân mã [52], nhưng chúng phức tạp hơn.

Chương 20

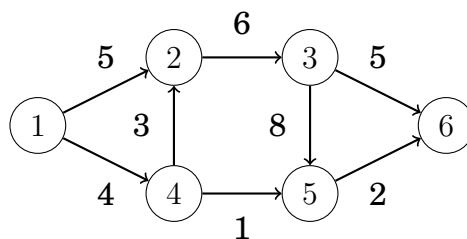
Luồng và lát cắt

Trong chương này, chúng ta tập trung vào hai bài toán sau:

- **Tìm luồng cực đại:** Lượng luồng lớn nhất mà ta có thể gửi từ một đỉnh đến một đỉnh khác là bao nhiêu?
- **Tìm lát cắt cực tiểu:** Tập cạnh có tổng trọng số nhỏ nhất nào tách hai đỉnh của đồ thị?

Đầu vào của cả hai bài toán là một đồ thị có hướng, có trọng số và chứa hai đỉnh đặc biệt: *nguồn* là đỉnh không có cạnh đi vào, và *đích* là đỉnh không có cạnh đi ra.

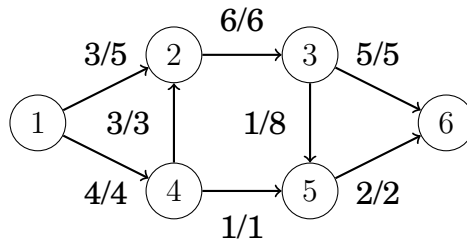
Làm ví dụ, ta sẽ dùng đồ thị sau, trong đó đỉnh 1 là nguồn và đỉnh 6 là đích:



Luồng cực đại

Trong bài toán **luồng cực đại**, nhiệm vụ của ta là gửi càng nhiều luồng càng tốt từ nguồn đến đích. Trọng số của mỗi cạnh là một sức chứa, giới hạn lượng luồng có thể đi qua cạnh đó. Tại mỗi đỉnh trung gian, lượng luồng đi vào và đi ra phải bằng nhau.

Ví dụ, giá trị lớn nhất của một luồng trong đồ thị ví dụ là 7. Hình sau cho thấy ta có thể định tuyến luồng như thế nào:

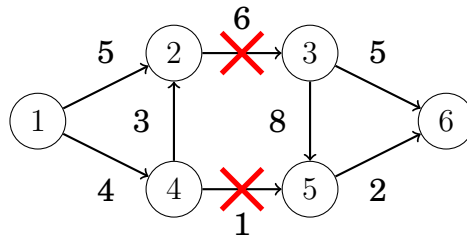


Ký hiệu v/k nghĩa là có v đơn vị luồng được định tuyến qua một cạnh có sức chứa k đơn vị. Giá trị của luồng là 7, vì nguồn gửi $3 + 4$ đơn vị luồng và đích nhận $5 + 2$ đơn vị luồng. Dễ thấy rằng luồng này là cực đại, vì tổng sức chứa của các cạnh đi vào đích là 7.

Lát cắt cực tiểu

Trong bài toán **lát cắt cực tiểu**, nhiệm vụ của ta là xóa một tập cạnh khỏi đồ thị sao cho sau khi xóa sẽ không còn đường đi từ nguồn đến đích và tổng trọng số của các cạnh bị xóa là nhỏ nhất.

Kích thước nhỏ nhất của một lát cắt trong đồ thị ví dụ là 7. Chỉ cần xóa các cạnh $2 \rightarrow 3$ và $4 \rightarrow 5$:



Sau khi xóa các cạnh, sẽ không còn đường đi từ nguồn đến đích. Kích thước của lát cắt là 7, vì trọng số của các cạnh bị xóa là 6 và 1. Lát cắt này là cực tiểu, vì không có cách hợp lệ nào để xóa các cạnh khỏi đồ thị sao cho tổng trọng số của chúng nhỏ hơn 7.

Không phải ngẫu nhiên mà giá trị cực đại của một luồng và kích thước cực tiểu của một lát cắt lại bằng nhau trong ví dụ trên. Hóa ra luồng cực đại và lát cắt cực tiểu *luôn* có cùng kích thước, nên hai khái niệm này là hai mặt của cùng một vấn đề.

Tiếp theo ta sẽ thảo luận thuật toán Ford–Fulkerson, có thể dùng để tìm luồng cực đại và lát cắt cực tiểu của một đồ thị. Thuật toán cũng giúp ta hiểu *vì sao* chúng có cùng kích thước.

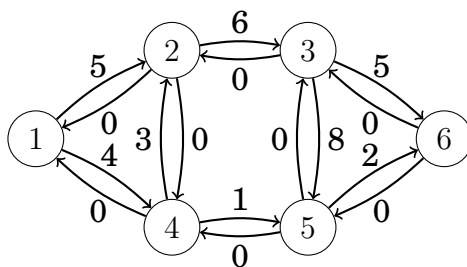
20.1 Thuật toán Ford–Fulkerson

Thuật toán **Ford–Fulkerson** [25] tìm luồng cực đại trong một đồ thị. Thuật toán bắt đầu với một luồng rỗng, và ở mỗi bước tìm một đường đi từ nguồn

đến đích để tạo thêm luồng. Cuối cùng, khi thuật toán không thể tăng luồng thêm nữa, ta đã tìm được luồng cực đại.

Thuật toán dùng một cách biểu diễn đặc biệt của đồ thị, trong đó mỗi cạnh ban đầu có một cạnh ngược theo hướng còn lại. Trọng số của mỗi cạnh cho biết ta còn có thể định tuyến thêm bao nhiêu luồng qua cạnh đó. Ở đầu thuật toán, trọng số của mỗi cạnh ban đầu bằng sức chứa của cạnh và trọng số của mỗi cạnh ngược bằng không.

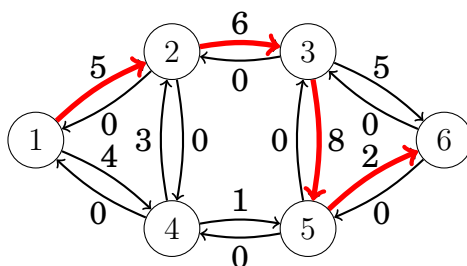
Cách biểu diễn mới của đồ thị ví dụ như sau:



Mô tả thuật toán

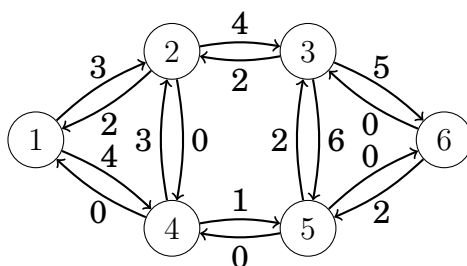
Thuật toán Ford–Fulkerson gồm nhiều vòng. Ở mỗi vòng, thuật toán tìm một đường đi từ nguồn đến đích sao cho mỗi cạnh trên đường đi có trọng số dương. Nếu có nhiều đường đi khả dĩ, ta có thể chọn bất kỳ đường nào.

Ví dụ, giả sử ta chọn đường đi sau:



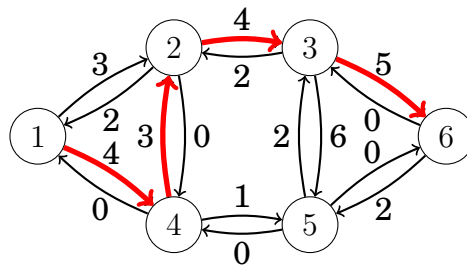
Sau khi chọn đường đi, luồng tăng thêm x đơn vị, trong đó x là trọng số cạnh nhỏ nhất trên đường đi. Ngoài ra, trọng số của mỗi cạnh trên đường đi giảm đi x và trọng số của mỗi cạnh ngược tăng thêm x .

Trên đường đi trên, trọng số của các cạnh là 5, 6, 8 và 2. Trọng số nhỏ nhất là 2, nên luồng tăng thêm 2 và đồ thị mới như sau:



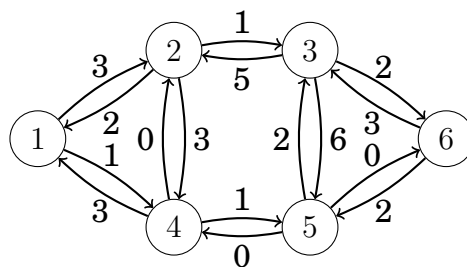
Ý tưởng là việc tăng luồng sẽ làm giảm lượng luồng có thể đi qua các cạnh trong tương lai. Mặt khác, ta có thể hủy luồng về sau bằng các cạnh ngược của đồ thị nếu hóa ra định tuyến luồng theo cách khác sẽ có lợi hơn.

Thuật toán tăng luồng miễn là còn một đường đi từ nguồn đến đích qua các cạnh có trọng số dương. Trong ví dụ hiện tại, đường đi tiếp theo có thể là:

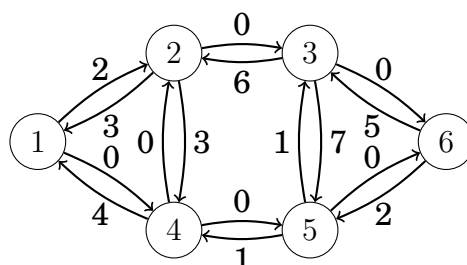


Trọng số cạnh nhỏ nhất trên đường đi này là 3, nên đường đi làm luồng tăng thêm 3, và tổng luồng sau khi xử lý đường đi là 5.

Đồ thị mới sẽ như sau:



Ta vẫn cần thêm hai vòng nữa trước khi đạt luồng cực đại. Ví dụ, ta có thể chọn các đường đi $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ và $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$. Cả hai đường đi đều làm luồng tăng thêm 1, và đồ thị cuối cùng như sau:



Không thể tăng luồng thêm nữa, vì không có đường đi từ nguồn đến đích với các cạnh có trọng số dương. Do đó, thuật toán kết thúc và luồng cực đại là 7.

Tìm đường đi

Thuật toán Ford–Fulkerson không quy định ta nên chọn các đường đi tăng luồng như thế nào. Trong mọi trường hợp, thuật toán sớm muộn cũng kết thúc và tìm đúng luồng cực đại. Tuy nhiên, hiệu quả của thuật toán phụ thuộc vào cách chọn các đường đi.

Một cách đơn giản để tìm đường đi là dùng tìm kiếm theo chiều sâu. Thông thường cách này hoạt động tốt, nhưng trong trường hợp xấu nhất, mỗi đường đi chỉ làm luồng tăng thêm 1 và thuật toán sẽ chậm. May mắn là ta có thể tránh tình huống này bằng một trong các kỹ thuật sau:

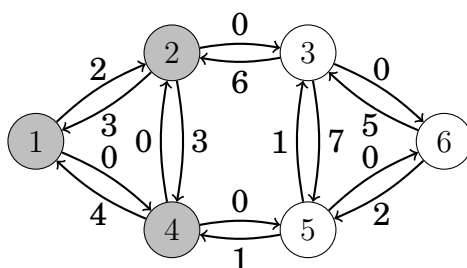
Thuật toán **Edmonds–Karp** [18] chọn mỗi đường đi sao cho số cạnh trên đường đi là nhỏ nhất có thể. Điều này có thể thực hiện bằng tìm kiếm theo chiều rộng thay vì tìm kiếm theo chiều sâu để tìm đường đi. Có thể chứng minh rằng cách này đảm bảo luồng tăng nhanh, và độ phức tạp thời gian của thuật toán là $O(m^2n)$.

Thuật toán **scaling** [2] dùng tìm kiếm theo chiều sâu để tìm các đường đi mà mỗi cạnh có trọng số ít nhất bằng một giá trị ngưỡng. Ban đầu, giá trị ngưỡng là một số lớn, chẳng hạn tổng tất cả trọng số cạnh của đồ thị. Mỗi khi không tìm được đường đi, giá trị ngưỡng được chia cho 2. Độ phức tạp thời gian của thuật toán là $O(m^2 \log c)$, trong đó c là giá trị ngưỡng ban đầu.

Trong thực tế, thuật toán scaling dễ cài đặt hơn, vì có thể dùng tìm kiếm theo chiều sâu để tìm đường đi. Cả hai thuật toán đều đủ hiệu quả cho các bài toán thường xuất hiện trong lập trình thi đấu.

Lát cắt cực tiểu

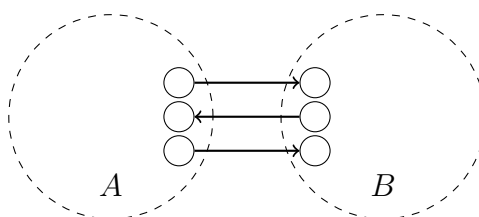
Hóa ra một khi thuật toán Ford–Fulkerson đã tìm được luồng cực đại, nó cũng đã xác định một lát cắt cực tiểu. Gọi A là tập các đỉnh có thể đến được từ nguồn bằng các cạnh có trọng số dương. Trong đồ thị ví dụ, A chứa các đỉnh 1, 2 và 4:



Khi đó lát cắt cực tiểu gồm các cạnh của đồ thị ban đầu bắt đầu ở một đỉnh trong A , kết thúc ở một đỉnh ngoài A , và có sức chứa đã được dùng hết trong luồng cực đại. Trong đồ thị trên, các cạnh như vậy là $2 \rightarrow 3$ và $4 \rightarrow 5$, tương ứng với lát cắt cực tiểu $6 + 1 = 7$.

Vì sao luồng do thuật toán tạo ra là cực đại và vì sao lát cắt là cực tiểu? Lý do là một đồ thị không thể chứa một luồng có giá trị lớn hơn trọng số của bất kỳ lát cắt nào của đồ thị. Do đó, mỗi khi một luồng và một lát cắt có cùng kích thước, chúng là luồng cực đại và lát cắt cực tiểu.

Hãy xét một lát cắt bất kỳ của đồ thị sao cho nguồn thuộc A , đích thuộc B và có một số cạnh giữa hai tập:



Kích thước của lát cắt là tổng trọng số các cạnh đi từ A sang B . Đây là một cận trên cho luồng trong đồ thị, vì luồng phải đi từ A sang B . Vì vậy, giá trị của luồng cực đại nhỏ hơn hoặc bằng kích thước của mọi lát cắt trong đồ thị.

Mặt khác, thuật toán Ford–Fulkerson tạo ra một luồng có giá trị *đúng bằng* kích thước của một lát cắt trong đồ thị. Vì vậy, luồng đó phải là luồng cực đại và lát cắt đó phải là lát cắt cực tiểu.

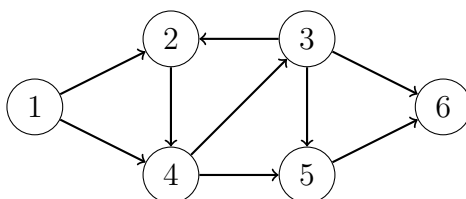
20.2 Các đường đi rời nhau

Nhiều bài toán đồ thị có thể được giải bằng cách quy về bài toán luồng cực đại. Ví dụ đầu tiên của ta về một bài toán như vậy như sau: cho một đồ thị có hướng với một nguồn và một đích, nhiệm vụ là tìm số lượng lớn nhất các đường đi rời nhau từ nguồn đến đích.

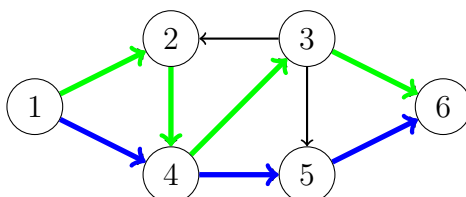
Đường đi rời cạnh

Trước hết ta sẽ tập trung vào bài toán tìm số lượng lớn nhất các **đường đi rời cạnh** từ nguồn đến đích. Điều này nghĩa là ta cần xây dựng một tập đường đi sao cho mỗi cạnh xuất hiện trong nhiều nhất một đường đi.

Ví dụ, xét đồ thị sau:



Trong đồ thị này, số lượng lớn nhất các đường đi rời cạnh là 2. Ta có thể chọn các đường đi $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ và $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ như sau:

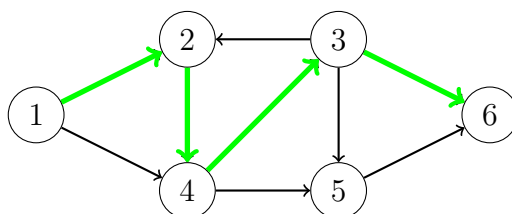


Hóa ra số lượng lớn nhất các đường đi rời cạnh bằng luồng cực đại của đồ thị, giả sử sức chứa của mỗi cạnh là một. Sau khi đã xây dựng luồng cực đại, có thể tìm các đường đi rời cạnh một cách tham lam bằng cách lần theo các đường đi từ nguồn đến đích.

Đường đi rời đỉnh

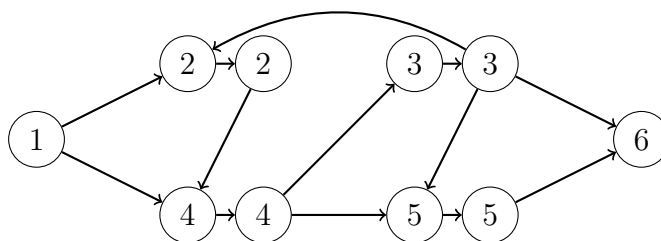
Bây giờ hãy xét một bài toán khác: tìm số lượng lớn nhất các **đường đi rời đỉnh** từ nguồn đến đích. Trong bài toán này, mọi đỉnh, ngoại trừ nguồn và đích, được xuất hiện trong nhiều nhất một đường đi. Số lượng đường đi rời đỉnh có thể nhỏ hơn số lượng đường đi rời cạnh.

Ví dụ, trong đồ thị trước, số lượng lớn nhất các đường đi rời đỉnh là 1:

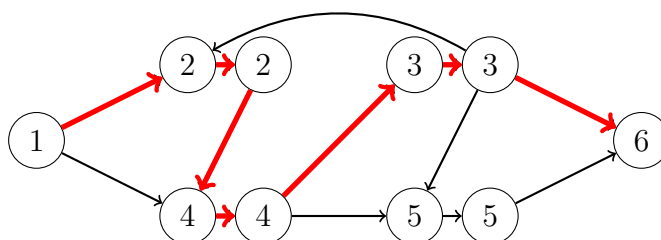


Ta cũng có thể quy bài toán này về bài toán luồng cực đại. Vì mỗi đỉnh chỉ được xuất hiện trong nhiều nhất một đường đi, ta phải giới hạn luồng đi qua các đỉnh. Một phương pháp chuẩn cho việc này là tách mỗi đỉnh thành hai đỉnh sao cho đỉnh thứ nhất nhận các cạnh đi vào của đỉnh ban đầu, đỉnh thứ hai nhận các cạnh đi ra của đỉnh ban đầu, và có một cạnh mới từ đỉnh thứ nhất đến đỉnh thứ hai.

Trong ví dụ của ta, đồ thị trở thành như sau:



Luồng cực đại của đồ thị như sau:



Do đó, số lượng lớn nhất các đường đi rời đỉnh từ nguồn đến đích là 1.

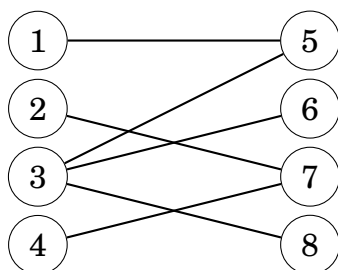
20.3 Ghép cặp cực đại

Bài toán **ghép cặp cực đại** yêu cầu tìm một tập cặp đỉnh có kích thước lớn nhất trong một đồ thị vô hướng sao cho mỗi cặp được nối bằng một cạnh và mỗi đỉnh thuộc nhiều nhất một cặp.

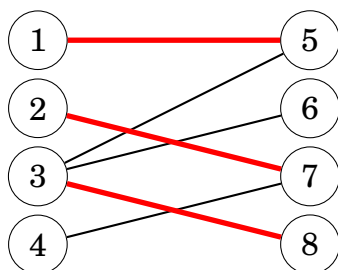
Có các thuật toán đa thức để tìm ghép cặp cực đại trong đồ thị tổng quát [17], nhưng các thuật toán như vậy phức tạp và hiếm khi xuất hiện trong lập trình thi đấu. Tuy nhiên, trong đồ thị hai phía, bài toán ghép cặp cực đại dễ giải hơn nhiều, vì ta có thể quy nó về bài toán luồng cực đại.

Tìm ghép cặp cực đại

Các đỉnh của một đồ thị hai phía luôn có thể được chia thành hai nhóm sao cho mọi cạnh của đồ thị đi từ nhóm bên trái sang nhóm bên phải. Ví dụ, trong đồ thị hai phía sau, hai nhóm là $\{1, 2, 3, 4\}$ và $\{5, 6, 7, 8\}$.

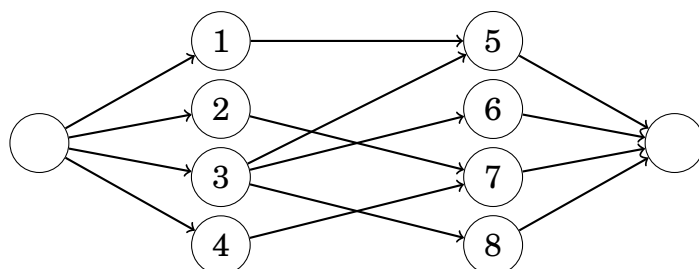


Kích thước của một ghép cặp cực đại trong đồ thị này là 3:

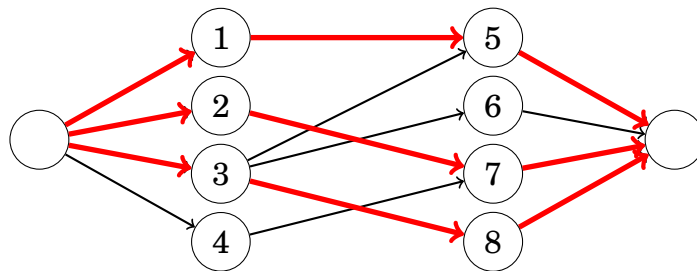


Ta có thể quy bài toán ghép cặp cực đại trong đồ thị hai phía về bài toán luồng cực đại bằng cách thêm hai đỉnh mới vào đồ thị: một nguồn và một đích. Ta cũng thêm các cạnh từ nguồn đến mỗi đỉnh bên trái và từ mỗi đỉnh bên phải đến đích. Sau đó, kích thước của luồng cực đại trong đồ thị bằng kích thước của ghép cặp cực đại trong đồ thị ban đầu.

Ví dụ, phép quy về cho đồ thị trên như sau:



Luồng cực đại của đồ thị này như sau:

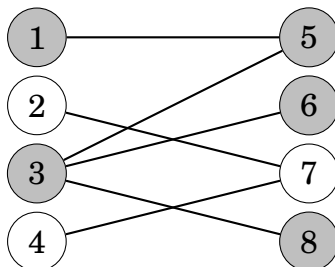


Định lý Hall

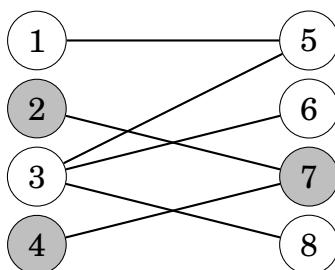
Định lý Hall có thể được dùng để xác định một đồ thị hai phía có ghép cặp chứa tất cả các đỉnh bên trái hoặc bên phải hay không. Nếu số đỉnh bên trái và bên phải bằng nhau, định lý Hall cho ta biết liệu có thể xây dựng một **ghép cặp hoàn hảo** chứa tất cả các đỉnh của đồ thị hay không.

Giả sử ta muốn tìm một ghép cặp chứa tất cả các đỉnh bên trái. Gọi X là một tập bất kỳ các đỉnh bên trái và gọi $f(X)$ là tập các đỉnh kề của chúng. Theo định lý Hall, một ghép cặp chứa tất cả các đỉnh bên trái tồn tại khi và chỉ khi với mọi X , điều kiện $|X| \leq |f(X)|$ đúng.

Hãy xét định lý Hall trong đồ thị ví dụ. Trước hết, đặt $X = \{1, 3\}$, khi đó $f(X) = \{5, 6, 8\}$:



Điều kiện của định lý Hall đúng, vì $|X| = 2$ và $|f(X)| = 3$. Tiếp theo, đặt $X = \{2, 4\}$, khi đó $f(X) = \{7\}$:



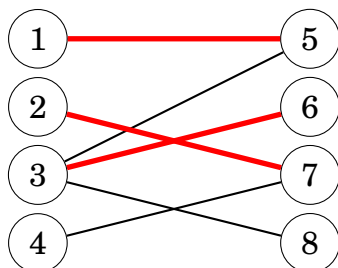
Trong trường hợp này, $|X| = 2$ và $|f(X)| = 1$, nên điều kiện của định lý Hall không đúng. Điều này nghĩa là không thể tạo một ghép cặp hoàn hảo cho đồ thị. Kết quả này không bất ngờ, vì ta đã biết rằng ghép cặp cực đại của đồ thị có kích thước 3 chứ không phải 4.

Nếu điều kiện của định lý Hall không đúng, tập X cung cấp một lời giải thích *vì sao* ta không thể tạo một ghép cặp như vậy. Vì X chứa nhiều đỉnh hơn $f(X)$, không có đủ cặp cho tất cả các đỉnh trong X . Ví dụ, trong đồ thị trên, cả hai đỉnh 2 và 4 đều cần được ghép với đỉnh 7, điều này là không thể.

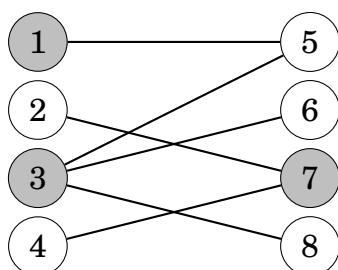
Định lý König

Phủ đỉnh cực tiểu của một đồ thị là một tập đỉnh nhỏ nhất sao cho mỗi cạnh của đồ thị có ít nhất một đầu mút trong tập đó. Trong đồ thị tổng quát, tìm phủ đỉnh cực tiểu là một bài toán NP-khó. Tuy nhiên, nếu đồ thị là hai phía, **định lý König** cho biết rằng kích thước của một phủ đỉnh cực tiểu và kích thước của một ghép cặp cực đại luôn bằng nhau. Vì vậy, ta có thể tính kích thước của phủ đỉnh cực tiểu bằng thuật toán luồng cực đại.

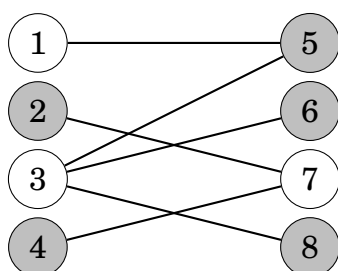
Hãy xét đồ thị sau với một ghép cặp cực đại có kích thước 3:



Bây giờ định lý König cho ta biết rằng kích thước của một phủ đỉnh cực tiểu cũng là 3. Một phủ như vậy có thể được xây dựng như sau:



Các đỉnh *không* thuộc một phủ đỉnh cực tiểu tạo thành một **tập độc lập cực đại**. Đây là tập đỉnh lớn nhất có thể sao cho không có hai đỉnh nào trong tập được nối bằng một cạnh. Một lần nữa, tìm tập độc lập cực đại trong đồ thị tổng quát là một bài toán NP-khó, nhưng trong đồ thị hai phía ta có thể dùng định lý König để giải bài toán hiệu quả. Trong đồ thị ví dụ, tập độc lập cực đại như sau:

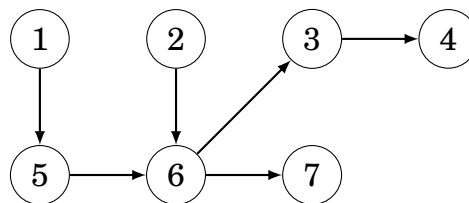


20.4 Phủ đường đi

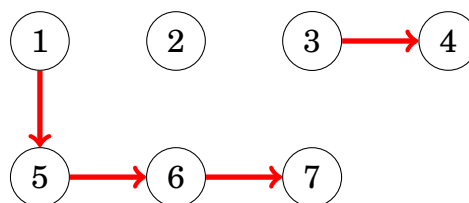
Phủ đường đi là một tập các đường đi trong đồ thị sao cho mỗi đỉnh của đồ thị thuộc ít nhất một đường đi. Hóa ra trong các đồ thị có hướng không chu trình, ta có thể quy bài toán tìm một phủ đường đi cực tiểu về bài toán tìm một luồng cực đại trong một đồ thị khác.

Phủ đường đi rời đỉnh

Trong một **phủ đường đi rời đỉnh**, mỗi đỉnh thuộc đúng một đường đi. Làm ví dụ, xét đồ thị sau:



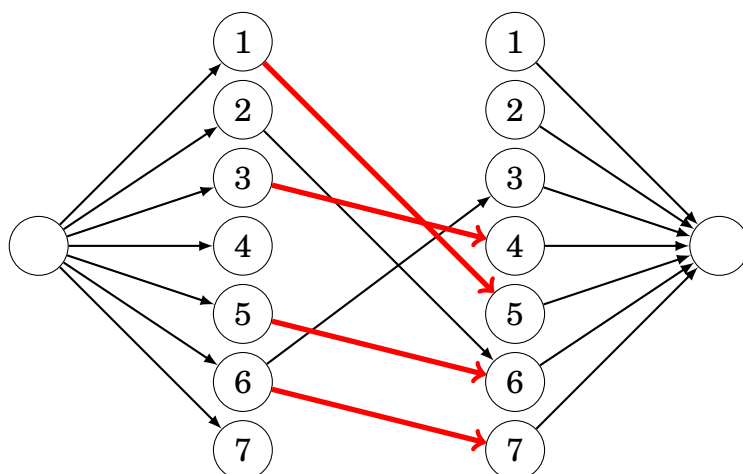
Một phủ đường đi rời đỉnh cực tiểu của đồ thị này gồm ba đường đi. Ví dụ, ta có thể chọn các đường đi sau:



Lưu ý rằng một trong các đường đi chỉ chứa đỉnh 2, nên một đường đi có thể không chứa cạnh nào.

Ta có thể tìm một phủ đường đi rời đỉnh cực tiểu bằng cách xây dựng một *đồ thị ghép cặp*, trong đó mỗi đỉnh của đồ thị ban đầu được biểu diễn bằng hai đỉnh: một đỉnh bên trái và một đỉnh bên phải. Có một cạnh từ một đỉnh bên trái đến một đỉnh bên phải nếu trong đồ thị ban đầu có cạnh tương ứng. Ngoài ra, đồ thị ghép cặp chứa một nguồn và một đích, và có các cạnh từ nguồn đến tất cả các đỉnh bên trái và từ tất cả các đỉnh bên phải đến đích.

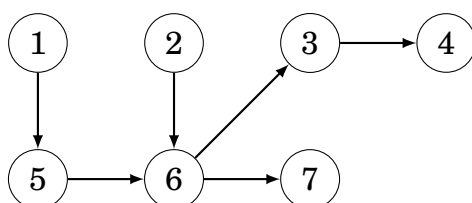
Một ghép cặp cực đại trong đồ thị thu được tương ứng với một phủ đường đi rời đỉnh cực tiểu trong đồ thị ban đầu. Ví dụ, đồ thị ghép cặp sau cho đồ thị trên chứa một ghép cặp cực đại có kích thước 4:



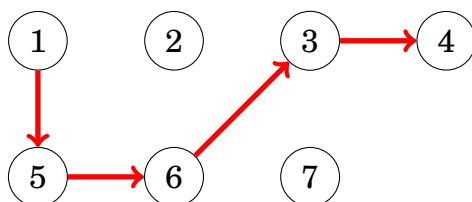
Mỗi cạnh trong ghép cặp cực đại của đồ thị ghép cặp tương ứng với một cạnh trong phủ đường đi rời đỉnh cực tiểu của đồ thị ban đầu. Vì vậy, kích thước của phủ đường đi rời đỉnh cực tiểu là $n - c$, trong đó n là số đỉnh trong đồ thị ban đầu và c là kích thước của ghép cặp cực đại.

Phủ đường đi tổng quát

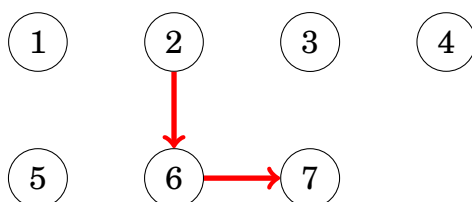
Phủ đường đi tổng quát là một phủ đường đi trong đó một đỉnh có thể thuộc nhiều hơn một đường đi. Một phủ đường đi tổng quát cực tiểu có thể nhỏ hơn một phủ đường đi rời đỉnh cực tiểu, vì một đỉnh có thể được dùng nhiều lần trong các đường đi. Hãy xét lại đồ thị sau:



Phủ đường đi tổng quát cực tiểu của đồ thị này gồm hai đường đi. Ví dụ, đường đi thứ nhất có thể như sau:

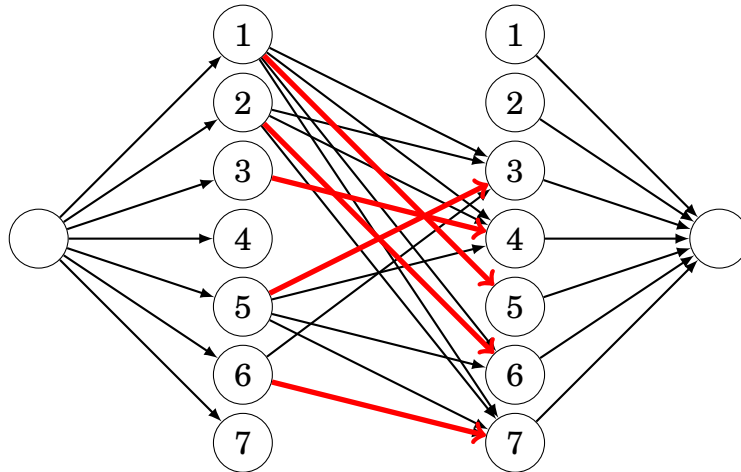


Và đường đi thứ hai có thể như sau:



Có thể tìm phủ đường đi tổng quát cực tiểu gần giống như phủ đường đi rời đỉnh cực tiểu. Chỉ cần thêm một số cạnh mới vào đồ thị ghép cặp sao cho có cạnh $a \rightarrow b$ mỗi khi có một đường đi từ a đến b trong đồ thị ban đầu (có thể đi qua nhiều cạnh).

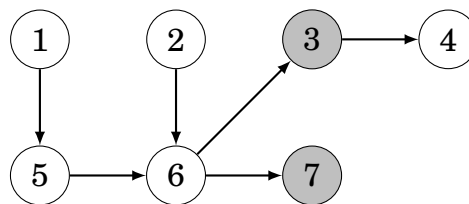
Đồ thị ghép cặp cho đồ thị trên như sau:



Định lý Dilworth

Phản xích là một tập đỉnh của đồ thị sao cho không có đường đi từ bất kỳ đỉnh nào đến một đỉnh khác bằng các cạnh của đồ thị. **Định lý Dilworth** phát biểu rằng trong một đồ thị có hướng không chu trình, kích thước của một phủ đường đi tổng quát cực tiểu bằng kích thước của một phản xích cực đại.

Ví dụ, các đỉnh 3 và 7 tạo thành một phản xích trong đồ thị sau:



Đây là một phản xích cực đại, vì không thể xây dựng phản xích nào chứa ba đỉnh. Trước đó ta đã thấy rằng kích thước của phủ đường đi tổng quát cực tiểu của đồ thị này gồm hai đường đi.

Phần III

Chủ đề nâng cao

Chương 21

Lý thuyết số

Lý thuyết số là một nhánh của toán học nghiên cứu các số nguyên. Lý thuyết số là một lĩnh vực hấp dẫn, vì nhiều câu hỏi liên quan đến số nguyên rất khó giải dù ban đầu trông có vẻ đơn giản.

Ví dụ, xét phương trình sau:

$$x^3 + y^3 + z^3 = 33$$

Để tìm ba số thực x, y và z thỏa mãn phương trình. Ví dụ, ta có thể chọn

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

Tuy nhiên, trong lý thuyết số, việc có tồn tại ba số *nguyên* x, y và z thỏa mãn phương trình hay không vẫn là một bài toán mở [6].

Trong chương này, ta sẽ tập trung vào các khái niệm và thuật toán cơ bản trong lý thuyết số. Trong toàn bộ chương, ta sẽ giả sử rằng mọi số đều là số nguyên, trừ khi có nói khác đi.

21.1 Số nguyên tố và ước

Một số a được gọi là **ước** của số b nếu a chia hết b . Nếu a là ước của b , ta viết $a \mid b$, ngược lại ta viết $a \nmid b$. Ví dụ, các ước của 24 là 1, 2, 3, 4, 6, 8, 12 và 24.

Một số $n > 1$ là **số nguyên tố** nếu các ước dương duy nhất của nó là 1 và n . Ví dụ, 7, 19 và 41 là các số nguyên tố, nhưng 35 không phải số nguyên tố, vì $5 \cdot 7 = 35$. Với mọi số $n > 1$, tồn tại duy nhất một **phân tích thừa số nguyên tố**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

trong đó $p_1, p_2, \dots, p_k$ là các số nguyên tố phân biệt và $\alpha_1, \alpha_2, \dots, \alpha_k$ là các số dương. Ví dụ, phân tích thừa số nguyên tố của 84 là

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Số lượng ước của một số n là

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

vì với mỗi số nguyên tố p_i , có $\alpha_i + 1$ cách chọn số lần nó xuất hiện trong một ước. Ví dụ, số lượng ước của 84 là $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. Các ước là 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 và 84.

Tổng các ước của n là

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

trong đó công thức sau dựa trên công thức cấp số nhân. Ví dụ, tổng các ước của 84 là

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

Tích các ước của n là

$$\mu(n) = n^{\tau(n)/2},$$

vì ta có thể tạo $\tau(n)/2$ cặp từ các ước, mỗi cặp có tích bằng n . Ví dụ, các ước của 84 tạo ra các cặp $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$, v.v., và tích các ước là $\mu(84) = 84^6 = 351298031616$.

Một số n được gọi là **số hoàn hảo** nếu $n = \sigma(n) - n$, tức là n bằng tổng các ước của nó nằm giữa 1 và $n - 1$. Ví dụ, 28 là một số hoàn hảo, vì $28 = 1 + 2 + 4 + 7 + 14$.

Số lượng số nguyên tố

Dễ chứng minh rằng có vô hạn số nguyên tố. Nếu số lượng số nguyên tố là hữu hạn, ta có thể xây dựng một tập $P = \{p_1, p_2, \dots, p_n\}$ chứa tất cả các số nguyên tố. Ví dụ, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, v.v. Tuy nhiên, dùng P , ta có thể tạo một số nguyên tố mới

$$p_1 p_2 \cdots p_n + 1$$

lớn hơn mọi phần tử trong P . Đây là một mâu thuẫn, nên số lượng số nguyên tố phải là vô hạn.

Mật độ số nguyên tố

Mật độ số nguyên tố cho biết các số nguyên tố xuất hiện thường xuyên như thế nào trong các số. Gọi $\pi(n)$ là số lượng số nguyên tố giữa 1 và n . Ví dụ, $\pi(10) = 4$, vì có 4 số nguyên tố giữa 1 và 10: 2, 3, 5 và 7.

Có thể chứng minh rằng

$$\pi(n) \approx \frac{n}{\ln n},$$

điều này nghĩa là số nguyên tố xuất hiện khá thường xuyên. Ví dụ, số lượng số nguyên tố giữa 1 và 10^6 là $\pi(10^6) = 78498$, và $10^6 / \ln 10^6 \approx 72382$.

Các giả thuyết

Có nhiều *giả thuyết* liên quan đến số nguyên tố. Phần lớn mọi người cho rằng các giả thuyết này đúng, nhưng chưa ai chứng minh được chúng. Ví dụ, các giả thuyết sau rất nổi tiếng:

- **Giả thuyết Goldbach:** Mỗi số nguyên chẵn $n > 2$ có thể được biểu diễn dưới dạng tổng $n = a + b$ sao cho cả a và b đều là số nguyên tố.
- **Giả thuyết số nguyên tố sinh đôi:** Có vô hạn cặp dạng $\{p, p + 2\}$, trong đó cả p và $p + 2$ đều là số nguyên tố.
- **Giả thuyết Legendre:** Luôn có một số nguyên tố giữa các số n^2 và $(n + 1)^2$, trong đó n là một số nguyên dương bất kỳ.

Các thuật toán cơ bản

Nếu một số n không phải số nguyên tố, nó có thể được biểu diễn dưới dạng tích $a \cdot b$, trong đó $a \leq \sqrt{n}$ hoặc $b \leq \sqrt{n}$, nên chắc chắn nó có một ước giữa 2 và $\lfloor \sqrt{n} \rfloor$. Dùng nhận xét này, ta vừa có thể kiểm tra một số có phải số nguyên tố hay không, vừa tìm được phân tích thừa số nguyên tố của một số trong thời gian $O(\sqrt{n})$.

Hàm prime sau kiểm tra số n cho trước có phải số nguyên tố hay không. Hàm thử chia n cho tất cả các số giữa 2 và $\lfloor \sqrt{n} \rfloor$, và nếu không số nào chia hết n , thì n là số nguyên tố.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

Hàm factors sau xây dựng một vector chứa phân tích thừa số nguyên tố của n . Hàm chia n cho các thừa số nguyên tố của nó, và thêm chúng vào vector. Quá trình kết thúc khi số n còn lại không có ước nào giữa 2 và $\lfloor \sqrt{n} \rfloor$. Nếu $n > 1$, nó là số nguyên tố và là thừa số cuối cùng.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

}

Lưu ý rằng mỗi thừa số nguyên tố xuất hiện trong vector đúng số lần nó chia số đó. Ví dụ, $24 = 2^3 \cdot 3$, nên kết quả của hàm là $[2, 2, 2, 3]$.

Sàng Eratosthenes

Sàng Eratosthenes thực hiện tiền xử lý bằng cách xây dựng một mảng để ta có thể kiểm tra hiệu quả một số cho trước trong khoảng $2 \dots n$ có phải số nguyên tố hay không, và nếu không, tìm một thừa số nguyên tố của số đó.

Thuật toán xây dựng một mảng sieve sử dụng các vị trí $2, 3, \dots, n$. Giá trị $\text{sieve}[k] = 0$ nghĩa là k là số nguyên tố, và giá trị $\text{sieve}[k] \neq 0$ nghĩa là k không phải số nguyên tố và một thừa số nguyên tố của nó là $\text{sieve}[k]$.

Thuật toán duyệt lần lượt các số $2 \dots n$. Mỗi khi tìm thấy một số nguyên tố mới x , thuật toán ghi nhận rằng các bội của x ($2x, 3x, 4x, \dots$) không phải số nguyên tố, vì số x chia hết chúng.

Ví dụ, nếu $n = 20$, mảng như sau:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Đoạn mã sau cài đặt sàng Eratosthenes. Đoạn mã giả sử rằng mỗi phần tử của sieve ban đầu bằng không.

```
for (int x = 2; x <= n; x++) {  
    if (sieve[x]) continue;  
    for (int u = 2*x; u <= n; u += x) {  
        sieve[u] = x;  
    }  
}
```

Vòng lặp trong của thuật toán được thực hiện n/x lần với mỗi giá trị x . Vì vậy, một cận trên cho thời gian chạy của thuật toán là tổng điều hòa

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

Thực ra, thuật toán hiệu quả hơn, vì vòng lặp trong chỉ được thực hiện nếu số x là số nguyên tố. Có thể chứng minh rằng thời gian chạy của thuật toán chỉ là $O(n \log \log n)$, một độ phức tạp rất gần với $O(n)$.

Thuật toán Euclid

Ước chung lớn nhất của hai số a và b , $\text{gcd}(a, b)$, là số lớn nhất chia hết cả a và b , và **bội chung nhỏ nhất** của a và b , $\text{lcm}(a, b)$, là số nhỏ nhất chia hết cho cả a và b . Ví dụ, $\text{gcd}(24, 36) = 12$ và $\text{lcm}(24, 36) = 72$.

Ước chung lớn nhất và bội chung nhỏ nhất có liên hệ như sau:

$$\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

Thuật toán Euclid¹ cung cấp một cách hiệu quả để tìm ước chung lớn nhất của hai số. Thuật toán dựa trên công thức sau:

$$\text{gcd}(a, b) = \begin{cases} a & b = 0 \\ \text{gcd}(b, a \bmod b) & b \neq 0 \end{cases}$$

Ví dụ,

$$\text{gcd}(24, 36) = \text{gcd}(36, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12.$$

Thuật toán có thể được cài đặt như sau:

```
int gcd(int a, int b) {  
    if (b == 0) return a;  
    return gcd(b, a%b);  
}
```

Có thể chứng minh rằng thuật toán Euclid chạy trong thời gian $O(\log n)$, trong đó $n = \min(a, b)$. Trường hợp xấu nhất của thuật toán là khi a và b là hai số Fibonacci liên tiếp. Ví dụ,

$$\text{gcd}(13, 8) = \text{gcd}(8, 5) = \text{gcd}(5, 3) = \text{gcd}(3, 2) = \text{gcd}(2, 1) = \text{gcd}(1, 0) = 1.$$

Hàm phi Euler

Hai số a và b là **nguyên tố cùng nhau** nếu $\text{gcd}(a, b) = 1$. **Hàm phi Euler** $\varphi(n)$ cho biết số lượng các số nguyên tố cùng nhau với n nằm giữa 1 và n . Ví dụ, $\varphi(12) = 4$, vì 1, 5, 7 và 11 nguyên tố cùng nhau với 12.

Giá trị của $\varphi(n)$ có thể được tính từ phân tích thừa số nguyên tố của n bằng công thức

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1).$$

Ví dụ, $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$. Lưu ý rằng $\varphi(n) = n - 1$ nếu n là số nguyên tố.

21.2 Số học modulo

Trong **số học modulo**, tập các số được giới hạn sao cho chỉ dùng các số $0, 1, 2, \dots, m - 1$, trong đó m là một hằng số. Mỗi số x được biểu diễn bởi số

¹Euclid là một nhà toán học Hy Lạp sống vào khoảng năm 300 trước Công nguyên. Đây có lẽ là thuật toán đầu tiên được biết đến trong lịch sử.

$x \bmod m$: phần dư sau khi chia x cho m . Ví dụ, nếu $m = 17$, thì 75 được biểu diễn bởi $75 \bmod 17 = 7$.

Thông thường ta có thể lấy phần dư trước khi thực hiện tính toán. Cụ thể, các công thức sau đúng:

$$\begin{aligned}(x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\(x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\(x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\x^n \bmod m &= (x \bmod m)^n \bmod m\end{aligned}$$

Lũy thừa modulo

Ta thường cần tính hiệu quả giá trị $x^n \bmod m$. Điều này có thể thực hiện trong thời gian $O(\log n)$ bằng đệ quy sau:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ chẵn} \\ x^{n-1} \cdot x & n \text{ lẻ} \end{cases}$$

Điều quan trọng là trong trường hợp n chẵn, giá trị $x^{n/2}$ chỉ được tính một lần. Điều này đảm bảo rằng độ phức tạp thời gian của thuật toán là $O(\log n)$, vì n luôn bị chia đôi khi nó chẵn.

Hàm sau tính giá trị $x^n \bmod m$:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x, n/2, m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

Định lý Fermat và định lý Euler

Định lý Fermat phát biểu rằng

$$x^{m-1} \bmod m = 1$$

khi m là số nguyên tố và x nguyên tố cùng nhau với m . Điều này cũng suy ra

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

Tổng quát hơn, **định lý Euler** phát biểu rằng

$$x^{\varphi(m)} \bmod m = 1$$

khi x và m nguyên tố cùng nhau. Định lý Fermat suy ra từ định lý Euler, vì nếu m là số nguyên tố, thì $\varphi(m) = m - 1$.

Nghịch đảo modulo

Nghịch đảo của x modulo m là một số x^{-1} sao cho

$$xx^{-1} \bmod m = 1.$$

Ví dụ, nếu $x = 6$ và $m = 17$, thì $x^{-1} = 3$, vì $6 \cdot 3 \bmod 17 = 1$.

Dùng nghịch đảo modulo, ta có thể chia các số theo modulo m , vì chia cho x tương ứng với nhân với x^{-1} . Ví dụ, để tính giá trị $36/6 \bmod 17$, ta có thể dùng công thức $2 \cdot 3 \bmod 17$, vì $36 \bmod 17 = 2$ và $6^{-1} \bmod 17 = 3$.

Tuy nhiên, nghịch đảo modulo không phải lúc nào cũng tồn tại. Ví dụ, nếu $x = 2$ và $m = 4$, phương trình

$$xx^{-1} \bmod m = 1$$

không thể giải, vì mọi bội của 2 đều chẵn và phần dư không bao giờ có thể là 1 khi $m = 4$. Hóa ra giá trị $x^{-1} \bmod m$ có thể được tính chính xác khi x và m nguyên tố cùng nhau.

Nếu nghịch đảo modulo tồn tại, nó có thể được tính bằng công thức

$$x^{-1} = x^{\varphi(m)-1}.$$

Nếu m là số nguyên tố, công thức trở thành

$$x^{-1} = x^{m-2}.$$

Ví dụ,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Công thức này cho phép ta tính hiệu quả nghịch đảo modulo bằng thuật toán lũy thừa modulo. Công thức có thể được suy ra bằng định lý Euler. Trước hết, nghịch đảo modulo phải thỏa mãn phương trình sau:

$$xx^{-1} \bmod m = 1.$$

Mặt khác, theo định lý Euler,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

nên các số x^{-1} và $x^{\varphi(m)-1}$ bằng nhau.

Số học trong máy tính

Trong lập trình, số nguyên không dấu được biểu diễn modulo 2^k , trong đó k là số bit của kiểu dữ liệu. Một hệ quả thường gặp là số sẽ bị quay vòng nếu nó trở nên quá lớn.

Ví dụ, trong C++, các số kiểu unsigned int được biểu diễn modulo 2^{32} . Đoạn mã sau khai báo một biến unsigned int có giá trị 123456789. Sau đó, giá trị này sẽ được nhân với chính nó, và kết quả là $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3 Giải phương trình

Phương trình Diophantine

Một **phương trình Diophantine** là một phương trình dạng

$$ax + by = c,$$

trong đó a , b và c là các hằng số và cần tìm các giá trị của x và y . Mọi số trong phương trình phải là số nguyên. Ví dụ, một nghiệm của phương trình $5x + 2y = 11$ là $x = 3$ và $y = -2$.

Ta có thể giải hiệu quả một phương trình Diophantine bằng thuật toán Euclid. Hóa ra ta có thể mở rộng thuật toán Euclid để nó tìm các số x và y thỏa mãn phương trình sau:

$$ax + by = \gcd(a, b)$$

Một phương trình Diophantine có thể giải được nếu c chia hết cho $\gcd(a, b)$, và ngược lại thì không thể giải được.

Ví dụ, hãy tìm các số x và y thỏa mãn phương trình sau:

$$39x + 15y = 12$$

Phương trình có thể giải được, vì $\gcd(39, 15) = 3$ và $3 \mid 12$. Khi thuật toán Euclid tính ước chung lớn nhất của 39 và 15, nó tạo ra chuỗi lời gọi hàm sau:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

Điều này tương ứng với các phương trình sau:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Dùng các phương trình này, ta có thể suy ra

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

và bằng cách nhân biểu thức này với 4, ta được

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

nên một nghiệm của phương trình là $x = 8$ và $y = -20$.

Nghiệm của một phương trình Diophantine không duy nhất, vì ta có thể tạo vô hạn nghiệm nếu biết một nghiệm. Nếu một cặp (x, y) là nghiệm, thì mọi cặp

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

cũng là nghiệm, trong đó k là số nguyên bất kỳ.

Định lý số dư Trung Hoa

Định lý số dư Trung Hoa giải một hệ phương trình dạng

$$\begin{aligned}x &= a_1 \bmod m_1 \\x &= a_2 \bmod m_2 \\&\dots \\x &= a_n \bmod m_n\end{aligned}$$

trong đó mọi cặp trong $m_1, m_2, \dots, m_n$ đều nguyên tố cùng nhau.

Gọi x_m^{-1} là nghịch đảo của x modulo m , và

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Dùng ký hiệu này, một nghiệm của các phương trình là

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

Trong nghiệm này, với mỗi $k = 1, 2, \dots, n$,

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

vì

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Vì tất cả các hạng tử khác trong tổng đều chia hết cho m_k , chúng không ảnh hưởng đến phần dư, và $x \bmod m_k = a_k$.

Ví dụ, một nghiệm của

$$\begin{aligned}x &= 3 \bmod 5 \\x &= 4 \bmod 7 \\x &= 2 \bmod 3\end{aligned}$$

là

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Một khi đã tìm được một nghiệm x , ta có thể tạo vô hạn nghiệm khác, vì mọi số có dạng

$$x + m_1 m_2 \cdots m_n$$

đều là nghiệm.

21.4 Các kết quả khác

Định lý Lagrange

Định lý Lagrange phát biểu rằng mọi số nguyên dương có thể được biểu diễn dưới dạng tổng của bốn bình phương, tức là $a^2 + b^2 + c^2 + d^2$. Ví dụ, số 123 có thể được biểu diễn dưới dạng tổng $8^2 + 5^2 + 5^2 + 3^2$.

Định lý Zeckendorf

Định lý Zeckendorf phát biểu rằng mọi số nguyên dương có một biểu diễn duy nhất dưới dạng tổng các số Fibonacci sao cho không có hai số nào bằng nhau hoặc là hai số Fibonacci liên tiếp. Ví dụ, số 74 có thể được biểu diễn dưới dạng tổng $55 + 13 + 5 + 1$.

Bộ ba Pythagoras

Một **bộ ba Pythagoras** là một bộ ba (a, b, c) thỏa mãn định lý Pythagoras $a^2 + b^2 = c^2$, nghĩa là tồn tại một tam giác vuông có độ dài các cạnh là a , b và c . Ví dụ, $(3, 4, 5)$ là một bộ ba Pythagoras.

Nếu (a, b, c) là một bộ ba Pythagoras, mọi bộ ba dạng (ka, kb, kc) cũng là bộ ba Pythagoras với $k > 1$. Một bộ ba Pythagoras là *nguyên thủy* nếu a , b và c nguyên tố cùng nhau, và mọi bộ ba Pythagoras có thể được xây dựng từ các bộ ba nguyên thủy bằng một hệ số nhân k .

Công thức Euclid có thể được dùng để sinh tất cả các bộ ba Pythagoras nguyên thủy. Mỗi bộ ba như vậy có dạng

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

trong đó $0 < m < n$, n và m nguyên tố cùng nhau và ít nhất một trong n và m là số chẵn. Ví dụ, khi $m = 1$ và $n = 2$, công thức sinh ra bộ ba Pythagoras nhỏ nhất

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

Định lý Wilson

Định lý Wilson phát biểu rằng một số n là số nguyên tố khi và chỉ khi

$$(n - 1)! \bmod n = n - 1.$$

Ví dụ, số 11 là số nguyên tố, vì

$$10! \bmod 11 = 10,$$

và số 12 không phải số nguyên tố, vì

$$11! \bmod 12 = 0 \neq 11.$$

Do đó, định lý Wilson có thể được dùng để xác định một số có phải số nguyên tố hay không. Tuy nhiên, trong thực tế, định lý này không thể áp dụng cho các giá trị lớn của n , vì rất khó tính các giá trị của $(n - 1)!$ khi n lớn.

Chương 22

Tổ hợp

Tổ hợp nghiên cứu các phương pháp đếm các cách kết hợp của các đối tượng. Thông thường, mục tiêu là tìm một cách đếm các tổ hợp một cách hiệu quả mà không cần sinh riêng từng tổ hợp.

Ví dụ, xét bài toán đếm số cách biểu diễn một số nguyên n dưới dạng tổng các số nguyên dương. Ví dụ, có 8 biểu diễn cho 4:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- 4

Một bài toán tổ hợp thường có thể được giải bằng một hàm đệ quy. Trong bài toán này, ta có thể định nghĩa hàm $f(n)$ cho biết số biểu diễn của n . Ví dụ, theo ví dụ trên, $f(4) = 8$. Các giá trị của hàm có thể được tính đệ quy như sau:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

Trường hợp cơ sở là $f(0) = 1$, vì tổng rỗng biểu diễn số 0. Sau đó, nếu $n > 0$, ta xét mọi cách chọn số đầu tiên của tổng. Nếu số đầu tiên là k , có $f(n-k)$ biểu diễn cho phần còn lại của tổng. Vì vậy, ta tính tổng của mọi giá trị dạng $f(n-k)$ với $k < n$.

Các giá trị đầu tiên của hàm là:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

Đôi khi, một công thức đệ quy có thể được thay thế bằng một công thức đóng. Trong bài toán này,

$$f(n) = 2^{n-1},$$

dựa trên thực tế rằng có $n - 1$ vị trí có thể đặt dấu $+$ trong tổng và ta có thể chọn bất kỳ tập con nào của chúng.

22.1 Hệ số nhị thức

Hệ số nhị thức $\binom{n}{k}$ bằng số cách ta có thể chọn một tập con gồm k phần tử từ một tập có n phần tử. Ví dụ, $\binom{5}{3} = 10$, vì tập $\{1, 2, 3, 4, 5\}$ có 10 tập con gồm 3 phần tử:

$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$

Công thức 1

Các hệ số nhị thức có thể được tính đệ quy như sau:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Ý tưởng là cố định một phần tử x trong tập. Nếu x được đưa vào tập con, ta phải chọn $k - 1$ phần tử từ $n - 1$ phần tử, và nếu x không được đưa vào tập con, ta phải chọn k phần tử từ $n - 1$ phần tử.

Các trường hợp cơ sở của đệ quy là

$$\binom{n}{0} = \binom{n}{n} = 1,$$

vì luôn có đúng một cách để xây dựng tập con rỗng và một tập con chứa tất cả các phần tử.

Công thức 2

Một cách khác để tính các hệ số nhị thức là:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Có $n!$ hoán vị của n phần tử. Ta xét tất cả các hoán vị và luôn đưa k phần tử đầu tiên của hoán vị vào tập con. Vì thứ tự của các phần tử trong tập con và ngoài tập con không quan trọng, kết quả được chia cho $k!$ và $(n - k)!$

Tính chất

Với các hệ số nhị thức,

$$\binom{n}{k} = \binom{n}{n-k},$$

vì thực chất ta chia một tập có n phần tử thành hai tập con: tập thứ nhất chứa k phần tử và tập thứ hai chứa $n - k$ phần tử.

Tổng các hệ số nhị thức là

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Lý do có tên gọi “hệ số nhị thức” có thể thấy khi nhị thức $(a + b)$ được nâng lên lũy thừa thứ n :

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

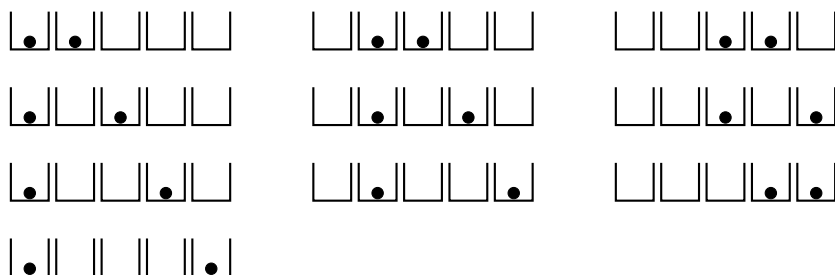
Các hệ số nhị thức cũng xuất hiện trong **tam giác Pascal**, trong đó mỗi giá trị bằng tổng của hai giá trị phía trên:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ 1 & 4 & 6 & 4 & 1 & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \end{array}$$

Hộp và bóng

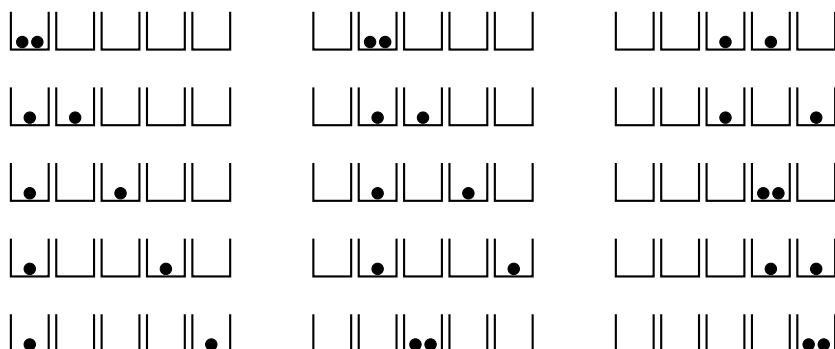
“Hộp và bóng” là một mô hình hữu ích, trong đó ta đếm các cách đặt k quả bóng vào n chiếc hộp. Hãy xét ba kịch bản:

Kịch bản 1: Mỗi hộp có thể chứa nhiều nhất một quả bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 10 lời giải:



Trong kịch bản này, đáp án trực tiếp là hệ số nhị thức $\binom{n}{k}$.

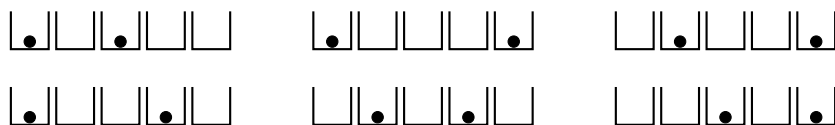
Kịch bản 2: Một hộp có thể chứa nhiều quả bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 15 lời giải:



Quá trình đặt bóng vào các hộp có thể được biểu diễn bằng một chuỗi gồm các ký hiệu “o” và “→”. Ban đầu, giả sử ta đang đứng ở hộp ngoài cùng bên trái. Ký hiệu “o” nghĩa là ta đặt một quả bóng vào hộp hiện tại, và ký hiệu “→” nghĩa là ta di chuyển sang hộp kế tiếp bên phải.

Dùng ký hiệu này, mỗi lời giải là một chuỗi chứa k lần ký hiệu “o” và $n - 1$ lần ký hiệu “→”. Ví dụ, lời giải phía trên bên phải trong hình trên tương ứng với chuỗi “→ → o → o →”. Vì vậy, số lời giải là $\binom{k+n-1}{k}$.

Kịch bản 3: Mỗi hộp có thể chứa nhiều nhất một quả bóng, và thêm nữa, không có hai hộp kề nhau cùng chứa bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 6 lời giải:



Trong kịch bản này, ta có thể giả sử rằng k quả bóng ban đầu đã được đặt vào các hộp và có một hộp trống giữa mỗi hai hộp kề nhau có bóng. Nhiệm vụ còn lại là chọn vị trí cho các hộp trống còn lại. Có $n - 2k + 1$ hộp như vậy và $k + 1$ vị trí cho chúng. Vì vậy, dùng công thức của kịch bản 2, số lời giải là $\binom{n-k+1}{n-2k+1}$.

Hệ số đa thức

Hệ số đa thức

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

bằng số cách ta có thể chia n phần tử thành các tập con có kích thước $k_1, k_2, \dots, k_m$, trong đó $k_1 + k_2 + \cdots + k_m = n$. Hệ số đa thức có thể được xem là một tổng quát hóa của hệ số nhị thức; nếu $m = 2$, công thức trên tương ứng với công thức hệ số nhị thức.

22.2 Số Catalan

Số Catalan C_n bằng số lượng biểu thức ngoặc hợp lệ gồm n dấu ngoặc trái và n dấu ngoặc phải.

Ví dụ, $C_3 = 5$, vì ta có thể xây dựng các biểu thức ngoặc sau bằng ba dấu ngoặc trái và ba dấu ngoặc phải:

- $()()()$
- $((()))$
- $()(())$
- $((())())$
- $((()()))$

Biểu thức ngoặc

Chính xác thì *biểu thức ngoặc hợp lệ* là gì? Các quy tắc sau định nghĩa chính xác tất cả các biểu thức ngoặc hợp lệ:

- Biểu thức ngoặc rỗng là hợp lệ.
- Nếu biểu thức A hợp lệ, thì biểu thức (A) cũng hợp lệ.
- Nếu các biểu thức A và B hợp lệ, thì biểu thức AB cũng hợp lệ.

Một cách khác để đặc trưng các biểu thức ngoặc hợp lệ là nếu ta chọn bất kỳ tiền tố nào của biểu thức như vậy, nó phải chứa ít nhất nhiều dấu ngoặc trái bằng số dấu ngoặc phải. Ngoài ra, biểu thức đầy đủ phải chứa số lượng dấu ngoặc trái và dấu ngoặc phải bằng nhau.

Công thức 1

Các số Catalan có thể được tính bằng công thức

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

Tổng này xét các cách chia biểu thức thành hai phần sao cho cả hai phần đều là các biểu thức hợp lệ và phần thứ nhất ngắn nhất có thể nhưng không rỗng. Với mỗi i , phần thứ nhất chứa $i + 1$ cặp dấu ngoặc và số lượng biểu thức là tích của các giá trị sau:

- C_i : số cách xây dựng một biểu thức bằng các dấu ngoặc của phần thứ nhất, không tính cặp ngoặc ngoài cùng
- C_{n-i-1} : số cách xây dựng một biểu thức bằng các dấu ngoặc của phần thứ hai

Trường hợp cơ sở là $C_0 = 1$, vì ta có thể xây dựng một biểu thức ngoặc rỗng bằng không cặp ngoặc.

Công thức 2

Các số Catalan cũng có thể được tính bằng hệ số nhị thức:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Công thức có thể được giải thích như sau:

Có tổng cộng $\binom{2n}{n}$ cách xây dựng một biểu thức ngoặc (không nhất thiết hợp lệ) chứa n dấu ngoặc trái và n dấu ngoặc phải. Hãy tính số lượng các biểu thức như vậy *không* hợp lệ.

Nếu một biểu thức ngoặc không hợp lệ, nó phải chứa một tiền tố mà trong đó số dấu ngoặc phải lớn hơn số dấu ngoặc trái. Ý tưởng là đảo chiều từng dấu ngoặc thuộc tiền tố đó. Ví dụ, biểu thức $()()()$ chứa tiền tố $()$, và sau khi đảo tiền tố, biểu thức trở thành $((()())$.

Biểu thức thu được gồm $n + 1$ dấu ngoặc trái và $n - 1$ dấu ngoặc phải. Số lượng các biểu thức như vậy là $\binom{2n}{n+1}$, bằng số lượng biểu thức ngoặc không hợp lệ. Vì vậy, số lượng biểu thức ngoặc hợp lệ có thể được tính bằng công thức

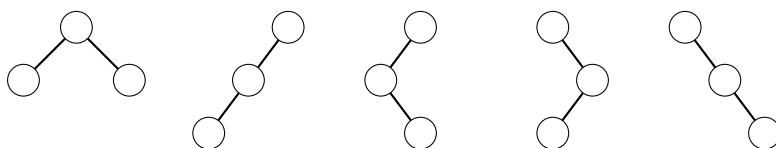
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Đếm cây

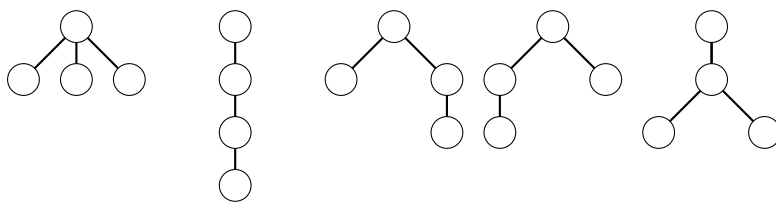
Các số Catalan cũng liên quan đến cây:

- có C_n cây nhị phân gồm n đỉnh
- có C_{n-1} cây có gốc gồm n đỉnh

Ví dụ, với $C_3 = 5$, các cây nhị phân là



và các cây có gốc là

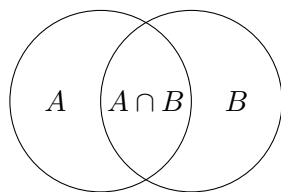


22.3 Nguyên lý bù trừ

Nguyên lý bù trừ là một kỹ thuật có thể dùng để đếm kích thước của hợp các tập khi biết kích thước của các giao, và ngược lại. Một ví dụ đơn giản của kỹ thuật này là công thức

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

trong đó A và B là các tập và $|X|$ ký hiệu kích thước của X . Công thức có thể được minh họa như sau:

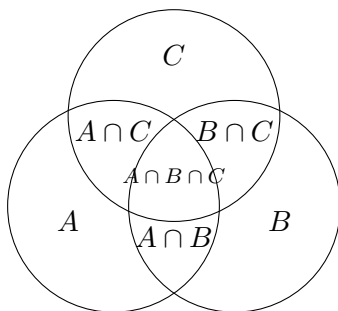


Mục tiêu của ta là tính kích thước của hợp $A \cup B$, tương ứng với diện tích của vùng thuộc ít nhất một hình tròn. Hình vẽ cho thấy ta có thể tính diện tích của $A \cup B$ bằng cách trước hết cộng diện tích của A và B , rồi trừ đi diện tích của $A \cap B$.

Ý tưởng tương tự có thể được áp dụng khi số lượng tập lớn hơn. Khi có ba tập, công thức bù trừ là

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

và hình tương ứng là



Trong trường hợp tổng quát, kích thước của hợp $X_1 \cup X_2 \cup \dots \cup X_n$ có thể được tính bằng cách xét mọi giao có thể chứa một số tập trong $X_1, X_2, \dots, X_n$. Nếu giao chứa số lẻ tập, kích thước của nó được cộng vào đáp án, ngược lại kích thước của nó được trừ khỏi đáp án.

Lưu ý rằng cũng có các công thức tương tự để tính kích thước của một giao từ kích thước của các hợp. Ví dụ,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

và

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Hoán vị mất trật tự

Làm ví dụ, hãy đếm số **hoán vị mất trật tự** của các phần tử $\{1, 2, \dots, n\}$, tức là các hoán vị trong đó không phần tử nào nằm ở vị trí ban đầu của nó. Ví dụ, khi $n = 3$, có hai hoán vị mất trật tự: $(2, 3, 1)$ và $(3, 1, 2)$.

Một cách giải bài toán là dùng nguyên lý bù trừ. Gọi X_k là tập các hoán vị chứa phần tử k ở vị trí k . Ví dụ, khi $n = 3$, các tập như sau:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Dùng các tập này, số hoán vị mất trật tự bằng

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

vì vậy chỉ cần tính kích thước của hợp. Dùng nguyên lý bù trừ, việc này quy về tính kích thước các giao, có thể thực hiện hiệu quả. Ví dụ, khi $n = 3$, kích thước của $|X_1 \cup X_2 \cup X_3|$ là

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

nên số lời giải là $3! - 4 = 2$.

Hóa ra bài toán cũng có thể được giải mà không dùng nguyên lý bù trừ. Gọi $f(n)$ là số hoán vị mất trật tự của $\{1, 2, \dots, n\}$. Ta có thể dùng công thức đệ quy sau:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Công thức có thể được suy ra bằng cách xét các khả năng phần tử 1 thay đổi vị trí trong hoán vị mất trật tự. Có $n-1$ cách chọn một phần tử x thay thế phần tử 1. Với mỗi lựa chọn như vậy, có hai khả năng:

Khả năng 1: Ta cũng thay thế phần tử x bằng phần tử 1. Sau đó, nhiệm vụ còn lại là xây dựng một hoán vị mất trật tự của $n-2$ phần tử.

Khả năng 2: Ta thay thế phần tử x bằng một phần tử khác 1. Bây giờ ta phải xây dựng một hoán vị mất trật tự của $n-1$ phần tử, vì ta không thể thay thế phần tử x bằng phần tử 1, và tất cả các phần tử khác phải được thay đổi.

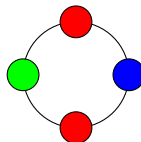
22.4 Bổ đề Burnside

Bổ đề Burnside có thể được dùng để đếm số tổ hợp sao cho chỉ một đại diện được tính cho mỗi nhóm các tổ hợp đối xứng. Bổ đề Burnside phát biểu rằng số lượng tổ hợp là

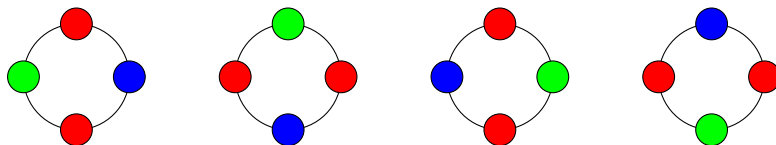
$$\sum_{k=1}^n \frac{c(k)}{n},$$

trong đó có n cách thay đổi vị trí của một tổ hợp, và có $c(k)$ tổ hợp không đổi khi áp dụng cách thứ k .

Làm ví dụ, hãy tính số vòng cổ gồm n hạt, trong đó mỗi hạt có m màu có thể. Hai vòng cổ là đối xứng nếu chúng giống nhau sau khi xoay. Ví dụ, vòng cổ



có các vòng cổ đối xứng sau:



Có n cách thay đổi vị trí của một vòng cổ, vì ta có thể xoay nó $0, 1, \dots, n-1$ bước theo chiều kim đồng hồ. Nếu số bước là 0, tất cả m^n vòng cổ giữ nguyên, và nếu số bước là 1, chỉ m vòng cổ mà mọi hạt có cùng màu giữ nguyên.

Tổng quát hơn, khi số bước là k , tổng cộng

$$m^{\gcd(k,n)}$$

vòng cổ giữ nguyên, trong đó $\gcd(k, n)$ là ước chung lớn nhất của k và n . Lý do là các khối hạt có kích thước $\gcd(k, n)$ sẽ thay thế lẫn nhau. Vì vậy, theo bổ đề Burnside, số vòng cổ là

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

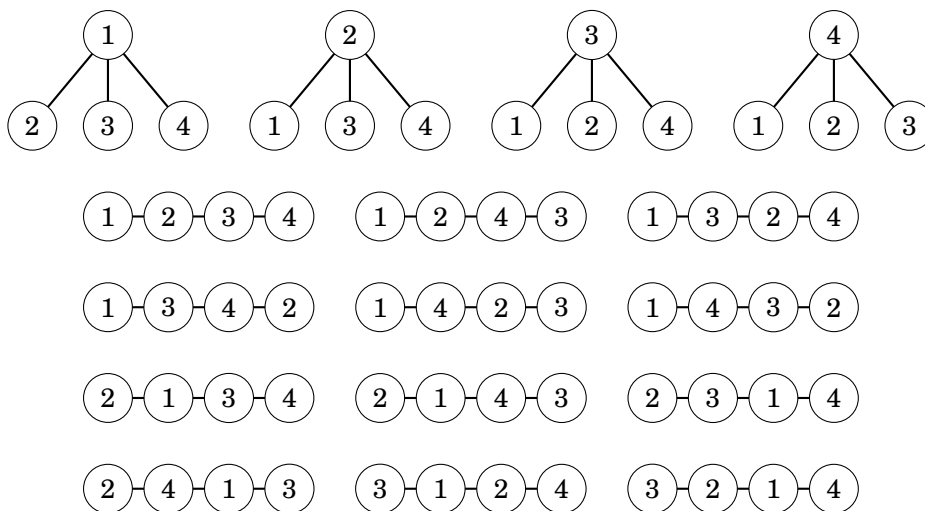
Ví dụ, số vòng cổ độ dài 4 với 3 màu là

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Công thức Cayley

Công thức Cayley phát biểu rằng có n^{n-2} cây có nhãn chứa n đỉnh. Các đỉnh được gán nhãn $1, 2, \dots, n$, và hai cây là khác nhau nếu cấu trúc hoặc cách gán nhãn của chúng khác nhau.

Ví dụ, khi $n = 4$, số lượng cây có nhãn là $4^{4-2} = 16$:

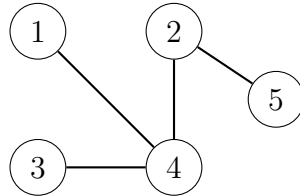


Tiếp theo ta sẽ xem công thức Cayley có thể được suy ra bằng mã Prüfer như thế nào.

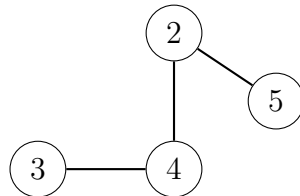
Mã Prüfer

Một **mã Prüfer** là một dãy gồm $n - 2$ số mô tả một cây có nhãn. Mã được xây dựng bằng một quá trình xóa $n - 2$ lá khỏi cây. Ở mỗi bước, lá có nhãn nhỏ nhất bị xóa, và nhãn của đỉnh kề duy nhất của nó được thêm vào mã.

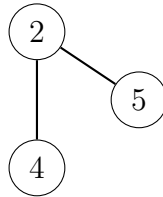
Ví dụ, hãy tính mã Prüfer của đồ thị sau:



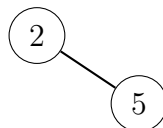
Trước hết ta xóa đỉnh 1 và thêm đỉnh 4 vào mã:



Sau đó ta xóa đỉnh 3 và thêm đỉnh 4 vào mã:



Cuối cùng ta xóa đỉnh 4 và thêm đỉnh 2 vào mã:



Vì vậy, mã Prüfer của đồ thị là $[4, 4, 2]$.

Ta có thể xây dựng mã Prüfer cho mọi cây, và quan trọng hơn, cây ban đầu có thể được khôi phục từ một mã Prüfer. Do đó, số lượng cây có nhãn gồm n đỉnh bằng n^{n-2} , tức số lượng mã Prüfer có kích thước n .

Chương 23

Ma trận

Ma trận là một khái niệm toán học tương ứng với một mảng hai chiều trong lập trình. Ví dụ,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

là một ma trận kích thước 3×4 , tức là nó có 3 hàng và 4 cột. Ký hiệu $[i, j]$ chỉ phần tử ở hàng i và cột j trong một ma trận. Ví dụ, trong ma trận trên, $A[2, 3] = 8$ và $A[3, 1] = 9$.

Một trường hợp đặc biệt của ma trận là **vector**, tức một ma trận một chiều kích thước $n \times 1$. Ví dụ,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

là một vector chứa ba phần tử.

Chuyển vị A^T của một ma trận A thu được khi hoán đổi các hàng và cột của A , tức là $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Một ma trận là **ma trận vuông** nếu nó có cùng số hàng và số cột. Ví dụ, ma trận sau là một ma trận vuông:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Phép toán

Tổng $A + B$ của các ma trận A và B được định nghĩa nếu hai ma trận có cùng kích thước. Kết quả là một ma trận trong đó mỗi phần tử là tổng của

các phần tử tương ứng trong A và B .

Ví dụ,

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Nhân một ma trận A với một giá trị x nghĩa là mỗi phần tử của A được nhân với x . Ví dụ,

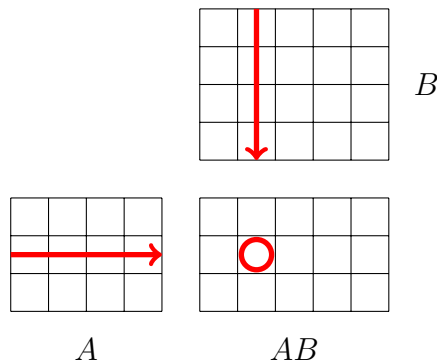
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Nhân ma trận

Tích AB của các ma trận A và B được định nghĩa nếu A có kích thước $a \times n$ và B có kích thước $n \times b$, tức là chiều rộng của A bằng chiều cao của B . Kết quả là một ma trận kích thước $a \times b$ có các phần tử được tính bằng công thức

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j].$$

Ý tưởng là mỗi phần tử của AB là tổng các tích của phần tử trong A và B theo hình sau:



Ví dụ,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Phép nhân ma trận có tính kết hợp, nên $A(BC) = (AB)C$ đúng, nhưng không có tính giao hoán, nên $AB = BA$ thường không đúng.

Ma trận đơn vị là một ma trận vuông trong đó mỗi phần tử trên đường chéo bằng 1 và mọi phần tử khác bằng 0. Ví dụ, ma trận sau là ma trận đơn vị 3×3 :

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Nhân một ma trận với ma trận đơn vị không làm thay đổi ma trận đó. Ví dụ,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{và} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Với một thuật toán trực tiếp, ta có thể tính tích của hai ma trận $n \times n$ trong thời gian $O(n^3)$. Cũng có các thuật toán hiệu quả hơn cho phép nhân ma trận¹, nhưng chúng chủ yếu có ý nghĩa lý thuyết và không cần thiết trong lập trình thi đấu.

Lũy thừa ma trận

Lũy thừa A^k của một ma trận A được định nghĩa nếu A là ma trận vuông. Định nghĩa dựa trên phép nhân ma trận:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ lần}}$$

Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Ngoài ra, A^0 là một ma trận đơn vị. Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Ma trận A^k có thể được tính hiệu quả trong thời gian $O(n^3 \log k)$ bằng thuật toán ở Chương 21.2. Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

Định thức

Định thức $\det(A)$ của một ma trận A được định nghĩa nếu A là ma trận vuông. Nếu A có kích thước 1×1 , thì $\det(A) = A[1, 1]$. Định thức của một ma trận lớn hơn được tính đệ quy bằng công thức

$$\det(A) = \sum_{j=1}^n A[1, j]C[1, j],$$

¹Thuật toán đầu tiên như vậy là thuật toán Strassen, được công bố năm 1969 [63], có độ phức tạp thời gian $O(n^{2.80735})$; thuật toán tốt nhất hiện nay [27] chạy trong thời gian $O(n^{2.37286})$.

trong đó $C[i, j]$ là **phần bù đại số** của A tại $[i, j]$. Phần bù đại số được tính bằng công thức

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

trong đó $M[i, j]$ thu được bằng cách xóa hàng i và cột j khỏi A . Do hệ số $(-1)^{i+j}$ trong phần bù đại số, các định thức luân phiên mang dấu dương và âm. Ví dụ,

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

và

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

Định thức của A cho ta biết có tồn tại một **ma trận nghịch đảo** A^{-1} sao cho $A \cdot A^{-1} = I$ hay không, trong đó I là ma trận đơn vị. Hóa ra A^{-1} tồn tại khi và chỉ khi $\det(A) \neq 0$, và nó có thể được tính bằng công thức

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Ví dụ,

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 Truy hồi tuyến tính

Một **truy hồi tuyến tính** là một hàm $f(n)$ có các giá trị ban đầu là $f(0), f(1), \dots, f(k-1)$ và các giá trị lớn hơn được tính đệ quy bằng công thức

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

trong đó $c_1, c_2, \dots, c_k$ là các hệ số hằng.

Quy hoạch động có thể được dùng để tính bất kỳ giá trị $f(n)$ nào trong thời gian $O(kn)$ bằng cách tính lần lượt mọi giá trị $f(0), f(1), \dots, f(n)$. Tuy nhiên, nếu k nhỏ, có thể tính $f(n)$ hiệu quả hơn nhiều trong thời gian $O(k^3 \log n)$ bằng các phép toán ma trận.

Số Fibonacci

Một ví dụ đơn giản về truy hồi tuyến tính là hàm sau định nghĩa các số Fibonacci:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Trong trường hợp này, $k = 2$ và $c_1 = c_2 = 1$.

Để tính các số Fibonacci một cách hiệu quả, ta biểu diễn công thức Fibonacci dưới dạng một ma trận vuông X kích thước 2×2 , sao cho hệ thức sau đúng:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Vì vậy, các giá trị $f(i)$ và $f(i+1)$ được đưa vào làm “đầu vào” cho X , và X tính các giá trị $f(i+1)$ và $f(i+2)$ từ chúng. Hóa ra một ma trận như vậy là

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Ví dụ,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Vì vậy, ta có thể tính $f(n)$ bằng công thức

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Giá trị của X^n có thể được tính trong thời gian $O(\log n)$, nên giá trị của $f(n)$ cũng có thể được tính trong thời gian $O(\log n)$.

Trường hợp tổng quát

Bây giờ hãy xét trường hợp tổng quát, trong đó $f(n)$ là một truy hồi tuyến tính bất kỳ. Một lần nữa, mục tiêu của ta là xây dựng một ma trận X sao cho

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Một ma trận như vậy là

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

Trong $k-1$ hàng đầu tiên, mọi phần tử đều bằng 0 trừ một phần tử bằng 1. Các hàng này thay $f(i)$ bằng $f(i+1)$, $f(i+1)$ bằng $f(i+2)$, v.v. Hàng cuối chứa các hệ số của truy hồi để tính giá trị mới $f(i+k)$.

Bây giờ, $f(n)$ có thể được tính trong thời gian $O(k^3 \log n)$ bằng công thức

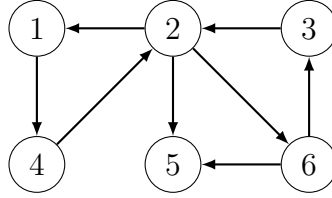
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3 Đồ thị và ma trận

Đếm đường đi

Các lũy thừa của ma trận kề của một đồ thị có một tính chất thú vị. Khi V là ma trận kề của một đồ thị không trọng số, ma trận V^n chứa số lượng đường đi gồm n cạnh giữa các đỉnh trong đồ thị.

Ví dụ, với đồ thị



ma trận kề là

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Bây giờ, ví dụ, ma trận

$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

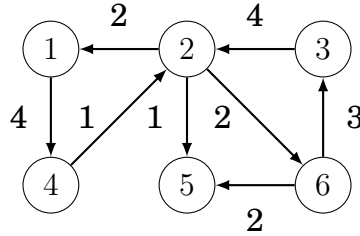
chứa số lượng đường đi gồm 4 cạnh giữa các đỉnh. Ví dụ, $V^4[2, 5] = 2$, vì có hai đường đi gồm 4 cạnh từ đỉnh 2 đến đỉnh 5: $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ và $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Đường đi ngắn nhất

Dùng ý tưởng tương tự trong một đồ thị có trọng số, ta có thể tính cho mỗi cặp đỉnh độ dài nhỏ nhất của một đường đi giữa chúng chứa đúng n cạnh.

Để tính điều này, ta phải định nghĩa phép nhân ma trận theo một cách mới, sao cho ta không tính số lượng đường đi mà tối thiểu hóa độ dài đường đi.

Làm ví dụ, xét đồ thị sau:



Hãy xây dựng một ma trận kề trong đó ∞ nghĩa là cạnh không tồn tại, và các giá trị khác tương ứng với trọng số cạnh. Ma trận là

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Thay vì công thức

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

bây giờ ta dùng công thức

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j]$$

cho phép nhân ma trận, nên ta tính một giá trị nhỏ nhất thay vì một tổng, và một tổng các phần tử thay vì một tích. Sau sửa đổi này, các lũy thừa ma trận tương ứng với đường đi ngắn nhất trong đồ thị.

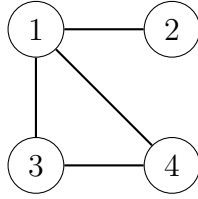
Ví dụ, vì

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

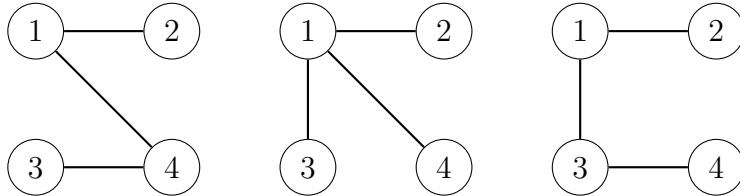
ta có thể kết luận rằng độ dài nhỏ nhất của một đường đi gồm 4 cạnh từ đỉnh 2 đến đỉnh 5 là 8. Một đường đi như vậy là $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

Định lý Kirchhoff

Định lý Kirchhoff cung cấp một cách tính số lượng cây khung của một đồ thị dưới dạng định thức của một ma trận đặc biệt. Ví dụ, đồ thị



có ba cây khung:



Để tính số lượng cây khung, ta xây dựng một **ma trận Laplace** L , trong đó $L[i, i]$ là bậc của đỉnh i và $L[i, j] = -1$ nếu có cạnh giữa các đỉnh i và j , ngược lại $L[i, j] = 0$. Ma trận Laplace của đồ thị trên như sau:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Có thể chứng minh rằng số lượng cây khung bằng định thức của một ma trận thu được khi ta xóa một hàng bất kỳ và một cột bất kỳ khỏi L . Ví dụ, nếu ta xóa hàng đầu tiên và cột đầu tiên, kết quả là

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

Định thức luôn giống nhau, bất kể ta xóa hàng và cột nào khỏi L .

Lưu ý rằng công thức Cayley trong Chương 22.5 là một trường hợp đặc biệt của định lý Kirchhoff, vì trong một đồ thị đầy đủ gồm n đỉnh

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

Chương 24

Xác suất

Một **xác suất** là một số thực nằm giữa 0 và 1 cho biết một sự kiện có khả năng xảy ra đến mức nào. Nếu một sự kiện chắc chắn xảy ra, xác suất của nó là 1, và nếu một sự kiện không thể xảy ra, xác suất của nó là 0. Xác suất của một sự kiện được ký hiệu là $P(\dots)$, trong đó ba dấu chấm mô tả sự kiện.

Ví dụ, khi gieo một con xúc xắc, kết quả là một số nguyên từ 1 đến 6, và xác suất của mỗi kết quả là $1/6$. Chẳng hạn, ta có thể tính các xác suất sau:

- $P(\text{"kết quả là 4"}) = 1/6$
- $P(\text{"kết quả không phải là 6"}) = 5/6$
- $P(\text{"kết quả là số chẵn"}) = 1/2$

24.1 Tính toán

Để tính xác suất của một sự kiện, ta có thể dùng tổ hợp hoặc mô phỏng quá trình sinh ra sự kiện đó. Làm ví dụ, hãy tính xác suất rút được ba lá bài có cùng giá trị từ một bộ bài đã xáo (ví dụ, ♠8, ♣8 và ♦8).

Cách 1

Ta có thể tính xác suất bằng công thức

$$\frac{\text{số kết quả mong muốn}}{\text{tổng số kết quả}}.$$

Trong bài toán này, các kết quả mong muốn là những kết quả mà giá trị của mỗi lá bài đều giống nhau. Có $13 \binom{4}{3}$ kết quả như vậy, vì có 13 khả năng cho giá trị của các lá bài và $\binom{4}{3}$ cách chọn 3 chất từ 4 chất có thể.

Có tổng cộng $\binom{52}{3}$ kết quả, vì ta chọn 3 lá từ 52 lá. Do đó, xác suất của sự kiện là

$$\frac{13 \binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Cách 2

Một cách khác để tính xác suất là mô phỏng quá trình sinh ra sự kiện. Trong ví dụ này, ta rút ba lá bài, nên quá trình gồm ba bước. Ta cần mỗi bước của quá trình đều thành công.

Việc rút lá đầu tiên chắc chắn thành công, vì không có ràng buộc nào. Bước thứ hai thành công với xác suất $3/51$, vì còn 51 lá bài và 3 lá trong số đó có cùng giá trị với lá đầu tiên. Tương tự, bước thứ ba thành công với xác suất $2/50$.

Xác suất để toàn bộ quá trình thành công là

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 Sự kiện

Một sự kiện trong lý thuyết xác suất có thể được biểu diễn bằng một tập

$$A \subset X,$$

trong đó X chứa tất cả các kết quả có thể và A là một tập con các kết quả. Ví dụ, khi gieo một con xúc xắc, các kết quả là

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Bây giờ, chẳng hạn, sự kiện "kết quả là số chẵn" tương ứng với tập

$$A = \{2, 4, 6\}.$$

Mỗi kết quả x được gán một xác suất $p(x)$. Khi đó, xác suất $P(A)$ của một sự kiện A có thể được tính bằng tổng xác suất của các kết quả theo công thức

$$P(A) = \sum_{x \in A} p(x).$$

Ví dụ, khi gieo một con xúc xắc, $p(x) = 1/6$ với mỗi kết quả x , nên xác suất của sự kiện "kết quả là số chẵn" là

$$p(2) + p(4) + p(6) = 1/2.$$

Tổng xác suất của các kết quả trong X phải bằng 1, tức là $P(X) = 1$.

Vì các sự kiện trong lý thuyết xác suất là các tập, ta có thể xử lý chúng bằng các phép toán tập hợp chuẩn:

- **Phần bù** $\bar{A}$ nghĩa là " A không xảy ra". Ví dụ, khi gieo một con xúc xắc, phần bù của $A = \{2, 4, 6\}$ là $\bar{A} = \{1, 3, 5\}$.
- **Hợp** $A \cup B$ nghĩa là " A hoặc B xảy ra". Ví dụ, hợp của $A = \{2, 5\}$ và $B = \{4, 5, 6\}$ là $A \cup B = \{2, 4, 5, 6\}$.
- **Giao** $A \cap B$ nghĩa là " A và B cùng xảy ra". Ví dụ, giao của $A = \{2, 5\}$ và $B = \{4, 5, 6\}$ là $A \cap B = \{5\}$.

Phần bù

Xác suất của phần bù $\bar{A}$ được tính bằng công thức

$$P(\bar{A}) = 1 - P(A).$$

Đôi khi, ta có thể giải một bài toán dễ dàng bằng phần bù, tức là giải bài toán đối ngược. Ví dụ, xác suất nhận được ít nhất một mặt sáu khi gieo xúc xắc mười lần là

$$1 - (5/6)^{10}.$$

Ở đây $5/6$ là xác suất để kết quả của một lần gieo không phải là sáu, và $(5/6)^{10}$ là xác suất để không lần nào trong mười lần gieo ra mặt sáu. Phần bù của sự kiện này chính là đáp án của bài toán.

Hợp

Xác suất của hợp $A \cup B$ được tính bằng công thức

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Ví dụ, khi gieo một con xúc xắc, hợp của các sự kiện

$$A = \text{"kết quả là số chẵn"}$$

và

$$B = \text{"kết quả nhỏ hơn 4"}$$

là

$$A \cup B = \text{"kết quả là số chẵn hoặc nhỏ hơn 4"},$$

và xác suất của nó là

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Nếu các sự kiện A và B **rời nhau**, tức là $A \cap B$ rỗng, xác suất của sự kiện $A \cup B$ đơn giản là

$$P(A \cup B) = P(A) + P(B).$$

Xác suất có điều kiện

Xác suất có điều kiện

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

là xác suất của A với giả thiết rằng B xảy ra. Vì vậy, khi tính xác suất của A , ta chỉ xét các kết quả cũng thuộc về B .

Với các tập ở trên,

$$P(A|B) = 1/3,$$

vì các kết quả của B là $\{1, 2, 3\}$, và một trong số chúng là số chẵn. Đây là xác suất để kết quả là số chẵn nếu ta biết rằng kết quả nằm trong khoảng $1 \dots 3$.

Giao

Dùng xác suất có điều kiện, xác suất của giao $A \cap B$ có thể được tính bằng công thức

$$P(A \cap B) = P(A)P(B|A).$$

Các sự kiện A và B là **độc lập** nếu

$$P(A|B) = P(A) \quad \text{và} \quad P(B|A) = P(B),$$

nghĩa là việc B xảy ra không làm thay đổi xác suất của A , và ngược lại. Trong trường hợp này, xác suất của giao là

$$P(A \cap B) = P(A)P(B).$$

Ví dụ, khi rút một lá bài từ bộ bài, các sự kiện

$$A = \text{"chất là chuồn"}$$

và

$$B = \text{"giá trị là bốn"}$$

là độc lập. Do đó sự kiện

$$A \cap B = \text{"lá bài là bốn chuồn"}$$

xảy ra với xác suất

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3 Biến ngẫu nhiên

Một **biến ngẫu nhiên** là một giá trị được sinh ra bởi một quá trình ngẫu nhiên. Ví dụ, khi gieo hai con xúc xắc, một biến ngẫu nhiên có thể là

$$X = \text{"tổng của các kết quả"}.$$

Chẳng hạn, nếu các kết quả là $[4, 6]$ (nghĩa là ta gieo được bốn trước rồi sáu sau), thì giá trị của X là 10.

Ta ký hiệu $P(X = x)$ là xác suất để giá trị của biến ngẫu nhiên X là x . Ví dụ, khi gieo hai con xúc xắc, $P(X = 10) = 3/36$, vì tổng số kết quả là 36 và có ba cách có thể để nhận được tổng 10: $[4, 6]$, $[5, 5]$ và $[6, 4]$.

Giá trị kỳ vọng

Giá trị kỳ vọng $E[X]$ biểu thị giá trị trung bình của một biến ngẫu nhiên X . Giá trị kỳ vọng có thể được tính bằng tổng

$$\sum_x P(X = x)x,$$

trong đó x chạy qua mọi giá trị có thể của X .

Ví dụ, khi gieo một con xúc xắc, kết quả kỳ vọng là

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Một tính chất hữu ích của giá trị kỳ vọng là **tính tuyến tính**. Điều này nghĩa là tổng $E[X_1 + X_2 + \dots + X_n]$ luôn bằng tổng $E[X_1] + E[X_2] + \dots + E[X_n]$. Công thức này vẫn đúng ngay cả khi các biến ngẫu nhiên phụ thuộc lẫn nhau.

Ví dụ, khi gieo hai con xúc xắc, tổng kỳ vọng là

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Hãy xét một bài toán trong đó n quả bóng được đặt ngẫu nhiên vào n hộp, và nhiệm vụ của ta là tính số hộp rỗng kỳ vọng. Mỗi quả bóng có xác suất như nhau để được đặt vào bất kỳ hộp nào. Ví dụ, nếu $n = 2$, các khả năng như sau:



Trong trường hợp này, số hộp rỗng kỳ vọng là

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

Trong trường hợp tổng quát, xác suất để một hộp cụ thể rỗng là

$$\left(\frac{n-1}{n}\right)^n,$$

vì không quả bóng nào được đặt vào hộp đó. Do đó, dùng tính tuyến tính, số hộp rỗng kỳ vọng là

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Phân phối

Phân phối của một biến ngẫu nhiên X cho biết xác suất của mỗi giá trị mà X có thể nhận. Phân phối gồm các giá trị $P(X = x)$. Ví dụ, khi gieo hai con xúc xắc, phân phối của tổng hai kết quả là:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Trong một **phân phối đều**, biến ngẫu nhiên X có n giá trị có thể $a, a + 1, \dots, b$ và xác suất của mỗi giá trị là $1/n$. Ví dụ, khi gieo một con xúc xắc, $a = 1, b = 6$ và $P(X = x) = 1/6$ với mỗi giá trị x .

Giá trị kỳ vọng của X trong phân phối đều là

$$E[X] = \frac{a+b}{2}.$$

Trong một **phân phối nhị thức**, có n lần thử được thực hiện và xác suất để một lần thử thành công là p . Biến ngẫu nhiên X đếm số lần thử thành công, và xác suất của một giá trị x là

$$P(X = x) = p^x(1-p)^{n-x} \binom{n}{x},$$

trong đó p^x và $(1-p)^{n-x}$ tương ứng với các lần thử thành công và không thành công, và $\binom{n}{x}$ là số cách ta có thể chọn thứ tự các lần thử.

Ví dụ, khi gieo một con xúc xắc mười lần, xác suất gieo được mặt sáu đúng ba lần là $(1/6)^3(5/6)^7 \binom{10}{3}$.

Giá trị kỳ vọng của X trong phân phối nhị thức là

$$E[X] = pn.$$

Trong một **phân phối hình học**, xác suất để một lần thử thành công là p , và ta tiếp tục cho đến khi thành công đầu tiên xảy ra. Biến ngẫu nhiên X đếm số lần thử cần thiết, và xác suất của một giá trị x là

$$P(X = x) = (1-p)^{x-1}p,$$

trong đó $(1-p)^{x-1}$ tương ứng với các lần thử không thành công và p tương ứng với lần thử thành công đầu tiên.

Ví dụ, nếu ta gieo xúc xắc cho đến khi ra mặt sáu, xác suất để số lần gieo đúng bằng 4 là $(5/6)^3 1/6$.

Giá trị kỳ vọng của X trong phân phối hình học là

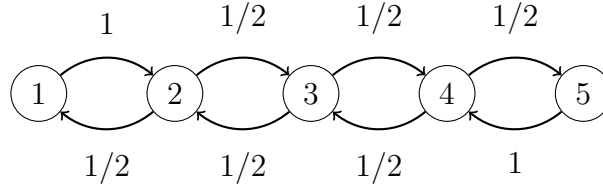
$$E[X] = \frac{1}{p}.$$

24.4 Xích Markov

Xích Markov (Markov chain) là một quá trình ngẫu nhiên gồm các trạng thái và các chuyển tiếp giữa chúng. Với mỗi trạng thái, ta biết các xác suất để chuyển sang những trạng thái khác. Một xích Markov có thể được biểu diễn bằng một đồ thị trong đó các đỉnh là trạng thái và các cạnh là chuyển tiếp.

Làm ví dụ, xét một bài toán trong đó ta đang ở tầng 1 của một tòa nhà có n tầng. Ở mỗi bước, ta đi ngẫu nhiên lên một tầng hoặc xuống một tầng, ngoại trừ việc ta luôn đi lên một tầng từ tầng 1 và đi xuống một tầng từ tầng n . Xác suất để ta ở tầng m sau k bước là bao nhiêu?

Trong bài toán này, mỗi tầng của tòa nhà tương ứng với một trạng thái trong xích Markov. Ví dụ, nếu $n = 5$, đồ thị như sau:



Phân phối xác suất của một xích Markov là một vector $[p_1, p_2, \dots, p_n]$, trong đó p_k là xác suất để trạng thái hiện tại là k . Công thức $p_1 + p_2 + \dots + p_n = 1$ luôn đúng.

Trong tình huống trên, phân phối ban đầu là $[1, 0, 0, 0, 0]$, vì ta luôn bắt đầu ở tầng 1. Phân phối tiếp theo là $[0, 1, 0, 0, 0]$, vì ta chỉ có thể đi từ tầng 1 lên tầng 2. Sau đó, ta có thể đi lên một tầng hoặc xuống một tầng, nên phân phối tiếp theo là $[1/2, 0, 1/2, 0, 0]$, v.v.

Một cách hiệu quả để mô phỏng bước đi trong xích Markov là dùng quy hoạch động. Ý tưởng là duy trì phân phối xác suất, và ở mỗi bước duyệt qua mọi khả năng ta có thể di chuyển. Dùng phương pháp này, ta có thể mô phỏng một bước đi dài m bước trong thời gian $O(n^2m)$.

Các chuyển tiếp của xích Markov cũng có thể được biểu diễn bằng một ma trận cập nhật phân phối xác suất. Trong tình huống trên, ma trận là

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

Khi nhân một phân phối xác suất với ma trận này, ta nhận được phân phối mới sau khi đi một bước. Ví dụ, ta có thể chuyển từ phân phối $[1, 0, 0, 0, 0]$ sang phân phối $[0, 1, 0, 0, 0]$ như sau:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Bằng cách tính lũy thừa ma trận hiệu quả, ta có thể tính phân phối sau m bước trong thời gian $O(n^3 \log m)$.

24.5 Thuật toán ngẫu nhiên

Đôi khi ta có thể dùng tính ngẫu nhiên để giải một bài toán, ngay cả khi bài toán không liên quan đến xác suất. Một **thuật toán ngẫu nhiên** là một thuật toán dựa trên ngẫu nhiên.

Một **thuật toán Monte Carlo** là một thuật toán ngẫu nhiên đôi khi có thể đưa ra đáp án sai. Để thuật toán như vậy hữu ích, xác suất cho đáp án sai cần nhỏ.

Một **thuật toán Las Vegas** là một thuật toán ngẫu nhiên luôn đưa ra đáp án đúng, nhưng thời gian chạy thay đổi ngẫu nhiên. Mục tiêu là thiết kế một thuật toán hiệu quả với xác suất cao.

Tiếp theo, ta sẽ xem ba bài toán ví dụ có thể được giải bằng ngẫu nhiên.

Thống kê thứ tự

Thống kê thứ tự (order statistic) thứ k của một mảng là phần tử ở vị trí k sau khi sắp xếp mảng theo thứ tự tăng dần. Ta có thể dễ dàng tính bất kỳ thống kê thứ tự nào trong thời gian $O(n \log n)$ bằng cách sắp xếp mảng trước, nhưng liệu có thật sự cần sắp xếp toàn bộ mảng chỉ để tìm một phần tử?

Hóa ra ta có thể tìm thống kê thứ tự bằng một thuật toán ngẫu nhiên mà không cần sắp xếp mảng. Thuật toán, gọi là **quickselect**¹, là một thuật toán Las Vegas: thời gian chạy của nó thường là $O(n)$ nhưng là $O(n^2)$ trong trường hợp xấu nhất.

Thuật toán chọn một phần tử ngẫu nhiên x của mảng, rồi chuyển các phần tử nhỏ hơn x sang phần trái của mảng, và tất cả các phần tử còn lại sang phần phải của mảng. Việc này mất thời gian $O(n)$ khi có n phần tử. Giả sử phần trái chứa a phần tử và phần phải chứa b phần tử. Nếu $a = k$, phần tử x là thống kê thứ tự thứ k . Ngược lại, nếu $a > k$, ta đệ quy tìm thống kê thứ tự thứ k trong phần trái, và nếu $a < k$, ta đệ quy tìm thống kê thứ tự thứ r trong phần phải, trong đó $r = k - a$. Việc tìm kiếm tiếp tục tương tự cho đến khi tìm được phần tử.

Khi mỗi phần tử x được chọn ngẫu nhiên, kích thước của mảng gần như giảm một nửa ở mỗi bước, nên độ phức tạp thời gian để tìm thống kê thứ tự thứ k xấp xỉ

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

Trường hợp xấu nhất của thuật toán vẫn cần thời gian $O(n^2)$, vì có thể x luôn được chọn sao cho nó là một trong các phần tử nhỏ nhất hoặc lớn nhất trong mảng và cần $O(n)$ bước. Tuy nhiên, xác suất để điều này xảy ra nhỏ đến mức trong thực tế gần như không gặp.

Kiểm chứng phép nhân ma trận

Bài toán tiếp theo là *kiểm chứng* xem $AB = C$ có đúng hay không khi A , B và C là các ma trận kích thước $n \times n$. Dĩ nhiên, ta có thể giải bài toán bằng cách tính lại tích AB (trong thời gian $O(n^3)$ bằng thuật toán cơ bản), nhưng ta có thể hy vọng rằng kiểm chứng đáp án sẽ dễ hơn tính lại từ đầu.

Hóa ra ta có thể giải bài toán bằng một thuật toán Monte Carlo² có độ phức tạp thời gian chỉ $O(n^2)$. Ý tưởng rất đơn giản: ta chọn một vector ngẫu nhiên X gồm n phần tử, rồi tính các ma trận ABX và CX . Nếu $ABX = CX$, ta báo rằng $AB = C$, ngược lại ta báo rằng $AB \neq C$.

¹Năm 1961, C. A. R. Hoare công bố hai thuật toán hiệu quả trung bình: **quicksort** [36] để sắp xếp mảng và **quickselect** [37] để tìm thống kê thứ tự.

²R. M. Freivalds công bố thuật toán này năm 1977 [26], và đôi khi nó được gọi là **Freivalds' algorithm**.

Độ phức tạp thời gian của thuật toán là $O(n^2)$, vì ta có thể tính các ma trận ABX và CX trong thời gian $O(n^2)$. Ta có thể tính ma trận ABX hiệu quả bằng biểu diễn $A(BX)$, nên chỉ cần hai phép nhân ma trận kích thước $n \times n$ và $n \times 1$.

Nhược điểm của thuật toán là có một xác suất nhỏ để thuật toán mắc lỗi khi báo rằng $AB = C$. Ví dụ,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

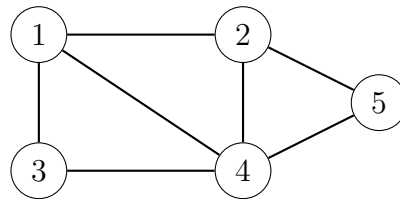
but

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

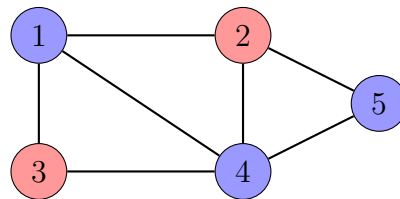
Tuy nhiên, trong thực tế, xác suất để thuật toán mắc lỗi là nhỏ, và ta có thể giảm xác suất này bằng cách kiểm chứng kết quả với nhiều vector ngẫu nhiên X trước khi báo rằng $AB = C$.

Tô màu đồ thị

Cho một đồ thị có n đỉnh và m cạnh, nhiệm vụ của ta là tìm một cách tô màu các đỉnh của đồ thị bằng hai màu sao cho với ít nhất $m/2$ cạnh, hai đầu mút có màu khác nhau. Ví dụ, trong đồ thị



một cách tô màu hợp lệ như sau:



Đồ thị trên có 7 cạnh, và với 5 cạnh trong số đó, hai đầu mút có màu khác nhau, nên cách tô màu là hợp lệ.

Bài toán có thể được giải bằng một thuật toán Las Vegas sinh các cách tô màu ngẫu nhiên cho đến khi tìm được một cách tô màu hợp lệ. Trong một cách tô màu ngẫu nhiên, màu của mỗi đỉnh được chọn độc lập sao cho xác suất của cả hai màu đều là $1/2$.

Trong một cách tô màu ngẫu nhiên, xác suất để hai đầu mút của một cạnh có màu khác nhau là $1/2$. Do đó, số cạnh kỳ vọng có hai đầu mút khác màu là $m/2$. Vì một cách tô màu ngẫu nhiên được kỳ vọng là hợp lệ, trong thực tế ta sẽ nhanh chóng tìm được một cách tô màu hợp lệ.

Chương 25

Lý thuyết trò chơi

Trong chương này, ta sẽ tập trung vào các trò chơi hai người không chứa yếu tố ngẫu nhiên. Mục tiêu của ta là tìm một chiến lược mà ta có thể theo để thắng trò chơi bất kể đối thủ làm gì, nếu một chiến lược như vậy tồn tại.

Hóa ra có một chiến lược tổng quát cho các trò chơi như vậy, và ta có thể phân tích chúng bằng **lý thuyết nim** (nim theory). Trước hết, ta sẽ phân tích các trò chơi đơn giản trong đó người chơi lấy que khỏi các đồng, rồi sau đó tổng quát hóa chiến lược dùng trong các trò chơi đó sang những trò chơi khác.

25.1 Trạng thái trò chơi

Hãy xét một trò chơi ban đầu có một đồng gồm n que. Hai người chơi A và B đi luân phiên, và người chơi A bắt đầu. Ở mỗi lượt, người chơi phải lấy 1, 2 hoặc 3 que khỏi đồng, và người chơi lấy que cuối cùng sẽ thắng trò chơi.

Ví dụ, nếu $n = 10$, trò chơi có thể diễn ra như sau:

- Người chơi A lấy 2 que (còn 8 que).
- Người chơi B lấy 3 que (còn 5 que).
- Người chơi A lấy 1 que (còn 4 que).
- Người chơi B lấy 2 que (còn 2 que).
- Người chơi A lấy 2 que và thắng.

Trò chơi này gồm các trạng thái $0, 1, 2, \dots, n$, trong đó số của trạng thái tương ứng với số que còn lại.

Trạng thái thắng và thua

Một **trạng thái thắng** là trạng thái mà người chơi sẽ thắng trò chơi nếu họ chơi tối ưu, và một **trạng thái thua** là trạng thái mà người chơi sẽ thua nếu đối thủ chơi tối ưu. Hóa ra ta có thể phân loại mọi trạng thái của một trò chơi sao cho mỗi trạng thái hoặc là trạng thái thắng, hoặc là trạng thái thua.

Trong trò chơi trên, trạng thái 0 rõ ràng là trạng thái thua, vì người chơi không thể đi nước nào. Các trạng thái 1, 2 và 3 là trạng thái thắng, vì ta có thể lấy 1, 2 hoặc 3 que và thắng trò chơi. Ngược lại, trạng thái 4 là trạng thái thua, vì mọi nước đi đều dẫn đến một trạng thái thắng cho đối thủ.

Tổng quát hơn, nếu có một nước đi dẫn từ trạng thái hiện tại đến một trạng thái thua, thì trạng thái hiện tại là trạng thái thắng, ngược lại trạng thái hiện tại là trạng thái thua. Dựa trên nhận xét này, ta có thể phân loại mọi trạng thái của một trò chơi, bắt đầu từ các trạng thái thua nơi không có nước đi hợp lệ.

Các trạng thái $0 \dots 15$ của trò chơi trên có thể được phân loại như sau (W ký hiệu trạng thái thắng và L ký hiệu trạng thái thua):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W

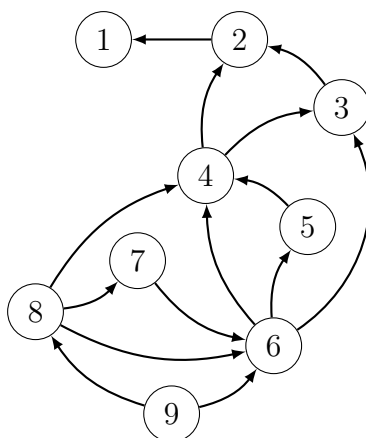
Trò chơi này dễ phân tích: một trạng thái k là trạng thái thua nếu k chia hết cho 4, và ngược lại nó là trạng thái thắng. Cách chơi tối ưu là luôn chọn một nước đi sao cho sau đó số que trong đồng chia hết cho 4. Cuối cùng, không còn que nào và đối thủ đã thua.

Dĩ nhiên, chiến lược này đòi hỏi rằng số que *không* chia hết cho 4 khi đến lượt ta. Nếu nó chia hết cho 4, ta không thể làm gì, và đối thủ sẽ thắng trò chơi nếu họ chơi tối ưu.

Đồ thị trạng thái

Bây giờ hãy xét một trò chơi que khác, trong đó ở mỗi trạng thái k , ta được phép lấy bất kỳ số que x nào sao cho x nhỏ hơn k và chia hết k . Ví dụ, ở trạng thái 8 ta có thể lấy 1, 2 hoặc 4 que, nhưng ở trạng thái 7 nước đi duy nhất được phép là lấy 1 que.

Hình sau biểu diễn các trạng thái $1 \dots 9$ của trò chơi dưới dạng một **đồ thị trạng thái**, trong đó các nút là trạng thái và các cạnh là nước đi giữa chúng:



Trạng thái cuối cùng trong trò chơi này luôn là trạng thái 1, đây là trạng thái thua, vì không có nước đi hợp lệ. Việc phân loại các trạng thái $1 \dots 9$ như sau:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Điều đáng ngạc nhiên là trong trò chơi này, mọi trạng thái mang số chẵn đều là trạng thái thắng, và mọi trạng thái mang số lẻ đều là trạng thái thua.

25.2 Trò chơi nim

Trò chơi nim là một trò chơi đơn giản có vai trò quan trọng trong lý thuyết trò chơi, vì nhiều trò chơi khác có thể được chơi bằng cùng một chiến lược. Trước hết, ta tập trung vào nim, rồi sau đó tổng quát hóa chiến lược sang các trò chơi khác.

Trong nim có n đồng, và mỗi đồng chứa một số que. Người chơi đi luân phiên, và ở mỗi lượt, người chơi chọn một đồng vẫn còn que rồi lấy đi bất kỳ số que nào khỏi đồng đó. Người thắng là người lấy que cuối cùng.

Các trạng thái trong nim có dạng $[x_1, x_2, \dots, x_n]$, trong đó x_k biểu thị số que trong đồng k . Ví dụ, $[10, 12, 5]$ là một trò chơi có ba đồng lần lượt gồm 10, 12 và 5 que. Trạng thái $[0, 0, \dots, 0]$ là trạng thái thua, vì không thể lấy thêm que nào, và đây luôn là trạng thái cuối cùng.

Phân tích

Hóa ra ta có thể dễ dàng phân loại bất kỳ trạng thái nim nào bằng cách tính **tổng nim** (nim sum) $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$, trong đó $\oplus$ là phép xor¹. Các trạng thái có tổng nim bằng 0 là trạng thái thua, và mọi trạng thái còn lại là trạng thái thắng. Ví dụ, tổng nim của $[10, 12, 5]$ là $10 \oplus 12 \oplus 5 = 3$, nên trạng thái này là trạng thái thắng.

Nhưng tổng nim liên quan thế nào đến trò chơi nim? Ta có thể giải thích điều này bằng cách xem tổng nim thay đổi ra sao khi trạng thái nim thay đổi.

Trạng thái thua: Trạng thái cuối cùng $[0, 0, \dots, 0]$ là trạng thái thua, và tổng nim của nó bằng 0, đúng như mong đợi. Ở các trạng thái thua khác, mọi nước đi đều dẫn đến một trạng thái thắng, vì khi một giá trị x_k thay đổi, tổng nim cũng thay đổi, nên tổng nim sẽ khác 0 sau nước đi.

Trạng thái thắng: Ta có thể đi đến một trạng thái thua nếu có một đồng k sao cho $x_k \oplus s < x_k$. Trong trường hợp này, ta có thể lấy que khỏi đồng k để nó còn $x_k \oplus s$ que, và điều đó sẽ dẫn đến một trạng thái thua. Luôn tồn tại một đồng như vậy, trong đó x_k có bit 1 ở vị trí của bit 1 ngoài cùng bên trái của s .

Làm ví dụ, xét trạng thái $[10, 12, 5]$. Trạng thái này là trạng thái thắng, vì tổng nim của nó là 3. Do đó phải tồn tại một nước đi dẫn đến trạng thái thua. Tiếp theo ta sẽ tìm nước đi đó.

Tổng nim của trạng thái là:

¹Chiến lược tối ưu cho nim được C. L. Bouton công bố năm 1901 [10].

10	1010
12	1100
5	0101
3	0011

Trong trường hợp này, đồng có 10 que là đồng duy nhất có bit 1 ở vị trí của bit 1 ngoài cùng bên trái của tổng nim:

10	10 <u>1</u> 0
12	1100
5	0101
3	00 <u>1</u> 1

Kích thước mới của đồng phải là $10 \oplus 3 = 9$, nên ta chỉ lấy một que. Sau đó, trạng thái sẽ là $[9, 12, 5]$, đây là một trạng thái thua:

9	1001
12	1100
5	0101
0	0000

Trò chơi misère

Trong một **trò chơi misère**, mục tiêu của trò chơi bị đảo ngược, nên người chơi lấy que cuối cùng sẽ thua trò chơi. Hóa ra trò chơi misère nim có thể được chơi tối ưu gần giống như trò chơi nim chuẩn.

Ý tưởng là ban đầu chơi misère như trò chơi chuẩn, nhưng đổi chiến lược ở cuối trò chơi. Chiến lược mới sẽ được áp dụng trong tình huống mà sau nước đi tiếp theo, mỗi đồng sẽ chứa nhiều nhất một que.

Trong trò chơi chuẩn, ta nên chọn một nước đi sao cho sau đó có số chẵn đồng gồm một que. Tuy nhiên, trong trò chơi misère, ta chọn một nước đi sao cho có số lẻ đồng gồm một que.

Chiến lược này đúng vì trạng thái mà chiến lược thay đổi luôn xuất hiện trong trò chơi, và trạng thái này là trạng thái thắng, vì nó chứa đúng một đồng có nhiều hơn một que nên tổng nim khác 0.

25.3 Định lý Sprague–Grundy

Định lý Sprague–Grundy² tổng quát hóa chiến lược dùng trong nim cho mọi trò chơi thỏa mãn các yêu cầu sau:

- Có hai người chơi đi luân phiên.
- Trò chơi gồm các trạng thái, và các nước đi có thể trong một trạng thái không phụ thuộc vào việc đến lượt ai.
- Trò chơi kết thúc khi một người chơi không thể đi nước nào.

²Định lý này được R. Sprague [61] và P. M. Grundy [31] phát hiện độc lập.

- Trò chơi chắc chắn sẽ kết thúc sau một số hữu hạn bước.
- Người chơi có đầy đủ thông tin về các trạng thái và nước đi được phép, và trò chơi không có ngẫu nhiên.

Ý tưởng là tính cho mỗi trạng thái trò chơi một số Grundy tương ứng với số que trong một đồng nim. Khi biết số Grundy của mọi trạng thái, ta có thể chơi trò chơi như trò chơi nim.

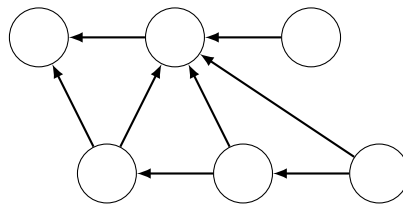
Số Grundy

Số Grundy của một trạng thái trò chơi là

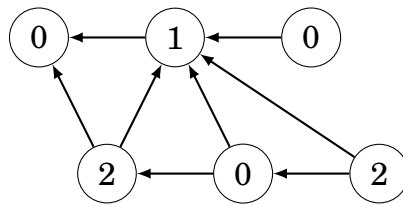
$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

trong đó $g_1, g_2, \dots, g_n$ là các số Grundy của những trạng thái mà ta có thể đi đến, và hàm mex cho số không âm nhỏ nhất không nằm trong tập. Ví dụ, $\text{mex}(\{0, 1, 3\}) = 2$. Nếu không có nước đi nào trong một trạng thái, số Grundy của nó là 0, vì $\text{mex}(\emptyset) = 0$.

Ví dụ, trong đồ thị trạng thái



các số Grundy như sau:



Số Grundy của một trạng thái thua là 0, và số Grundy của một trạng thái thắng là một số dương.

Số Grundy của một trạng thái tương ứng với số que trong một đồng nim. Nếu số Grundy bằng 0, ta chỉ có thể đi đến các trạng thái có số Grundy dương, và nếu số Grundy là $x > 0$, ta có thể đi đến các trạng thái có số Grundy chứa mọi số $0, 1, \dots, x - 1$.

Làm ví dụ, xét một trò chơi trong đó người chơi di chuyển một quân trong mê cung. Mỗi ô trong mê cung hoặc là ô trống hoặc là tường. Ở mỗi lượt, người chơi phải di chuyển quân đó một số bước sang trái hoặc lên trên. Người thắng là người chơi đi nước cuối cùng.

Hình sau cho thấy một trạng thái ban đầu có thể có của trò chơi, trong đó @ ký hiệu quân và * ký hiệu một ô mà quân có thể đi đến.

				*
				*
*	*	*	*	@

Các trạng thái của trò chơi là mọi ô trống trong mê cung. Trong mê cung trên, các số Grundy như sau:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Do đó, mỗi trạng thái của trò chơi mê cung tương ứng với một đồng trong trò chơi nim. Ví dụ, số Grundy của ô dưới bên phải là 2, nên đó là một trạng thái thắng. Ta có thể đi đến một trạng thái thua và thắng trò chơi bằng cách di chuyển bốn bước sang trái hoặc hai bước lên trên.

Lưu ý rằng khác với trò chơi nim gốc, có thể đi đến một trạng thái có số Grundy lớn hơn số Grundy của trạng thái hiện tại. Tuy nhiên, đối thủ luôn có thể chọn một nước đi triệt tiêu nước đi đó, nên không thể thoát khỏi một trạng thái thua.

Trò chơi con

Tiếp theo, ta giả sử trò chơi gồm các trò chơi con, và ở mỗi lượt, người chơi trước tiên chọn một trò chơi con rồi đi một nước trong trò chơi con đó. Trò chơi kết thúc khi không thể đi nước nào trong bất kỳ trò chơi con nào.

Trong trường hợp này, số Grundy của trò chơi là tổng nim của các số Grundy của các trò chơi con. Trò chơi có thể được chơi như một trò chơi nim bằng cách tính tất cả số Grundy của các trò chơi con rồi tính tổng nim của chúng.

Làm ví dụ, xét một trò chơi gồm ba mê cung. Trong trò chơi này, ở mỗi lượt, người chơi chọn một trong các mê cung rồi di chuyển quân trong mê cung đó. Giả sử trạng thái ban đầu của trò chơi như sau:

Các số Grundy của các mê cung như sau:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

0	1	2	3	
1	0		0	1
2		0	1	2
3		1	2	0
4	0	2	5	3

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

Trong trạng thái ban đầu, tổng nim của các số Grundy là $2 \oplus 3 \oplus 3 = 2$, nên người chơi đầu tiên có thể thắng trò chơi. Một nước đi tối ưu là di chuyển hai bước lên trên trong mê cung thứ nhất, tạo ra tổng nim $0 \oplus 3 \oplus 3 = 0$.

Trò chơi Grundy

Đôi khi một nước đi trong trò chơi chia trò chơi thành các trò chơi con độc lập với nhau. Trong trường hợp này, số Grundy của trò chơi là

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

trong đó n là số nước đi có thể và

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

trong đó nước đi k sinh ra các trò chơi con có số Grundy $a_{k,1}, a_{k,2}, \dots, a_{k,m}$.

Một ví dụ của dạng trò chơi này là **Grundy's game**. Ban đầu có một đồng duy nhất chứa n que. Ở mỗi lượt, người chơi chọn một đồng và chia nó thành hai đồng không rỗng sao cho hai đồng có kích thước khác nhau. Người chơi đi nước cuối cùng sẽ thắng trò chơi.

Gọi $f(n)$ là số Grundy của một đồng chứa n que. Số Grundy có thể được tính bằng cách duyệt mọi cách chia đồng thành hai đồng. Ví dụ, khi $n = 8$, các khả năng là $1 + 7$, $2 + 6$ và $3 + 5$, nên

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

Trong trò chơi này, giá trị của $f(n)$ dựa trên các giá trị của $f(1), \dots, f(n-1)$. Các trường hợp cơ sở là $f(1) = f(2) = 0$, vì không thể chia các đồng có 1 và 2 que. Các số Grundy đầu tiên là:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

Số Grundy với $n = 8$ là 2, nên có thể thắng trò chơi. Nước đi thắng là tạo ra các đồng $1 + 7$, vì $f(1) \oplus f(7) = 0$.

Chương 26

Thuật toán chuỗi

Chương này bàn về các thuật toán hiệu quả để xử lý chuỗi. Nhiều bài toán chuỗi có thể được giải dễ dàng trong thời gian $O(n^2)$, nhưng thách thức là tìm các thuật toán chạy trong thời gian $O(n)$ hoặc $O(n \log n)$.

Ví dụ, một bài toán xử lý chuỗi cơ bản là bài toán **khớp mẫu** (pattern matching): cho một chuỗi độ dài n và một mẫu độ dài m , nhiệm vụ của ta là tìm các vị trí xuất hiện của mẫu trong chuỗi. Chẳng hạn, mẫu ABC xuất hiện hai lần trong chuỗi ABABCABABC.

Bài toán khớp mẫu có thể được giải dễ dàng trong thời gian $O(nm)$ bằng một thuật toán vét cạn thử mọi vị trí mà mẫu có thể xuất hiện trong chuỗi. Tuy nhiên, trong chương này, ta sẽ thấy rằng có những thuật toán hiệu quả hơn chỉ cần thời gian $O(n + m)$.

26.1 Thuật ngữ chuỗi

Trong suốt chương này, ta giả sử rằng chuỗi dùng chỉ số bắt đầu từ 0. Do đó, một chuỗi s độ dài n gồm các ký tự $s[0], s[1], \dots, s[n-1]$. Tập các ký tự có thể xuất hiện trong chuỗi được gọi là **bảng chữ cái**. Ví dụ, bảng chữ cái $\{A, B, \dots, Z\}$ gồm các chữ cái in hoa tiếng Anh.

Chuỗi con liên tiếp (substring) là một dãy các ký tự liên tiếp trong một chuỗi. Ta dùng ký hiệu $s[a \dots b]$ để chỉ một chuỗi con của s bắt đầu tại vị trí a và kết thúc tại vị trí b . Một chuỗi độ dài n có $n(n+1)/2$ chuỗi con liên tiếp. Ví dụ, các chuỗi con liên tiếp của ABCD là A, B, C, D, AB, BC, CD, ABC, BCD và ABCD.

Dãy con (subsequence) là một dãy các ký tự (không nhất thiết liên tiếp) trong một chuỗi theo thứ tự ban đầu của chúng. Một chuỗi độ dài n có $2^n - 1$ dãy con. Ví dụ, các dãy con của ABCD là A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD và ABCD.

Tiền tố (prefix) là một chuỗi con bắt đầu từ đầu của một chuỗi, và **hậu tố** (suffix) là một chuỗi con kết thúc ở cuối của một chuỗi. Ví dụ, các tiền tố của ABCD là A, AB, ABC và ABCD, và các hậu tố của ABCD là D, CD, BCD và ABCD.

Phép xoay có thể được tạo bằng cách chuyển các ký tự của một chuỗi

từng ký tự một từ đầu ra cuối (hoặc ngược lại). Ví dụ, các phép xoay của ABCD là ABCD, BCDA, CDAB và DABC.

Chu kỳ là một tiền tố của một chuỗi sao cho chuỗi có thể được tạo bằng cách lặp lại chu kỳ đó. Lặp cuối có thể không đầy đủ và chỉ chứa một tiền tố của chu kỳ. Ví dụ, chu kỳ ngắn nhất của ABCABCA là ABC.

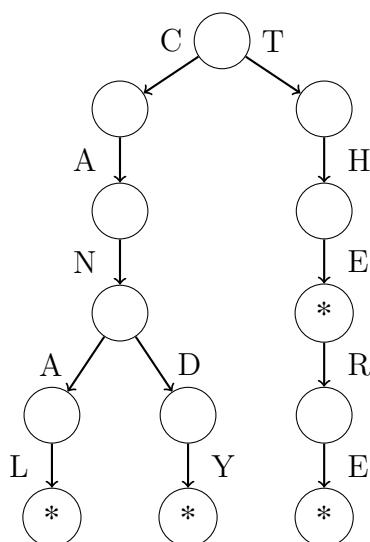
Biên (border) là một chuỗi vừa là tiền tố vừa là hậu tố của một chuỗi. Ví dụ, các biên của ABACABA là A, ABA và ABACABA.

Các chuỗi được so sánh bằng **thứ tự từ điển** (tương ứng với thứ tự bảng chữ cái). Điều này nghĩa là $x < y$ nếu $x \neq y$ và x là tiền tố của y , hoặc tồn tại một vị trí k sao cho $x[i] = y[i]$ khi $i < k$ và $x[k] < y[k]$.

26.2 Cấu trúc trie

Trie là một cây có gốc lưu một tập các chuỗi. Mỗi chuỗi trong tập được lưu dưới dạng một chuỗi các ký tự bắt đầu từ gốc. Nếu hai chuỗi có chung tiền tố, chúng cũng có chung một đoạn đường trong cây.

Ví dụ, xét trie sau:



Trie này tương ứng với tập {CANAL, CANDY, THE, THERE}. Ký tự * trong một nút nghĩa là có một chuỗi trong tập kết thúc tại nút đó. Ký tự như vậy là cần thiết, vì một chuỗi có thể là tiền tố của một chuỗi khác. Ví dụ, trong trie trên, THE là tiền tố của THERE.

Ta có thể kiểm tra trong thời gian $O(n)$ xem một trie có chứa một chuỗi độ dài n hay không, vì ta có thể đi theo đường bắt đầu từ nút gốc. Ta cũng có thể thêm một chuỗi độ dài n vào trie trong thời gian $O(n)$ bằng cách trước hết đi theo đường hiện có rồi thêm các nút mới vào trie nếu cần.

Dùng trie, ta có thể tìm tiền tố dài nhất của một chuỗi cho trước sao cho tiền tố đó thuộc về tập. Hơn nữa, bằng cách lưu thêm thông tin trong mỗi nút, ta có thể tính số chuỗi thuộc về tập và có một chuỗi cho trước làm tiền tố.

Một trie có thể được lưu trong một mảng


```
int trie[N][A];
```

trong đó N là số nút tối đa (tổng độ dài tối đa của các chuỗi trong tập) và A là kích thước bảng chữ cái. Các nút của trie được đánh số $0, 1, 2, \dots$ sao cho số của gốc là 0, và $\text{trie}[s][c]$ là nút tiếp theo trên đường khi ta đi từ nút s bằng ký tự c .

26.3 Băm chuỗi

Băm chuỗi (String hashing) là một kỹ thuật cho phép ta kiểm tra hiệu quả xem hai chuỗi có bằng nhau hay không¹. Ý tưởng của băm chuỗi là so sánh giá trị băm của các chuỗi thay vì từng ký tự riêng lẻ.

Tính giá trị băm

Giá trị băm của một chuỗi là một số được tính từ các ký tự của chuỗi. Nếu hai chuỗi giống nhau, giá trị băm của chúng cũng giống nhau, điều này cho phép so sánh chuỗi dựa trên giá trị băm.

Một cách thông dụng để cài đặt băm chuỗi là **băm đa thức** (polynomial hashing), nghĩa là giá trị băm của một chuỗi s độ dài n là

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

trong đó $s[0], s[1], \dots, s[n-1]$ được hiểu là mã của các ký tự trong s , và A cùng B là các hằng số được chọn trước.

Ví dụ, mã của các ký tự trong ALLEY là:

A	L	L	E	Y
65	76	76	69	89

Do đó, nếu $A = 3$ và $B = 97$, giá trị băm của ALLEY là

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

Tiền xử lý

Dùng băm đa thức, ta có thể tính giá trị băm của bất kỳ chuỗi con liên tiếp nào của chuỗi s trong thời gian $O(1)$ sau bước tiền xử lý $O(n)$. Ý tưởng là xây dựng một mảng h sao cho $h[k]$ chứa giá trị băm của tiền tố $s[0 \dots k]$. Các giá trị của mảng có thể được tính đệ quy như sau:

$$\begin{aligned} h[0] &= s[0] \\ h[k] &= (h[k-1]A + s[k]) \bmod B \end{aligned}$$

¹Kỹ thuật này được phổ biến bởi thuật toán khớp mẫu Karp–Rabin [42].

Ngoài ra, ta xây dựng một mảng p trong đó $p[k] = A^k \bmod B$:

$$\begin{aligned} p[0] &= 1 \\ p[k] &= (p[k-1]A) \bmod B. \end{aligned}$$

Việc xây dựng các mảng này mất thời gian $O(n)$. Sau đó, giá trị băm của bất kỳ chuỗi con liên tiếp nào $s[a \dots b]$ có thể được tính trong thời gian $O(1)$ bằng công thức

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

với giả thiết rằng $a > 0$. Nếu $a = 0$, giá trị băm đơn giản là $h[b]$.

Dùng giá trị băm

Ta có thể so sánh chuỗi hiệu quả bằng giá trị băm. Thay vì so sánh từng ký tự riêng lẻ của các chuỗi, ý tưởng là so sánh giá trị băm của chúng. Nếu các giá trị băm bằng nhau, các chuỗi *có lẽ* bằng nhau, và nếu các giá trị băm khác nhau, các chuỗi *chắc chắn* khác nhau.

Dùng băm, ta thường có thể làm cho thuật toán vết cặn trở nên hiệu quả. Làm ví dụ, xét bài toán khớp mẫu: cho một chuỗi s và một mẫu p , tìm các vị trí mà p xuất hiện trong s . Một thuật toán vết cặn duyệt qua mọi vị trí mà p có thể xuất hiện và so sánh các chuỗi từng ký tự một. Độ phức tạp thời gian của thuật toán như vậy là $O(n^2)$.

Ta có thể làm thuật toán vết cặn hiệu quả hơn bằng cách dùng băm, vì thuật toán so sánh các chuỗi con liên tiếp. Dùng băm, mỗi phép so sánh chỉ mất thời gian $O(1)$, vì chỉ các giá trị băm của chuỗi con được so sánh. Kết quả là một thuật toán có độ phức tạp thời gian $O(n)$, đây là độ phức tạp tốt nhất có thể cho bài toán này.

Bằng cách kết hợp băm và *tìm kiếm nhị phân*, ta cũng có thể tìm thứ tự từ điển của hai chuỗi trong thời gian logarit. Điều này có thể được thực hiện bằng cách tính độ dài của tiền tố chung của hai chuỗi bằng tìm kiếm nhị phân. Khi đã biết độ dài của tiền tố chung, ta chỉ cần kiểm tra ký tự tiếp theo sau tiền tố, vì ký tự này quyết định thứ tự của các chuỗi.

Va chạm và tham số

Một rủi ro rõ ràng khi so sánh giá trị băm là **va chạm** (collision), nghĩa là hai chuỗi có nội dung khác nhau nhưng giá trị băm bằng nhau. Trong trường hợp này, một thuật toán dựa vào giá trị băm sẽ kết luận rằng hai chuỗi bằng nhau, nhưng thực tế thì không, và thuật toán có thể cho kết quả sai.

Va chạm luôn có thể xảy ra, vì số chuỗi khác nhau lớn hơn số giá trị băm khác nhau. Tuy nhiên, xác suất va chạm nhỏ nếu các hằng số A và B được chọn cẩn thận. Một cách thường dùng là chọn các hằng ngẫu nhiên gần 10^9 , ví dụ như sau:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Dùng các hằng như vậy, kiểu long long có thể được dùng khi tính giá trị băm, vì các tích AB và BB sẽ vừa trong long long. Nhưng liệu có khoảng 10^9 giá trị băm khác nhau đã đủ chưa?

Hãy xét ba tình huống có thể dùng băm:

Tình huống 1: Hai chuỗi x và y được so sánh với nhau. Xác suất va chạm là $1/B$ nếu giả sử rằng mọi giá trị băm có xác suất như nhau.

Tình huống 2: Một chuỗi x được so sánh với các chuỗi $y_1, y_2, \dots, y_n$. Xác suất có một hoặc nhiều va chạm là

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

Tình huống 3: Mọi cặp chuỗi $x_1, x_2, \dots, x_n$ được so sánh với nhau. Xác suất có một hoặc nhiều va chạm là

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Bảng sau cho thấy các xác suất va chạm khi $n = 10^6$ và giá trị của B thay đổi:

hằng B	tình huống 1	tình huống 2	tình huống 3
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

Bảng cho thấy trong tình huống 1, xác suất va chạm là không đáng kể khi $B \approx 10^9$. Trong tình huống 2, va chạm có thể xảy ra nhưng xác suất vẫn khá nhỏ. Tuy nhiên, trong tình huống 3, tình hình rất khác: va chạm gần như luôn xảy ra khi $B \approx 10^9$.

Hiện tượng trong tình huống 3 được gọi là **ngịch lý sinh nhật**: nếu có n người trong một phòng, xác suất để *một vài* cặp người có cùng ngày sinh là lớn ngay cả khi n khá nhỏ. Tương tự trong băm, khi mọi giá trị băm được so sánh với nhau, xác suất để có hai giá trị băm bằng nhau là lớn.

Ta có thể làm xác suất va chạm nhỏ hơn bằng cách tính *nhiều* giá trị băm với các tham số khác nhau. Khả năng va chạm xảy ra ở tất cả các giá trị băm cùng lúc là thấp. Ví dụ, hai giá trị băm với tham số $B \approx 10^9$ tương ứng với một giá trị băm có tham số $B \approx 10^{18}$, làm cho xác suất va chạm rất nhỏ.

Một số người dùng các hằng $B = 2^{32}$ và $B = 2^{64}$, điều này tiện lợi, vì các phép toán với số nguyên 32 và 64 bit được tính modulo 2^{32} và 2^{64} . Tuy nhiên, đây *không* phải là lựa chọn tốt, vì có thể xây dựng dữ liệu vào luôn sinh ra va chạm khi dùng các hằng có dạng 2^x [51].

26.4 Thuật toán Z

Mảng Z z của một chuỗi s độ dài n chứa, với mỗi $k = 0, 1, \dots, n - 1$, độ dài của chuỗi con dài nhất của s bắt đầu tại vị trí k và là tiền tố của s . Do đó, $z[k] = p$ cho biết rằng $s[0 \dots p - 1]$ bằng $s[k \dots k + p - 1]$. Nhiều bài toán xử lý chuỗi có thể được giải hiệu quả bằng mảng Z.

Ví dụ, mảng Z của ACBACDACBACBACDA như sau:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Trong trường hợp này, chẳng hạn, $z[6] = 5$, vì chuỗi con ACBAC độ dài 5 là một tiền tố của s , nhưng chuỗi con ACBACB độ dài 6 không phải là tiền tố của s .

Mô tả thuật toán

Tiếp theo ta mô tả một thuật toán, gọi là **thuật toán Z** (Z-algorithm)², xây dựng mảng Z hiệu quả trong thời gian $O(n)$. Thuật toán tính các giá trị của mảng Z từ trái sang phải, vừa dùng thông tin đã lưu trong mảng Z, vừa so sánh chuỗi con từng ký tự một.

Để tính các giá trị mảng Z hiệu quả, thuật toán duy trì một đoạn $[x, y]$ sao cho $s[x \dots y]$ là tiền tố của s và y lớn nhất có thể. Vì ta biết rằng $s[0 \dots y - x]$ và $s[x \dots y]$ bằng nhau, ta có thể dùng thông tin này khi tính các giá trị Z cho các vị trí $x + 1, x + 2, \dots, y$.

Ở mỗi vị trí k , trước tiên ta kiểm tra giá trị $z[k - x]$. Nếu $k + z[k - x] < y$, ta biết rằng $z[k] = z[k - x]$. Tuy nhiên, nếu $k + z[k - x] \geq y$, $s[0 \dots y - k]$ bằng $s[k \dots y]$, và để xác định giá trị của $z[k]$ ta cần so sánh các chuỗi con từng ký tự một. Dù vậy, thuật toán vẫn chạy trong thời gian $O(n)$, vì ta bắt đầu so sánh tại các vị trí $y - k + 1$ và $y + 1$.

Ví dụ, hãy xây dựng mảng Z sau:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Sau khi tính giá trị $z[6] = 5$, đoạn $[x, y]$ hiện tại là $[6, 10]$:

						$\overbrace{\hspace{10em}}^{x \qquad \qquad \qquad y}$									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?

²Thuật toán Z được trình bày trong [32] như phương pháp đơn giản nhất đã biết cho khớp mẫu tuyến tính, và ý tưởng gốc được quy cho [50].

Bây giờ ta có thể tính các giá trị mảng Z tiếp theo một cách hiệu quả, vì ta biết rằng $s[0 \dots 4]$ và $s[6 \dots 10]$ bằng nhau. Trước hết, vì $z[1] = z[2] = 0$, ta biết ngay rằng $z[7] = z[8] = 0$:

						x			y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	



Tiếp theo, vì $z[3] = 2$, ta biết rằng $z[9] \geq 2$:

						x			y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
−	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	



Tuy nhiên, ta không có thông tin về chuỗi sau vị trí 10, nên cần so sánh các chuỗi con từng ký tự một:

						x			y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
−	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	

Hóa ra $z[9] = 7$, nên đoạn $[x, y]$ mới là $[9, 15]$:

									x			y			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
−	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?

Sau đó, mọi giá trị mảng Z còn lại có thể được xác định bằng thông tin đã lưu trong mảng Z :

									x			y				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
—	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1	

Dùng mảng Z

Việc chọn dùng băm chuỗi hay thuật toán Z thường là vấn đề sở thích. Khác với băm, thuật toán Z luôn đúng và không có rủi ro va chạm. Mặt khác, thuật toán Z khó cài đặt hơn và một số bài toán chỉ có thể được giải bằng băm.

Làm ví dụ, xét lại bài toán khớp mẫu, trong đó nhiệm vụ của ta là tìm các vị trí xuất hiện của một mẫu p trong một chuỗi s . Ta đã giải bài toán này hiệu quả bằng băm chuỗi, nhưng thuật toán Z cung cấp một cách khác để giải bài toán.

Một ý tưởng thường dùng trong xử lý chuỗi là xây dựng một chuỗi gồm nhiều chuỗi được ngăn cách bởi các ký tự đặc biệt. Trong bài toán này, ta có thể xây dựng chuỗi $p\#s$, trong đó p và s được ngăn cách bởi một ký tự đặc biệt $\#$ không xuất hiện trong các chuỗi. Mảng Z của $p\#s$ cho ta biết các vị trí mà p xuất hiện trong s , vì các vị trí đó chứa độ dài của p .

Ví dụ, nếu $s = \text{HATTIVATTI}$ và $p = \text{ATT}$, mảng Z như sau:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	T	T	#	H	A	T	T	I	V	A	T	T	I
–	0	0	0	0	3	0	0	0	0	3	0	0	0

Các vị trí 5 và 10 chứa giá trị 3, nghĩa là mẫu ATT xuất hiện tại các vị trí tương ứng trong HATTIVATTI.

Độ phức tạp thời gian của thuật toán thu được là tuyến tính, vì chỉ cần xây dựng mảng Z và duyệt qua các giá trị của nó.

Cài đặt

Dưới đây là một cài đặt ngắn của thuật toán Z trả về một vector tương ứng với mảng Z.

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0, min(z[i-x], y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```

Chương 27

Thuật toán căn bậc hai

Một **thuật toán căn bậc hai** là một thuật toán có căn bậc hai trong độ phức tạp thời gian. Căn bậc hai có thể được xem như “logarit của người nghèo”: độ phức tạp $O(\sqrt{n})$ tốt hơn $O(n)$ nhưng kém hơn $O(\log n)$. Dù vậy, nhiều thuật toán căn bậc hai rất nhanh và dùng tốt trong thực tế.

Ví dụ, xét bài toán xây dựng một cấu trúc dữ liệu hỗ trợ hai thao tác trên một mảng: thay đổi một phần tử ở vị trí cho trước và tính tổng các phần tử trong một đoạn cho trước. Trước đây ta đã giải bài toán này bằng cây Fenwick và cây phân đoạn, vốn hỗ trợ cả hai thao tác trong thời gian $O(\log n)$. Tuy nhiên, bây giờ ta sẽ giải bài toán theo một cách khác bằng cấu trúc căn bậc hai, cho phép thay đổi phần tử trong thời gian $O(1)$ và tính tổng trong thời gian $O(\sqrt{n})$.

Ý tưởng là chia mảng thành các *khối* có kích thước $\sqrt{n}$ sao cho mỗi khối lưu tổng các phần tử bên trong khối đó. Ví dụ, một mảng 16 phần tử sẽ được chia thành các khối 4 phần tử như sau:

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Trong cấu trúc này, việc thay đổi phần tử của mảng rất dễ, vì sau mỗi lần thay đổi ta chỉ cần cập nhật tổng của đúng một khối, điều này thực hiện được trong thời gian $O(1)$. Ví dụ, hình sau cho thấy giá trị của một phần tử và tổng của khối tương ứng thay đổi như thế nào:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Sau đó, để tính tổng các phần tử trong một đoạn, ta chia đoạn đó thành ba phần sao cho tổng cần tìm gồm giá trị của một vài phần tử riêng lẻ và tổng của các khối nằm giữa chúng:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Vì số phần tử riêng lẻ là $O(\sqrt{n})$ và số khối cũng là $O(\sqrt{n})$, truy vấn tổng mất thời gian $O(\sqrt{n})$. Mục đích của kích thước khối $\sqrt{n}$ là *cân bằng* hai yếu tố: mảng được chia thành $\sqrt{n}$ khối, và mỗi khối chứa $\sqrt{n}$ phần tử.

Trong thực tế, không nhất thiết phải dùng chính xác giá trị $\sqrt{n}$ làm tham số; thay vào đó ta có thể dùng các tham số k và n/k , trong đó k khác $\sqrt{n}$. Tham số tối ưu phụ thuộc vào bài toán và dữ liệu vào. Ví dụ, nếu một thuật toán thường xuyên đi qua các khối nhưng hiếm khi xét từng phần tử bên trong khối, ta có thể nên chia mảng thành $k < \sqrt{n}$ khối, mỗi khối chứa $n/k > \sqrt{n}$ phần tử.

27.1 Kết hợp thuật toán

Trong phần này ta thảo luận hai thuật toán căn bậc hai dựa trên việc kết hợp hai thuật toán thành một thuật toán. Trong cả hai trường hợp, ta có thể dùng riêng từng thuật toán và giải bài toán trong thời gian $O(n^2)$. Tuy nhiên, bằng cách kết hợp các thuật toán, thời gian chạy chỉ còn $O(n\sqrt{n})$.

Xử lý theo trường hợp

Giả sử ta được cho một lưới hai chiều gồm n ô. Mỗi ô được gán một chữ cái, và nhiệm vụ là tìm hai ô có cùng chữ cái với khoảng cách nhỏ nhất, trong đó khoảng cách giữa hai ô (x_1, y_1) và (x_2, y_2) là $|x_1 - x_2| + |y_1 - y_2|$. Ví dụ, xét lưới sau:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

Trong trường hợp này, khoảng cách nhỏ nhất là 2 giữa hai chữ cái 'E'.

Ta có thể giải bài toán bằng cách xét riêng từng chữ cái. Với cách tiếp cận này, bài toán mới là tính khoảng cách nhỏ nhất giữa hai ô có một chữ cái *cố định* c . Ta tập trung vào hai thuật toán cho việc này:

Thuật toán 1: Duyệt qua mọi cặp ô có chữ cái c , và tính khoảng cách nhỏ nhất giữa các ô như vậy. Việc này mất thời gian $O(k^2)$, trong đó k là số ô có chữ cái c .

Thuật toán 2: Thực hiện tìm kiếm theo chiều rộng bắt đầu đồng thời từ mọi ô có chữ cái c . Khoảng cách nhỏ nhất giữa hai ô có chữ cái c sẽ được tính trong thời gian $O(n)$.

Một cách giải bài toán là chọn một trong hai thuật toán và dùng nó cho mọi chữ cái. Nếu dùng Thuật toán 1, thời gian chạy là $O(n^2)$, vì tất cả ô có thể chứa cùng một chữ cái, và khi đó $k = n$. Nếu dùng Thuật toán 2, thời gian chạy cũng là $O(n^2)$, vì tất cả ô có thể có các chữ cái khác nhau, và khi đó cần n lần tìm kiếm.

Tuy nhiên, ta có thể *kết hợp* hai thuật toán và dùng các thuật toán khác nhau cho các chữ cái khác nhau tùy theo số lần mỗi chữ cái xuất hiện trong lưới. Giả sử một chữ cái c xuất hiện k lần. Nếu $k \leq \sqrt{n}$, ta dùng Thuật toán 1, còn nếu $k > \sqrt{n}$, ta dùng Thuật toán 2. Khi làm như vậy, tổng thời gian chạy của thuật toán chỉ là $O(n\sqrt{n})$.

Trước hết, giả sử ta dùng Thuật toán 1 cho một chữ cái c . Vì c xuất hiện nhiều nhất $\sqrt{n}$ lần trong lưới, ta so sánh mỗi ô có chữ cái c với các ô khác $O(\sqrt{n})$ lần. Do đó, thời gian dùng để xử lý tất cả các ô như vậy là $O(n\sqrt{n})$. Tiếp theo, giả sử ta dùng Thuật toán 2 cho một chữ cái c . Có nhiều nhất $\sqrt{n}$ chữ cái như vậy, nên xử lý các chữ cái đó cũng mất thời gian $O(n\sqrt{n})$.

Xử lý theo lô

Bài toán tiếp theo cũng liên quan đến một lưới hai chiều gồm n ô. Ban đầu, mọi ô trừ một ô đều màu trắng. Ta thực hiện $n - 1$ thao tác, mỗi thao tác trước hết tính khoảng cách nhỏ nhất từ một ô trắng cho trước đến một ô đen, rồi tô ô trắng đó thành màu đen.

Ví dụ, xét thao tác sau:

		*	

Trước hết, ta tính khoảng cách nhỏ nhất từ ô trắng được đánh dấu * đến một ô đen. Khoảng cách nhỏ nhất là 2, vì ta có thể đi hai bước sang trái để đến một ô đen. Sau đó, ta tô ô trắng đó thành màu đen:

Xét hai thuật toán sau:

Thuật toán 1: Dùng tìm kiếm theo chiều rộng để tính cho mỗi ô trắng khoảng cách đến ô đen gần nhất. Việc này mất thời gian $O(n)$, và sau khi tìm kiếm, ta có thể tìm khoảng cách nhỏ nhất từ bất kỳ ô trắng nào đến một ô đen trong thời gian $O(1)$.

Thuật toán 2: Duy trì một danh sách các ô đã được tô đen, duyệt qua danh sách này ở mỗi thao tác rồi thêm một ô mới vào danh sách. Một thao tác mất thời gian $O(k)$, trong đó k là độ dài danh sách.

Ta kết hợp các thuật toán trên bằng cách chia các thao tác thành $O(\sqrt{n})$ lô, mỗi lô gồm $O(\sqrt{n})$ thao tác. Ở đầu mỗi lô, ta thực hiện Thuật toán 1. Sau

đó, ta dùng Thuật toán 2 để xử lý các thao tác trong lô. Ta xóa danh sách của Thuật toán 2 giữa các lô. Ở mỗi thao tác, khoảng cách nhỏ nhất đến một ô đen là khoảng cách do Thuật toán 1 tính được hoặc khoảng cách do Thuật toán 2 tính được.

Thuật toán thu được chạy trong thời gian $O(n\sqrt{n})$. Thứ nhất, Thuật toán 1 được thực hiện $O(\sqrt{n})$ lần, và mỗi lần tìm kiếm chạy trong thời gian $O(n)$. Thứ hai, khi dùng Thuật toán 2 trong một lô, danh sách chứa $O(\sqrt{n})$ ô (vì ta xóa danh sách giữa các lô) và mỗi thao tác mất thời gian $O(\sqrt{n})$.

27.2 Phân hoạch số nguyên

Một số thuật toán căn bậc hai dựa trên quan sát sau: nếu một số nguyên dương n được biểu diễn thành tổng của các số nguyên dương, tổng đó luôn chứa nhiều nhất $O(\sqrt{n})$ số *phân biệt*. Lý do là để xây dựng một tổng chứa số lượng số phân biệt lớn nhất, ta nên chọn các số *nhỏ*. Nếu chọn các số $1, 2, \dots, k$, tổng thu được là

$$\frac{k(k+1)}{2}.$$

Do đó, số lượng số phân biệt tối đa là $k = O(\sqrt{n})$. Tiếp theo ta sẽ thảo luận hai bài toán có thể được giải hiệu quả bằng quan sát này.

Balo (Knapsack)

Giả sử ta được cho một danh sách các trọng lượng nguyên có tổng là n . Nhiệm vụ là tìm tất cả các tổng có thể tạo được bằng một tập con của các trọng lượng. Ví dụ, nếu các trọng lượng là $\{1, 3, 3\}$, các tổng có thể có là:

- 0 (tập rỗng)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Dùng cách tiếp cận balo chuẩn (xem Chương 7.4), bài toán có thể được giải như sau: ta định nghĩa một hàm $\text{possible}(x, k)$ có giá trị 1 nếu tổng x có thể được tạo bằng k trọng lượng đầu tiên, và bằng 0 trong trường hợp ngược lại. Vì tổng các trọng lượng là n , có nhiều nhất n trọng lượng và tất cả giá trị của hàm có thể được tính trong thời gian $O(n^2)$ bằng quy hoạch động.

Tuy nhiên, ta có thể làm thuật toán hiệu quả hơn bằng cách dùng sự thật rằng có nhiều nhất $O(\sqrt{n})$ trọng lượng *phân biệt*. Do đó, ta có thể xử lý các trọng lượng theo từng nhóm gồm các trọng lượng giống nhau. Ta có thể xử lý mỗi nhóm trong thời gian $O(n)$, từ đó có thuật toán thời gian $O(n\sqrt{n})$.

Ý tưởng là dùng một mảng ghi lại các tổng trọng lượng có thể tạo được bằng các nhóm đã xử lý cho đến hiện tại. Mảng chứa n phần tử: phần tử k

bằng 1 nếu tổng k có thể tạo được và bằng 0 nếu không. Để xử lý một nhóm trọng lượng, ta quét mảng từ trái sang phải và ghi lại các tổng trọng lượng mới có thể tạo được bằng nhóm này và các nhóm trước đó.

Xây dựng xâu

Cho một xâu s độ dài n và một tập các xâu D có tổng độ dài là m , xét bài toán đếm số cách tạo s bằng cách nối các xâu trong D . Ví dụ, nếu $s = \text{ABAB}$ và $D = \{A, B, AB\}$, có 4 cách:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Ta có thể giải bài toán bằng quy hoạch động: Gọi $\text{count}(k)$ là số cách xây dựng tiền tố $s[0 \dots k]$ bằng các xâu trong D . Khi đó $\text{count}(n - 1)$ cho đáp án của bài toán, và ta có thể giải bài toán trong thời gian $O(n^2)$ bằng một cấu trúc trie.

Tuy nhiên, ta có thể giải bài toán hiệu quả hơn bằng cách dùng băm xâu và sự thật rằng có nhiều nhất $O(\sqrt{m})$ độ dài xâu phân biệt trong D . Trước hết, ta xây dựng một tập H chứa tất cả giá trị băm của các xâu trong D . Sau đó, khi tính một giá trị $\text{count}(k)$, ta duyệt qua tất cả giá trị p sao cho có một xâu độ dài p trong D , tính giá trị băm của $s[k - p + 1 \dots k]$ và kiểm tra nó có thuộc H hay không. Vì có nhiều nhất $O(\sqrt{m})$ độ dài xâu phân biệt, ta thu được một thuật toán có thời gian chạy $O(n\sqrt{m})$.

27.3 Thuật toán Mo

Thuật toán Mo (Mo's algorithm)¹ có thể được dùng trong nhiều bài toán cần xử lý truy vấn đoạn trên một mảng *tĩnh*, tức là các giá trị của mảng không thay đổi giữa các truy vấn. Trong mỗi truy vấn, ta được cho một đoạn $[a, b]$, và cần tính một giá trị dựa trên các phần tử của mảng giữa vị trí a và b . Vì mảng là tĩnh, các truy vấn có thể được xử lý theo bất kỳ thứ tự nào, và thuật toán Mo xử lý các truy vấn theo một thứ tự đặc biệt bảo đảm thuật toán chạy hiệu quả.

Thuật toán Mo duy trì một *đoạn đang hoạt động* của mảng, và tại mọi thời điểm ta biết đáp án cho truy vấn tương ứng với đoạn đang hoạt động. Thuật toán xử lý từng truy vấn một, và luôn di chuyển hai đầu mút của đoạn đang hoạt động bằng cách thêm và xóa phần tử. Độ phức tạp thời gian của thuật toán là $O(n\sqrt{n}f(n))$, trong đó mảng có n phần tử, có n truy vấn và mỗi lần thêm hoặc xóa một phần tử mất thời gian $O(f(n))$.

¹Theo [12], thuật toán này được đặt theo tên Mo Tao, một lập trình viên thi đấu người Trung Quốc, nhưng kỹ thuật này đã xuất hiện sớm hơn trong tài liệu [44].

Mấu chốt trong thuật toán Mo là thứ tự xử lý các truy vấn: Mảng được chia thành các khối gồm $k = O(\sqrt{n})$ phần tử, và một truy vấn $[a_1, b_1]$ được xử lý trước một truy vấn $[a_2, b_2]$ nếu một trong hai điều kiện sau đúng:

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ or
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ và $b_1 < b_2$.

Do đó, tất cả truy vấn có đầu mút trái nằm trong cùng một khối được xử lý liên tiếp, sắp xếp theo đầu mút phải của chúng. Với thứ tự này, thuật toán chỉ thực hiện $O(n\sqrt{n})$ thao tác, vì đầu mút trái di chuyển $O(n)$ lần, mỗi lần $O(\sqrt{n})$ bước, còn đầu mút phải di chuyển $O(\sqrt{n})$ lần, mỗi lần $O(n)$ bước. Do đó, cả hai đầu mút di chuyển tổng cộng $O(n\sqrt{n})$ bước trong thuật toán.

Ví dụ

Ví dụ, xét một bài toán trong đó ta được cho một tập truy vấn, mỗi truy vấn tương ứng với một đoạn trong mảng, và nhiệm vụ là tính cho mỗi truy vấn số phần tử *phân biệt* trong đoạn.

Trong thuật toán Mo, các truy vấn luôn được sắp xếp theo cùng một cách, nhưng cách duy trì đáp án phụ thuộc vào từng bài toán. Trong bài toán này, ta có thể duy trì một mảng `count`, trong đó `count[x]` cho biết số lần phần tử x xuất hiện trong đoạn đang hoạt động.

Khi chuyển từ một truy vấn sang truy vấn khác, đoạn đang hoạt động thay đổi. Ví dụ, nếu đoạn hiện tại là

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

và đoạn tiếp theo là

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

sẽ có ba bước: đầu mút trái di chuyển một bước sang phải, và đầu mút phải di chuyển hai bước sang phải.

Sau mỗi bước, mảng `count` cần được cập nhật. Sau khi thêm một phần tử x , ta tăng giá trị của `count[x]` thêm 1, và nếu sau đó `count[x] = 1`, ta cũng tăng đáp án của truy vấn thêm 1. Tương tự, sau khi xóa một phần tử x , ta giảm giá trị của `count[x]` đi 1, và nếu sau đó `count[x] = 0`, ta cũng giảm đáp án của truy vấn đi 1.

Trong bài toán này, thời gian cần để thực hiện mỗi bước là $O(1)$, nên tổng độ phức tạp thời gian của thuật toán là $O(n\sqrt{n})$.

Chương 28

Cây phân đoạn nâng cao (Segment Trees Revisited)

Cây phân đoạn là một cấu trúc dữ liệu linh hoạt có thể được dùng để giải rất nhiều bài toán thuật toán. Tuy nhiên, vẫn còn nhiều chủ đề liên quan đến cây phân đoạn mà ta chưa đề cập. Bây giờ là lúc thảo luận một số biến thể nâng cao hơn của cây phân đoạn.

Cho đến nay, ta đã cài đặt các thao tác của cây phân đoạn bằng cách đi *từ dưới lên* trong cây. Ví dụ, ta đã tính tổng đoạn như sau (Chương 9.3):

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Tuy nhiên, trong các cây phân đoạn nâng cao hơn, thường cần cài đặt các thao tác theo một cách khác, *từ trên xuống*. Với cách tiếp cận này, hàm trở thành như sau:

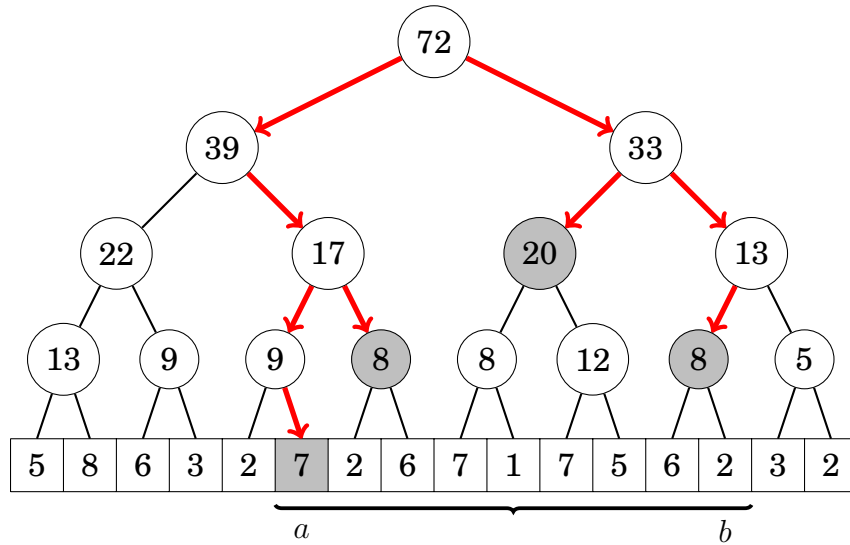
```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Bây giờ ta có thể tính bất kỳ giá trị nào của $\text{sum}_q(a, b)$ (tổng các giá trị của mảng trong đoạn $[a, b]$) như sau:

```
int s = sum(a, b, 1, 0, n-1);
```

Tham số k cho biết vị trí hiện tại trong tree. Ban đầu k bằng 1, vì ta bắt đầu tại gốc của cây. Đoạn $[x, y]$ tương ứng với k và ban đầu là $[0, n - 1]$. Khi tính tổng, nếu $[x, y]$ nằm ngoài $[a, b]$, tổng là 0, và nếu $[x, y]$ nằm hoàn toàn trong $[a, b]$, tổng có thể được lấy từ tree. Nếu $[x, y]$ chỉ nằm một phần trong $[a, b]$, việc tìm kiếm tiếp tục đệ quy xuống nửa trái và nửa phải của $[x, y]$. Nửa trái là $[x, d]$ và nửa phải là $[d + 1, y]$, trong đó $d = \lfloor \frac{x+y}{2} \rfloor$.

Hình sau cho thấy quá trình tìm kiếm khi tính giá trị $\text{sum}_q(a, b)$. Các nút màu xám biểu thị những nút nơi đệ quy dừng lại và tổng có thể được lấy từ tree.



Trong cài đặt này, các thao tác cũng mất thời gian $O(\log n)$, vì tổng số nút được thăm là $O(\log n)$.

28.1 Lan truyền lười (Lazy propagation)

Bằng **lan truyền lười**, ta có thể xây dựng một cây phân đoạn hỗ trợ cả cập nhật đoạn và truy vấn đoạn trong thời gian $O(\log n)$. Ý tưởng là thực hiện cập nhật và truy vấn từ trên xuống, đồng thời thực hiện cập nhật một cách *lười*, sao cho chúng chỉ được lan truyền xuống cây khi thật sự cần thiết.

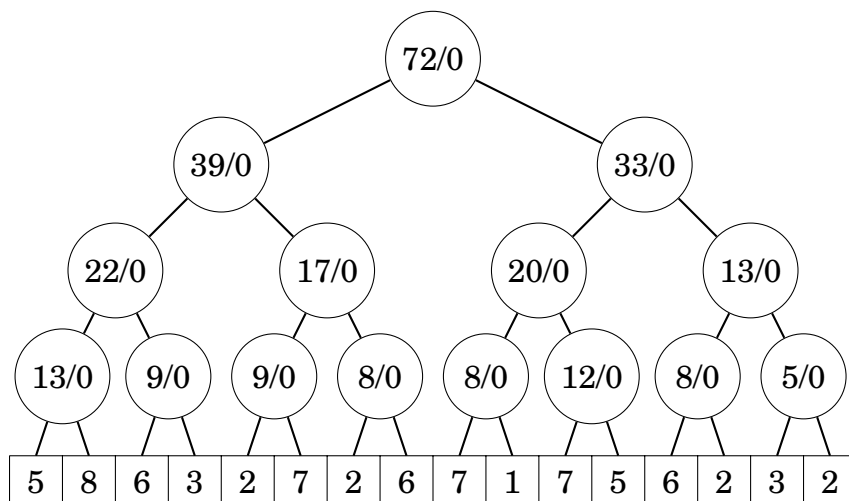
Trong một cây phân đoạn lười, các nút chứa hai loại thông tin. Giống như trong cây phân đoạn thông thường, mỗi nút chứa tổng hoặc một giá trị khác liên quan đến mảng con tương ứng. Ngoài ra, nút có thể chứa thông tin liên quan đến các cập nhật lười chưa được lan truyền xuống các con của nó.

Có hai loại cập nhật đoạn: mỗi giá trị trong đoạn hoặc được *tăng* thêm một giá trị nào đó hoặc được *gán* bằng một giá trị nào đó. Cả hai thao tác đều có thể được cài đặt bằng những ý tưởng tương tự, và thậm chí có thể xây dựng một cây hỗ trợ cả hai thao tác cùng lúc.

Cây phân đoạn lười

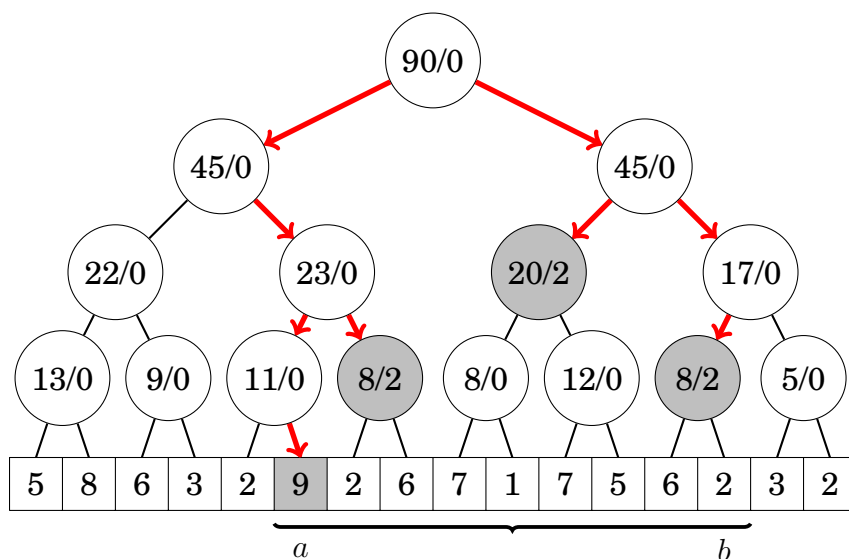
Xét một ví dụ trong đó mục tiêu của ta là xây dựng một cây phân đoạn hỗ trợ hai thao tác: tăng mỗi giá trị trong $[a, b]$ thêm một hằng số và tính tổng các giá trị trong $[a, b]$.

Ta sẽ xây dựng một cây trong đó mỗi nút có hai giá trị s/z : s biểu thị tổng các giá trị trong đoạn, và z biểu thị giá trị của một cặp nhật lười, tức là mọi giá trị trong đoạn cần được tăng thêm z . Trong cây sau, $z = 0$ ở mọi nút, nên không có cặp nhật lười nào đang chờ.



Khi các phần tử trong $[a, b]$ được tăng thêm u , ta đi từ gốc về phía lá và sửa các nút của cây như sau: Nếu đoạn $[x, y]$ của một nút nằm hoàn toàn trong $[a, b]$, ta tăng giá trị z của nút thêm u rồi dừng. Nếu $[x, y]$ chỉ thuộc một phần vào $[a, b]$, ta tăng giá trị s của nút thêm hu , trong đó h là kích thước giao của $[a, b]$ và $[x, y]$, rồi tiếp tục đi đệ quy trong cây.

Ví dụ, hình sau cho thấy cây sau khi tăng các phần tử trong $[a, b]$ thêm 2:

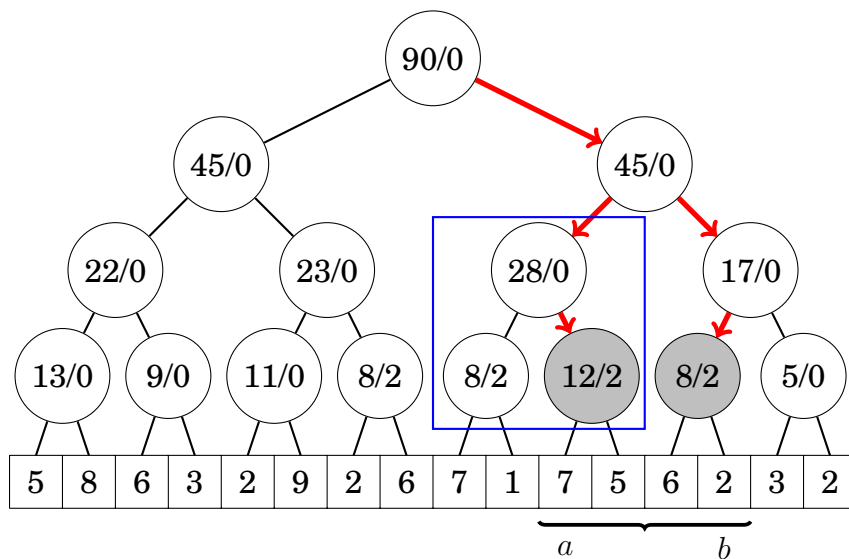


Ta cũng tính tổng các phần tử trong một đoạn $[a, b]$ bằng cách đi trong cây từ trên xuống. Nếu đoạn $[x, y]$ của một nút nằm hoàn toàn trong $[a, b]$, ta

cộng giá trị s của nút vào tổng. Ngược lại, ta tiếp tục tìm kiếm đệ quy xuống dưới trong cây.

Cả trong cập nhật lẫn truy vấn, giá trị của một cặp nhật lười luôn được lan truyền đến các con của nút trước khi xử lý nút đó. Ý tưởng là các cặp nhật chỉ được lan truyền xuống dưới khi cần thiết, điều này bảo đảm các thao tác luôn hiệu quả.

Hình sau cho thấy cây thay đổi như thế nào khi ta tính giá trị $\text{sum}_a(a, b)$. Hình chữ nhật chỉ các nút có giá trị thay đổi, vì một cặp nhật lười được lan truyền xuống dưới.



Lưu ý rằng đôi khi cần kết hợp các cặp nhật lười. Điều này xảy ra khi một nút đã có cặp nhật lười lại được gán thêm một cặp nhật lười khác. Khi tính tổng, việc kết hợp các cặp nhật lười rất dễ, vì sự kết hợp của hai cặp nhật z_1 và z_2 tương ứng với cặp nhật $z_1 + z_2$.

Cập nhật đa thức

Cập nhật lười có thể được tổng quát hóa để có thể cập nhật các đoạn bằng đa thức có dạng

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

Trong trường hợp này, cập nhật cho giá trị tại vị trí i trong $[a, b]$ là $p(i - a)$. Ví dụ, cộng đa thức $p(u) = u + 1$ vào $[a, b]$ nghĩa là giá trị tại vị trí a tăng thêm 1, giá trị tại vị trí $a + 1$ tăng thêm 2, và cứ tiếp tục như vậy.

Để hỗ trợ cập nhật đa thức, mỗi nút được gán $k + 2$ giá trị, trong đó k bằng bậc của đa thức. Giá trị s là tổng các phần tử trong đoạn, và các giá trị $z_0, z_1, \dots, z_k$ là các hệ số của một đa thức tương ứng với cặp nhật lười.

Bây giờ, tổng các giá trị trong một đoạn $[x, y]$ bằng

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

Giá trị của một tổng như vậy có thể được tính hiệu quả bằng các công thức tính tổng. Ví dụ, hạng tử z_0 tương ứng với tổng $(y - x + 1)z_0$, và hạng tử $z_1 u$ tương ứng với tổng

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

Khi lan truyền một cập nhật trong cây, các chỉ số của $p(u)$ thay đổi, vì trong mỗi đoạn $[x, y]$, các giá trị được tính cho $u = 0, 1, \dots, y - x$. Tuy nhiên, đây không phải vấn đề, vì $p'(u) = p(u + h)$ là một đa thức cùng bậc với $p(u)$. Ví dụ, nếu $p(u) = t_2 u^2 + t_1 u - t_0$, thì

$$p'(u) = t_2(u + h)^2 + t_1(u + h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

28.2 Cây động

Cây phân đoạn thông thường là tĩnh, nghĩa là mỗi nút có một vị trí cố định trong mảng và cây cần một lượng bộ nhớ cố định. Trong một **cây phân đoạn động**, bộ nhớ chỉ được cấp phát cho các nút thật sự được truy cập trong thuật toán, nhờ đó có thể tiết kiệm rất nhiều bộ nhớ.

Các nút của một cây động có thể được biểu diễn bằng struct:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Ở đây value là giá trị của nút, $[x, y]$ là đoạn tương ứng, và left cùng right trỏ đến cây con trái và cây con phải.

Sau đó, các nút có thể được tạo như sau:

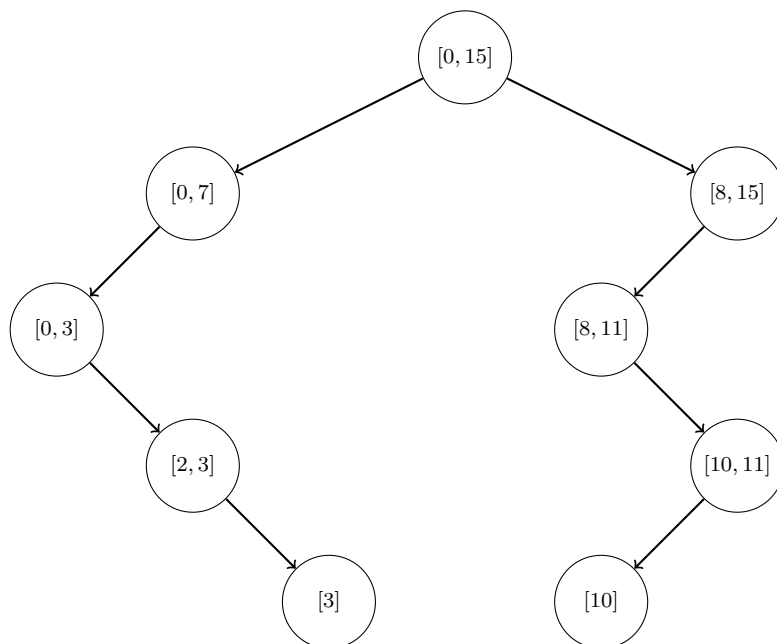
```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

Cây phân đoạn thưa

Cây phân đoạn động hữu ích khi mảng nền là *thưa*, tức là đoạn chỉ số được phép $[0, n - 1]$ rất lớn, nhưng phần lớn giá trị của mảng bằng không. Trong khi cây phân đoạn thông thường dùng $O(n)$ bộ nhớ, cây phân đoạn động chỉ dùng $O(k \log n)$ bộ nhớ, trong đó k là số thao tác đã thực hiện.

Một **cây phân đoạn thưa** ban đầu chỉ có một nút $[0, n - 1]$ có giá trị bằng không, nghĩa là mọi giá trị của mảng đều bằng không. Sau các cập

nhật, các nút mới được thêm động vào cây. Ví dụ, nếu $n = 16$ và các phần tử ở vị trí 3 và 10 đã được sửa đổi, cây chứa các nút sau:



Bất kỳ đường đi nào từ nút gốc đến một lá đều chứa $O(\log n)$ nút, nên mỗi thao tác thêm nhiều nhất $O(\log n)$ nút mới vào cây. Do đó, sau k thao tác, cây chứa nhiều nhất $O(k \log n)$ nút.

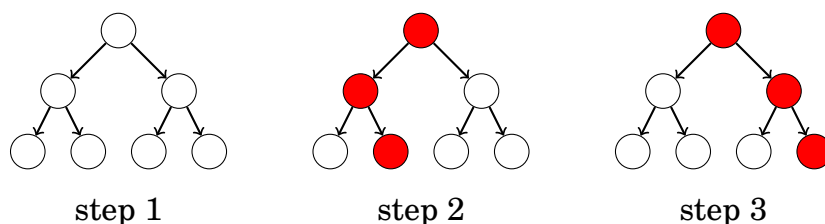
Lưu ý rằng nếu ta biết tất cả phần tử sẽ được cập nhật ngay từ đầu thuật toán, thì không cần cây phân đoạn động, vì ta có thể dùng cây phân đoạn thông thường với nén chỉ số (Chương 9.4). Tuy nhiên, điều này không thể làm được khi các chỉ số được sinh ra trong quá trình chạy thuật toán.

Cây phân đoạn bền vững

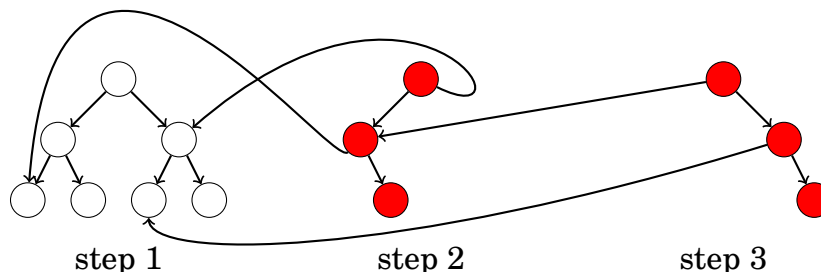
Bằng một cài đặt động, ta cũng có thể tạo một **cây phân đoạn bền vững (persistent segment tree)** lưu *lịch sử thay đổi* của cây. Trong cài đặt như vậy, ta có thể truy cập hiệu quả mọi phiên bản của cây đã từng tồn tại trong thuật toán.

Khi lịch sử thay đổi có sẵn, ta có thể thực hiện truy vấn trên bất kỳ cây trước đó nào giống như trên cây phân đoạn thông thường, vì toàn bộ cấu trúc của mỗi cây đều được lưu. Ta cũng có thể tạo cây mới dựa trên các cây trước đó và sửa đổi chúng độc lập.

Xét dãy cập nhật sau, trong đó các nút đỏ thay đổi và các nút khác giữ nguyên:



Sau mỗi cập nhật, phần lớn các nút của cây giữ nguyên, nên một cách hiệu quả để lưu lịch sử thay đổi là biểu diễn mỗi cây trong lịch sử như sự kết hợp của các nút mới và các cây con của những cây trước đó. Trong ví dụ này, lịch sử thay đổi có thể được lưu như sau:



Cấu trúc của mỗi cây trước đó có thể được khôi phục bằng cách lần theo các con trỏ bắt đầu từ nút gốc tương ứng. Vì mỗi thao tác chỉ thêm $O(\log n)$ nút mới vào cây, ta có thể lưu toàn bộ lịch sử thay đổi của cây.

28.3 Cấu trúc dữ liệu

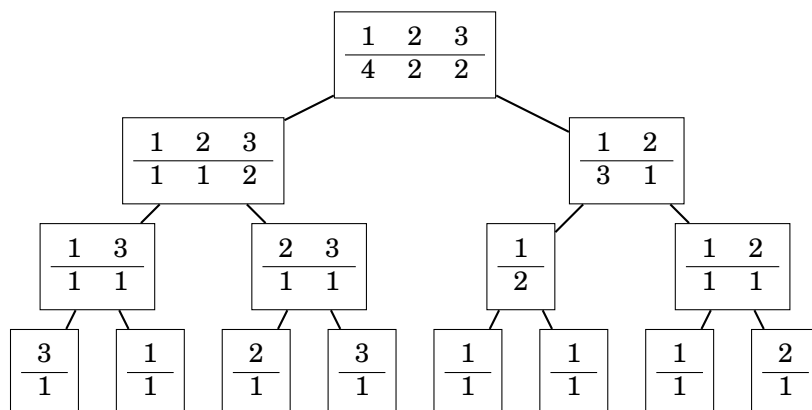
Thay vì các giá trị đơn lẻ, các nút trong cây phân đoạn cũng có thể chứa *cấu trúc dữ liệu* duy trì thông tin về các đoạn tương ứng. Trong một cây như vậy, các thao tác mất thời gian $O(f(n) \log n)$, trong đó $f(n)$ là thời gian cần để xử lý một nút đơn trong một thao tác.

Ví dụ, xét một cây phân đoạn hỗ trợ các truy vấn dạng “phần tử x xuất hiện bao nhiêu lần trong đoạn $[a, b]$?” Ví dụ, phần tử 1 xuất hiện ba lần trong đoạn sau:

3	1	2	3	1	1	1	2
---	---	---	---	---	---	---	---

Để hỗ trợ các truy vấn như vậy, ta xây dựng một cây phân đoạn trong đó mỗi nút được gán một cấu trúc dữ liệu có thể trả lời một phần tử bất kỳ x xuất hiện bao nhiêu lần trong đoạn tương ứng. Dùng cây này, đáp án cho một truy vấn có thể được tính bằng cách kết hợp kết quả từ các nút thuộc đoạn đó.

Ví dụ, cây phân đoạn sau tương ứng với mảng trên:



Ta có thể xây dựng cây sao cho mỗi nút chứa một cấu trúc map. Trong trường hợp này, thời gian cần để xử lý mỗi nút là $O(\log n)$, nên tổng độ phức tạp thời gian của một truy vấn là $O(\log^2 n)$. Cây dùng $O(n \log n)$ bộ nhớ, vì có $O(\log n)$ mức và mỗi mức chứa $O(n)$ phần tử.

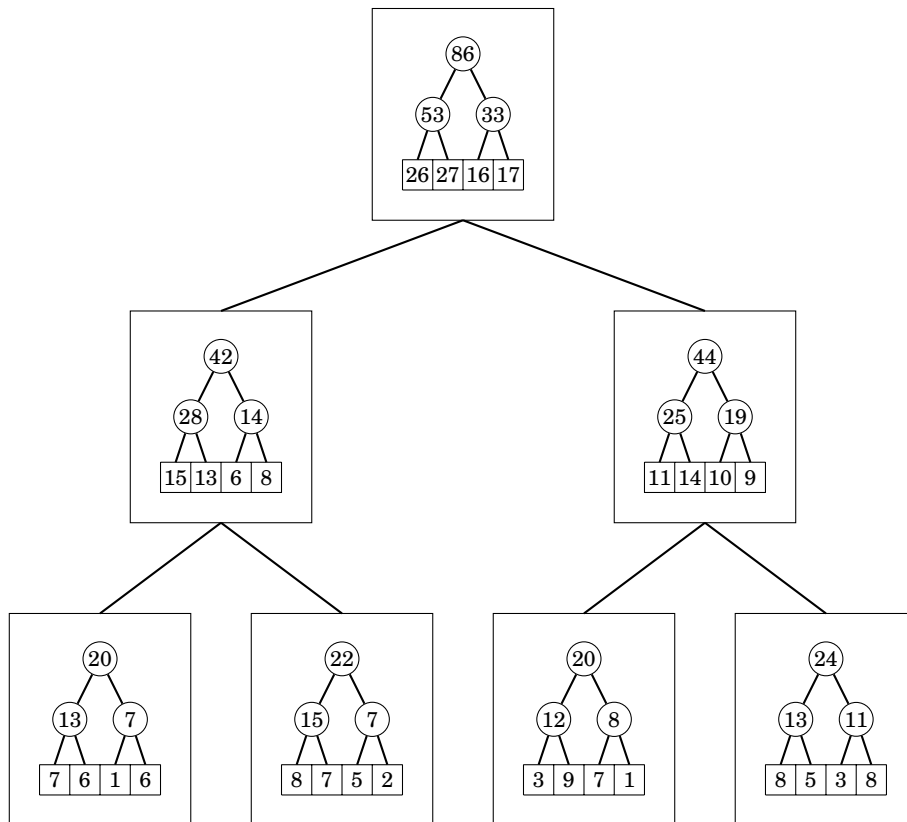
28.4 Hai chiều

Một **cây phân đoạn hai chiều** hỗ trợ các truy vấn liên quan đến các mảng con hình chữ nhật của một mảng hai chiều. Một cây như vậy có thể được cài đặt dưới dạng các cây phân đoạn lồng nhau: một cây lớn tương ứng với các hàng của mảng, và mỗi nút chứa một cây nhỏ tương ứng với một cột.

Ví dụ, trong mảng

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

tổng của bất kỳ mảng con nào có thể được tính từ cây phân đoạn sau:



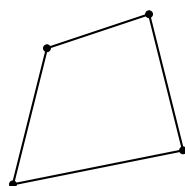
Các thao tác của cây phân đoạn hai chiều mất thời gian $O(\log^2 n)$, vì cây lớn và mỗi cây nhỏ đều gồm $O(\log n)$ mức. Cây cần $O(n^2)$ bộ nhớ, vì mỗi cây nhỏ chứa $O(n)$ giá trị.

Chương 29

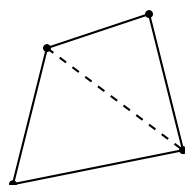
Hình học

Trong các bài toán hình học, thường khó tìm được một cách tiếp cận sao cho lời giải có thể được cài đặt thuận tiện và số trường hợp đặc biệt là ít.

Ví dụ, xét một bài toán trong đó ta được cho các đỉnh của một tứ giác (một đa giác có bốn đỉnh), và nhiệm vụ là tính diện tích của nó. Ví dụ, một dữ liệu vào có thể có dạng như sau:



Một cách tiếp cận bài toán là chia tứ giác thành hai tam giác bằng một đoạn thẳng nối hai đỉnh đối diện:

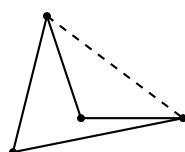


Sau đó, chỉ cần cộng diện tích của các tam giác. Diện tích của một tam giác có thể được tính, chẳng hạn, bằng **công thức Heron**

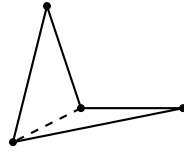
$$\sqrt{s(s-a)(s-b)(s-c)},$$

trong đó a , b và c là độ dài các cạnh của tam giác và $s = (a + b + c)/2$.

Đây là một cách có thể dùng để giải bài toán, nhưng có một điểm dễ mắc lỗi: chia tứ giác thành các tam giác như thế nào? Hóa ra đôi khi ta không thể chỉ chọn hai đỉnh đối diện tùy ý. Ví dụ, trong tình huống sau, đường chia nằm *bên ngoài* tứ giác:



Tuy nhiên, một cách vẽ đường khác lại đúng:



Với con người, rõ ràng đường nào là lựa chọn đúng, nhưng tình huống này lại khó đối với máy tính.

Tuy nhiên, hóa ra ta có thể giải bài toán bằng một phương pháp khác thuận tiện hơn cho lập trình viên. Cụ thể, có một công thức tổng quát

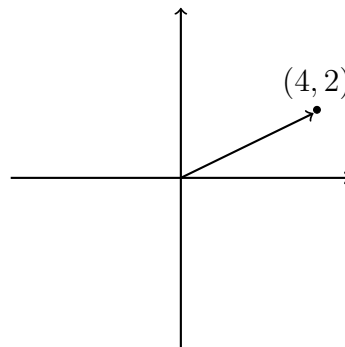
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

tính diện tích của một tứ giác có các đỉnh là (x_1, y_1) , (x_2, y_2) , (x_3, y_3) và (x_4, y_4) . Công thức này dễ cài đặt, không có trường hợp đặc biệt, và ta thậm chí có thể tổng quát hóa công thức cho *mọi* đa giác.

29.1 Số phức

Một **số phức** là một số có dạng $x + yi$, trong đó $i = \sqrt{-1}$ là **đơn vị ảo**. Diễn giải hình học của một số phức là nó biểu diễn một điểm hai chiều (x, y) hoặc một vectơ từ gốc tọa độ đến điểm (x, y) .

Ví dụ, $4 + 2i$ tương ứng với điểm và vectơ sau:



Lớp số phức `complex` của C++ rất hữu ích khi giải các bài toán hình học. Dùng lớp này, ta có thể biểu diễn điểm và vectơ dưới dạng số phức, và lớp này chứa các công cụ hữu ích trong hình học.

Trong đoạn mã sau, `C` là kiểu của một tọa độ và `P` là kiểu của một điểm hoặc một vectơ. Ngoài ra, đoạn mã định nghĩa các macro `X` và `Y` có thể dùng để tham chiếu đến tọa độ `x` và `y`.

```
typedef long long C;  
typedef complex<C> P;  
#define X real()  
#define Y imag()
```

Ví dụ, đoạn mã sau định nghĩa một điểm $p = (4, 2)$ và in tọa độ x và y của nó:

```
P p = {4,2};  
cout << p.X << " " << p.Y << "\n"; // 4 2
```

Đoạn mã sau định nghĩa các vectơ $v = (3, 1)$ và $u = (2, 2)$, rồi tính tổng $s = v + u$.

```
P v = {3,1};  
P u = {2,2};  
P s = v+u;  
cout << s.X << " " << s.Y << "\n"; // 5 3
```

Trong thực tế, kiểu tọa độ phù hợp thường là `long long` (số nguyên) hoặc `long double` (số thực). Nên dùng số nguyên bất cứ khi nào có thể, vì các phép tính với số nguyên là chính xác. Nếu cần dùng số thực, cần tính đến sai số chính xác khi so sánh các số. Một cách an toàn để kiểm tra hai số thực a và b có bằng nhau không là so sánh chúng bằng $|a - b| < \epsilon$, trong đó ϵ là một số nhỏ (ví dụ, $\epsilon = 10^{-9}$).

Hàm

Trong các ví dụ sau, kiểu tọa độ là `long double`.

Hàm `abs(v)` tính độ dài $|v|$ của một vectơ $v = (x, y)$ bằng công thức $\sqrt{x^2 + y^2}$. Hàm này cũng có thể được dùng để tính khoảng cách giữa các điểm (x_1, y_1) và (x_2, y_2) , vì khoảng cách đó bằng độ dài của vectơ $(x_2 - x_1, y_2 - y_1)$.

Đoạn mã sau tính khoảng cách giữa các điểm $(4, 2)$ và $(3, -1)$:

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.16228
```

Hàm `arg(v)` tính góc của một vectơ $v = (x, y)$ so với trục x . Hàm trả về góc theo radian, trong đó r radian bằng $180r/\pi$ độ. Góc của một vectơ hướng sang phải là 0, và góc giảm theo chiều kim đồng hồ, tăng ngược chiều kim đồng hồ.

Hàm `polar(s, a)` tạo một vectơ có độ dài s và hướng theo góc a . Một vectơ có thể được quay một góc a bằng cách nhân nó với một vectơ có độ dài 1 và góc a .

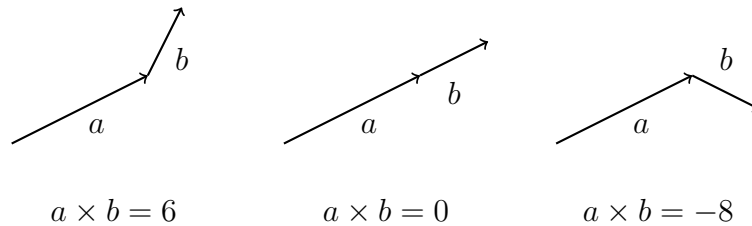
Đoạn mã sau tính góc của vectơ $(4, 2)$, quay nó $1/2$ radian ngược chiều kim đồng hồ, rồi tính lại góc:

```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0,0.5);  
cout << arg(v) << "\n"; // 0.963648
```

29.2 Điểm và đường thẳng

Tích có hướng (cross product) $a \times b$ của các vectơ $a = (x_1, y_1)$ và $b = (x_2, y_2)$ được tính bằng công thức $x_1y_2 - x_2y_1$. Tích có hướng cho biết liệu b rẽ trái (giá trị dương), không rẽ (bằng không) hay rẽ phải (giá trị âm) khi nó được đặt ngay sau a .

Hình sau minh họa các trường hợp trên:



Ví dụ, trong trường hợp đầu tiên $a = (4, 2)$ và $b = (1, 2)$. Đoạn mã sau tính tích có hướng bằng lớp complex:

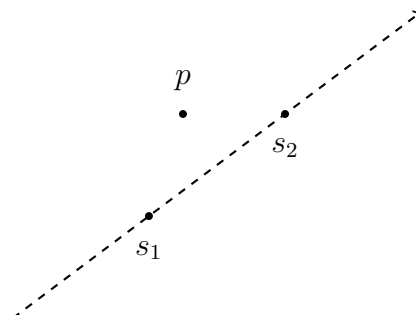
```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

Đoạn mã trên hoạt động vì hàm conj đổi dấu tọa độ y của một vectơ, và khi các vectơ $(x_1, -y_1)$ và (x_2, y_2) được nhân với nhau, tọa độ y của kết quả là $x_1y_2 - x_2y_1$.

Vị trí điểm

Tích có hướng có thể được dùng để kiểm tra một điểm nằm bên trái hay bên phải của một đường thẳng. Giả sử đường thẳng đi qua các điểm s_1 và s_2 , ta nhìn từ s_1 đến s_2 và điểm cần xét là p .

Ví dụ, trong hình sau, p nằm bên trái đường thẳng:

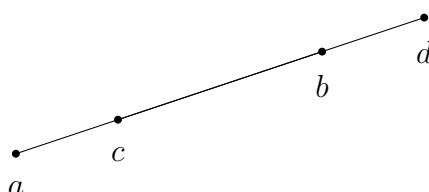


Tích có hướng $(p - s_1) \times (p - s_2)$ cho ta biết vị trí của điểm p . Nếu tích có hướng dương, p nằm bên trái, và nếu tích có hướng âm, p nằm bên phải. Cuối cùng, nếu tích có hướng bằng không, các điểm s_1, s_2 và p nằm trên cùng một đường thẳng.

Giao của đoạn thẳng

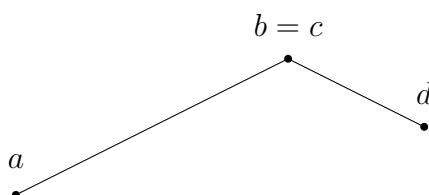
Tiếp theo ta xét bài toán kiểm tra hai đoạn thẳng ab và cd có cắt nhau hay không. Các trường hợp có thể là:

Trường hợp 1: Các đoạn thẳng nằm trên cùng một đường thẳng và chồng lên nhau. Trong trường hợp này, có vô số điểm giao. Ví dụ, trong hình sau, mọi điểm giữa c và b đều là điểm giao:



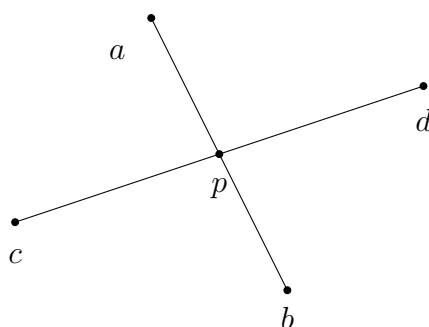
Trong trường hợp này, ta có thể dùng tích có hướng để kiểm tra mọi điểm có nằm trên cùng một đường thẳng hay không. Sau đó, ta có thể sắp xếp các điểm và kiểm tra hai đoạn thẳng có chồng lên nhau hay không.

Trường hợp 2: Các đoạn thẳng có một đỉnh chung và đó là điểm giao duy nhất. Ví dụ, trong hình sau, điểm giao là $b = c$:



Trường hợp này dễ kiểm tra, vì chỉ có bốn khả năng cho điểm giao: $a = c$, $a = d$, $b = c$ và $b = d$.

Trường hợp 3: Có đúng một điểm giao không phải là đỉnh của bất kỳ đoạn thẳng nào. Trong hình sau, điểm p là điểm giao:



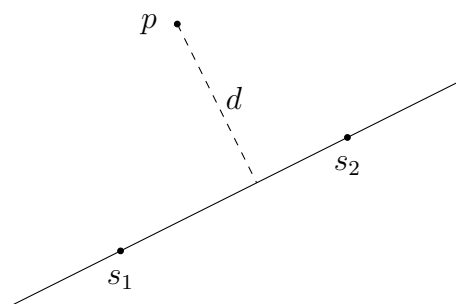
Trong trường hợp này, các đoạn thẳng cắt nhau khi và chỉ khi hai điểm c và d nằm ở hai phía khác nhau của đường thẳng qua a và b , và các điểm a và b nằm ở hai phía khác nhau của đường thẳng qua c và d . Ta có thể dùng tích có hướng để kiểm tra điều này.

Khoảng cách từ điểm đến đường thẳng

Một tính chất khác của tích có hướng là diện tích của một tam giác có thể được tính bằng công thức

$$\frac{|(a - c) \times (b - c)|}{2},$$

trong đó a , b và c là các đỉnh của tam giác. Dùng sự thật này, ta có thể suy ra công thức tính khoảng cách ngắn nhất giữa một điểm và một đường thẳng. Ví dụ, trong hình sau, d là khoảng cách ngắn nhất giữa điểm p và đường thẳng được xác định bởi các điểm s_1 và s_2 :

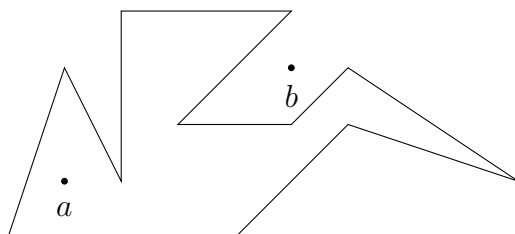


Diện tích của tam giác có các đỉnh s_1 , s_2 và p có thể được tính theo hai cách: nó vừa là $\frac{1}{2}|s_2 - s_1|d$ và $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Do đó, khoảng cách ngắn nhất là

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

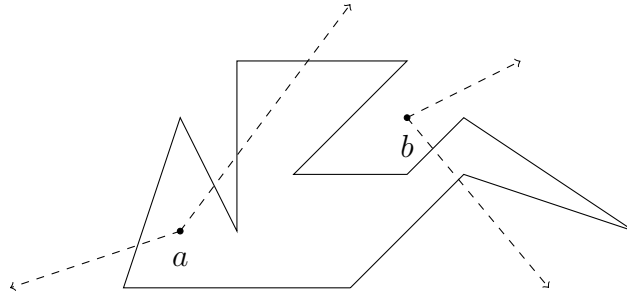
Điểm bên trong đa giác

Bây giờ xét bài toán kiểm tra một điểm nằm bên trong hay bên ngoài một đa giác. Ví dụ, trong hình sau, điểm a nằm bên trong đa giác và điểm b nằm bên ngoài đa giác.



Một cách thuận tiện để giải bài toán là bắn một tia từ điểm đó theo một hướng tùy ý và tính số lần tia chạm biên của đa giác. Nếu số lần là lẻ, điểm nằm bên trong đa giác, và nếu số lần là chẵn, điểm nằm bên ngoài đa giác.

Ví dụ, ta có thể bắn các tia sau:



Các tia từ a chạm biên đa giác 1 và 3 lần, nên a nằm bên trong đa giác. Tương ứng, các tia từ b chạm biên đa giác 0 và 2 lần, nên b nằm bên ngoài đa giác.

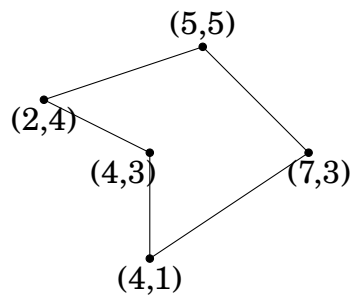
29.3 Diện tích đa giác

Một công thức tổng quát để tính diện tích của một đa giác, đôi khi gọi là **công thức dây giày (shoelace formula)**, là như sau:

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

Ở đây các đỉnh là $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \dots, p_n = (x_n, y_n)$ theo thứ tự sao cho p_i và p_{i+1} là các đỉnh kề nhau trên biên của đa giác, và đỉnh đầu tiên cùng đỉnh cuối cùng trùng nhau, tức là $p_1 = p_n$.

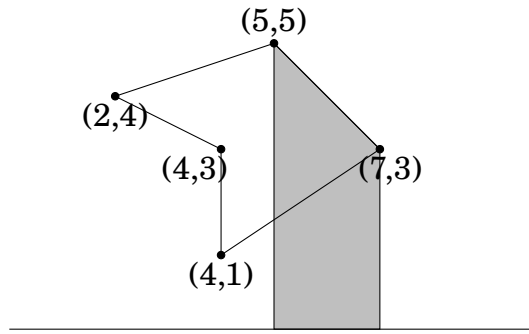
Ví dụ, diện tích của đa giác



là

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

Ý tưởng của công thức là duyệt qua các hình thang có một cạnh là một cạnh của đa giác, và một cạnh khác nằm trên đường ngang $y = 0$. Ví dụ:



Diện tích của một hình thang như vậy là

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

trong đó các đỉnh của đa giác là p_i và p_{i+1} . Nếu $x_{i+1} > x_i$, diện tích là dương, và nếu $x_{i+1} < x_i$, diện tích là âm.

Diện tích của đa giác là tổng diện tích của tất cả các hình thang như vậy, dẫn đến công thức

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Lưu ý rằng ta lấy giá trị tuyệt đối của tổng, vì giá trị của tổng có thể dương hoặc âm, tùy vào việc ta đi theo chiều kim đồng hồ hay ngược chiều kim đồng hồ dọc theo biên của đa giác.

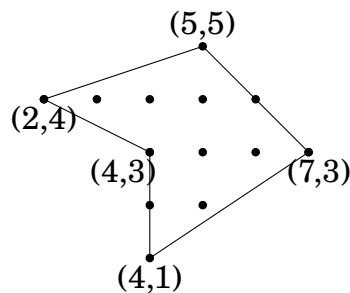
Định lý Pick

Định lý Pick (Pick's theorem) cung cấp một cách khác để tính diện tích của một đa giác với điều kiện mọi đỉnh của đa giác có tọa độ nguyên. Theo định lý Pick, diện tích của đa giác là

$$a + b/2 - 1,$$

trong đó a là số điểm nguyên bên trong đa giác và b là số điểm nguyên trên biên của đa giác.

Ví dụ, diện tích của đa giác



là $6 + 7/2 - 1 = 17/2$.

29.4 Hàm khoảng cách

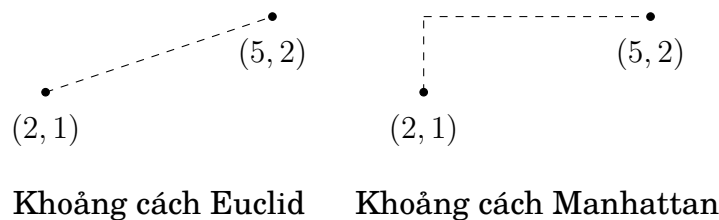
Một **hàm khoảng cách** định nghĩa khoảng cách giữa hai điểm. Hàm khoảng cách thông dụng là **khoảng cách Euclid (Euclidean distance)**, trong đó khoảng cách giữa các điểm (x_1, y_1) và (x_2, y_2) là

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Một hàm khoảng cách khác là **khoảng cách Manhattan (Manhattan distance)**, trong đó khoảng cách giữa các điểm (x_1, y_1) và (x_2, y_2) là

$$|x_1 - x_2| + |y_1 - y_2|.$$

Ví dụ, xét hình sau:



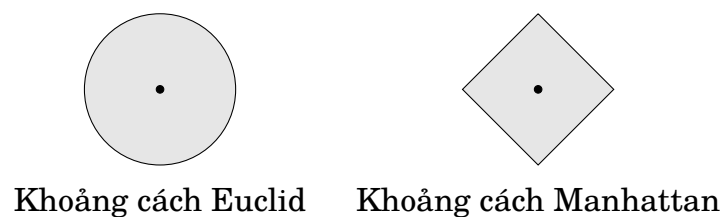
Khoảng cách Euclid giữa hai điểm là

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

và khoảng cách Manhattan là

$$|5 - 2| + |2 - 1| = 4.$$

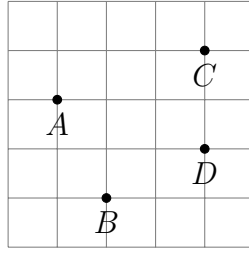
Hình sau cho thấy các vùng có khoảng cách không quá 1 từ điểm trung tâm, khi dùng khoảng cách Euclid và Manhattan:



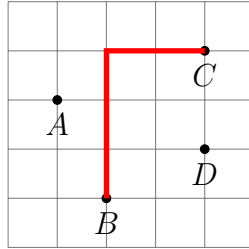
Xoay tọa độ

Một số bài toán dễ giải hơn nếu dùng khoảng cách Manhattan thay vì khoảng cách Euclid. Ví dụ, xét một bài toán trong đó ta được cho n điểm trên mặt phẳng hai chiều và nhiệm vụ là tính khoảng cách Manhattan lớn nhất giữa hai điểm bất kỳ.

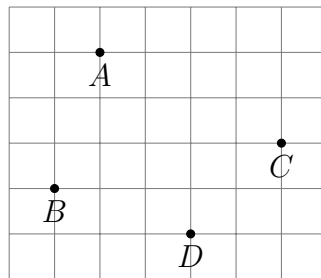
Ví dụ, xét tập điểm sau:



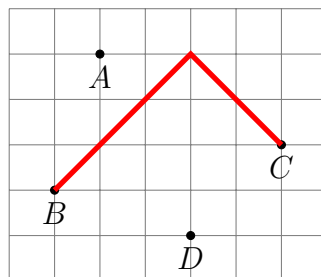
Khoảng cách Manhattan lớn nhất là 5 giữa các điểm B và C :



Một kỹ thuật hữu ích liên quan đến khoảng cách Manhattan là xoay mọi tọa độ 45 độ sao cho một điểm (x, y) trở thành $(x + y, y - x)$. Ví dụ, sau khi xoay các điểm trên, kết quả là:



Và khoảng cách lớn nhất là như sau:



Xét hai điểm $p_1 = (x_1, y_1)$ và $p_2 = (x_2, y_2)$ có tọa độ sau khi xoay là $p'_1 = (x'_1, y'_1)$ và $p'_2 = (x'_2, y'_2)$. Bây giờ có hai cách biểu diễn khoảng cách Manhattan giữa p_1 và p_2 :

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

Ví dụ, nếu $p_1 = (1, 0)$ và $p_2 = (3, 3)$, tọa độ sau khi xoay là $p'_1 = (1, -1)$ và $p'_2 = (6, 0)$, và khoảng cách Manhattan là

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

Tọa độ sau khi xoay cung cấp một cách đơn giản để thao tác với khoảng cách Manhattan, vì ta có thể xét riêng tọa độ x và y . Để cực đại hóa khoảng cách Manhattan giữa hai điểm, ta nên tìm hai điểm sao cho tọa độ sau khi xoay của chúng cực đại hóa giá trị

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

Điều này dễ làm, vì hoặc hiệu theo phương ngang hoặc hiệu theo phương dọc của các tọa độ sau khi xoay phải đạt lớn nhất.

Chương 30

Thuật toán đường quét (Sweep Line Algorithms)

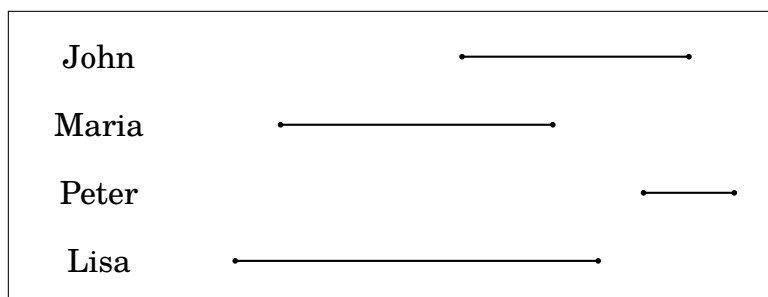
Nhiều bài toán hình học có thể được giải bằng các thuật toán **đường quét** (**sweep line**). Ý tưởng trong các thuật toán như vậy là biểu diễn một thể hiện của bài toán thành một tập sự kiện tương ứng với các điểm trên mặt phẳng. Các sự kiện được xử lý theo thứ tự tăng dần theo tọa độ x hoặc y .

Ví dụ, xét bài toán sau: Có một công ty với n nhân viên, và với mỗi nhân viên ta biết thời điểm đến và thời điểm rời đi trong một ngày nhất định. Nhiệm vụ là tính số lượng nhân viên lớn nhất cùng có mặt trong văn phòng tại một thời điểm.

Bài toán có thể được giải bằng cách mô hình hóa tình huống sao cho mỗi nhân viên được gán hai sự kiện tương ứng với thời điểm đến và rời đi của họ. Sau khi sắp xếp các sự kiện, ta duyệt qua chúng và theo dõi số người trong văn phòng. Ví dụ, bảng

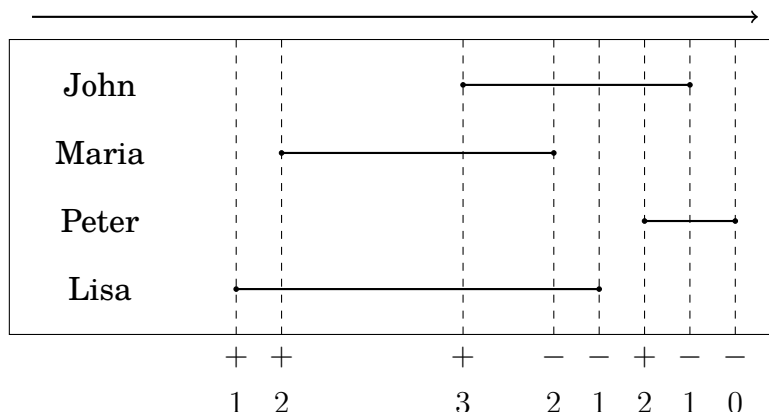
người	thời điểm đến	thời điểm đi
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

tương ứng với các sự kiện sau:



Ta duyệt các sự kiện từ trái sang phải và duy trì một bộ đếm. Mỗi khi một người đến, ta tăng giá trị bộ đếm lên một, và khi một người rời đi, ta giảm giá trị bộ đếm đi một. Đáp án của bài toán là giá trị lớn nhất của bộ đếm trong thuật toán.

Trong ví dụ, các sự kiện được xử lý như sau:

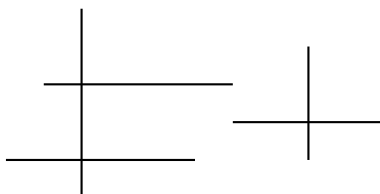


Các ký hiệu + và - cho biết giá trị của bộ đếm tăng hay giảm, và giá trị của bộ đếm được hiển thị bên dưới. Giá trị lớn nhất của bộ đếm là 3 giữa thời điểm John đến và Maria rời đi.

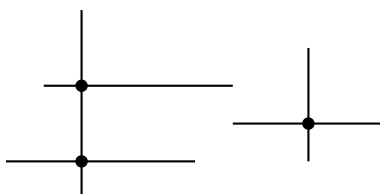
Thời gian chạy của thuật toán là $O(n \log n)$, vì việc sắp xếp các sự kiện mất thời gian $O(n \log n)$ và phần còn lại của thuật toán mất thời gian $O(n)$.

30.1 Điểm giao

Cho một tập gồm n đoạn thẳng, mỗi đoạn hoặc nằm ngang hoặc thẳng đứng, xét bài toán đếm tổng số điểm giao. Ví dụ, khi các đoạn thẳng là



có ba điểm giao:



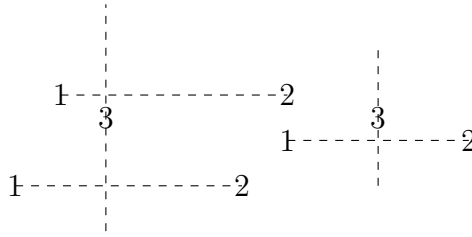
Rất dễ giải bài toán trong thời gian $O(n^2)$, vì ta có thể duyệt qua mọi cặp đoạn thẳng có thể và kiểm tra chúng có cắt nhau hay không. Tuy nhiên, ta có thể giải bài toán hiệu quả hơn trong thời gian $O(n \log n)$ bằng thuật toán đường quét và một cấu trúc dữ liệu truy vấn đoạn.

Ý tưởng là xử lý các đầu mút của các đoạn thẳng từ trái sang phải và tập trung vào ba loại sự kiện:

- (1) đoạn ngang bắt đầu
- (2) đoạn ngang kết thúc

(3) đoạn dọc

Các sự kiện sau tương ứng với ví dụ:



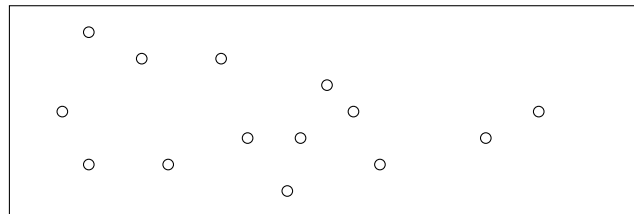
Ta duyệt các sự kiện từ trái sang phải và dùng một cấu trúc dữ liệu duy trì tập các tọa độ y nơi có một đoạn ngang đang hoạt động. Ở sự kiện 1, ta thêm tọa độ y của đoạn vào tập, và ở sự kiện 2, ta xóa tọa độ y khỏi tập.

Các điểm giao được tính ở sự kiện 3. Khi có một đoạn dọc giữa các điểm y_1 và y_2 , ta đếm số đoạn ngang đang hoạt động có tọa độ y nằm giữa y_1 và y_2 , rồi cộng số này vào tổng số điểm giao.

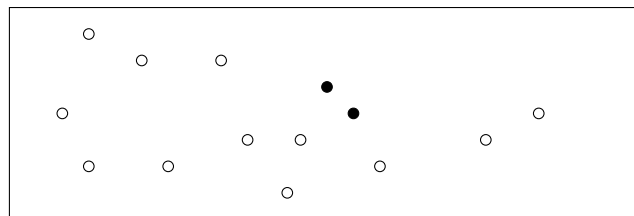
Để lưu tọa độ y của các đoạn ngang, ta có thể dùng cây Fenwick hoặc cây phân đoạn, có thể kèm theo nén chỉ số. Khi dùng các cấu trúc như vậy, xử lý mỗi sự kiện mất thời gian $O(\log n)$, nên tổng thời gian chạy của thuật toán là $O(n \log n)$.

30.2 Bài toán cặp gần nhất

Cho một tập gồm n điểm, bài toán tiếp theo là tìm hai điểm có khoảng cách Euclid nhỏ nhất. Ví dụ, nếu các điểm là



ta cần tìm các điểm sau:



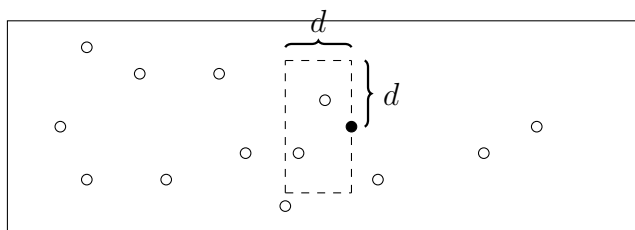
Đây là một ví dụ khác về bài toán có thể được giải trong thời gian $O(n \log n)$ bằng thuật toán đường quét¹. Ta duyệt các điểm từ trái sang phải và duy trì một giá trị d : khoảng cách nhỏ nhất giữa hai điểm đã thấy cho đến hiện tại.

¹Ngoài cách tiếp cận này, cũng có một thuật toán chia để trị thời gian $O(n \log n)$ [56] chia các điểm thành hai tập và giải đệ quy bài toán cho cả hai tập.

Ở mỗi điểm, ta tìm điểm gần nhất ở bên trái. Nếu khoảng cách nhỏ hơn d , đó là khoảng cách nhỏ nhất mới và ta cập nhật giá trị của d .

Nếu điểm hiện tại là (x, y) và có một điểm ở bên trái nằm trong khoảng cách nhỏ hơn d , tọa độ x của điểm đó phải nằm giữa $[x - d, x]$ và tọa độ y phải nằm giữa $[y - d, y + d]$. Do đó, chỉ cần xét các điểm nằm trong các khoảng này, nhờ vậy thuật toán hiệu quả.

Ví dụ, trong hình sau, vùng được đánh dấu bằng các đường đứt nét chứa các điểm có thể nằm trong khoảng cách d từ điểm đang hoạt động:



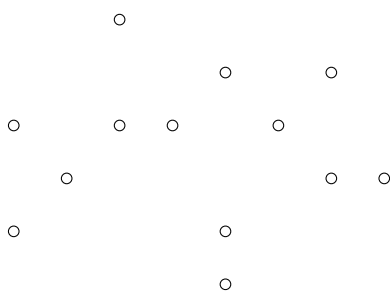
Tính hiệu quả của thuật toán dựa trên sự thật rằng vùng này luôn chỉ chứa $O(1)$ điểm. Ta có thể duyệt các điểm đó trong thời gian $O(\log n)$ bằng cách duy trì một tập các điểm có tọa độ x nằm giữa $[x - d, x]$, theo thứ tự tăng dần của tọa độ y.

Độ phức tạp thời gian của thuật toán là $O(n \log n)$, vì ta duyệt qua n điểm và với mỗi điểm tìm điểm gần nhất ở bên trái trong thời gian $O(\log n)$.

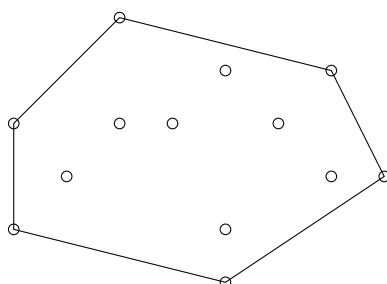
30.3 Bài toán bao lồi

Bao lồi (convex hull) là đa giác lồi nhỏ nhất chứa tất cả các điểm của một tập cho trước. Tính lồi nghĩa là đoạn thẳng nối bất kỳ hai đỉnh nào của đa giác đều nằm hoàn toàn bên trong đa giác.

Ví dụ, với các điểm



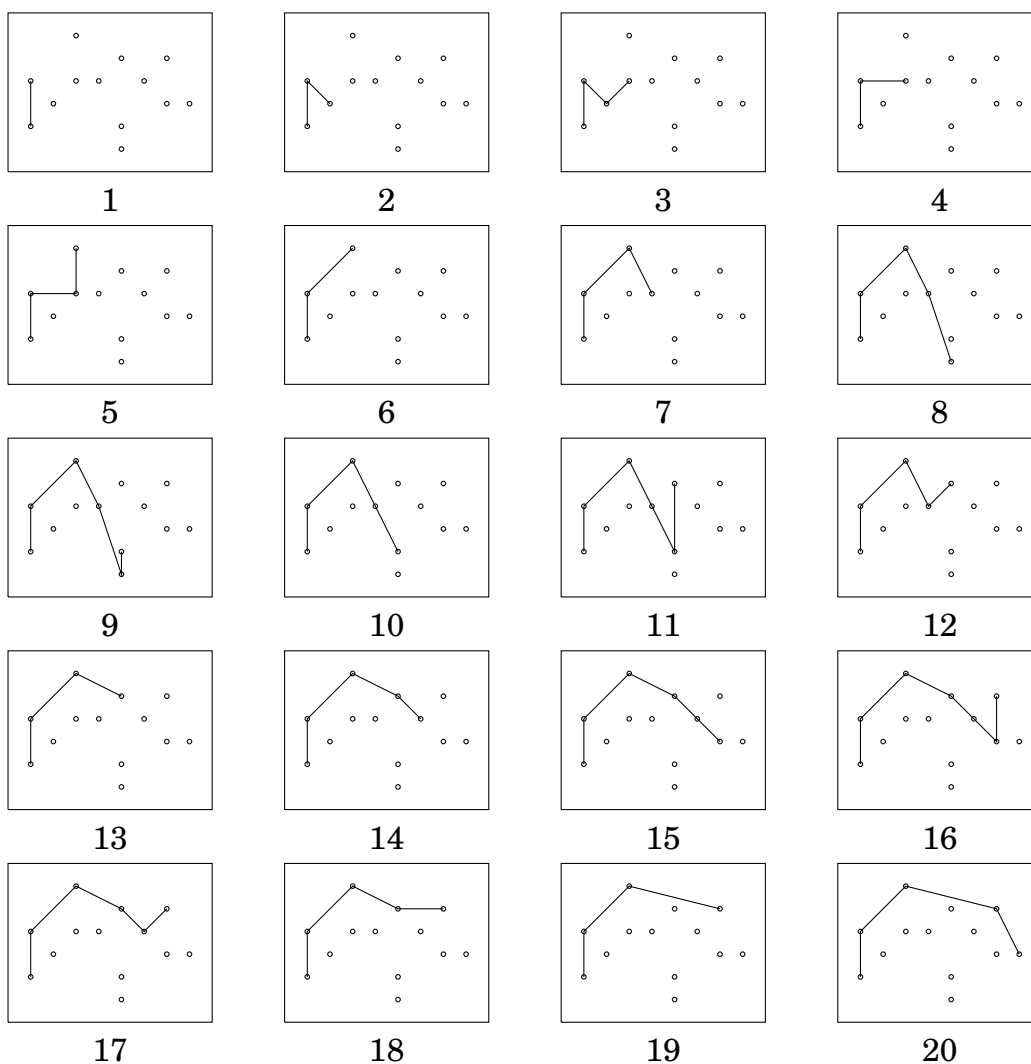
bao lồi là như sau:



Thuật toán Andrew (Andrew's algorithm) [3] cung cấp một cách dễ để xây dựng bao lồi cho một tập điểm trong thời gian $O(n \log n)$. Thuật toán trước hết tìm điểm trái nhất và điểm phải nhất, rồi xây dựng bao lồi thành hai phần: trước hết là bao trên, sau đó là bao dưới. Hai phần này tương tự nhau, nên ta có thể tập trung vào việc xây dựng bao trên.

Trước hết, ta sắp xếp các điểm chủ yếu theo tọa độ x và phụ theo tọa độ y. Sau đó, ta duyệt qua các điểm và thêm từng điểm vào bao. Mỗi khi thêm một điểm vào bao, ta bảo đảm rằng đoạn thẳng cuối cùng trong bao không rẽ trái. Chừng nào nó còn rẽ trái, ta liên tục xóa điểm áp chót khỏi bao.

Các hình sau cho thấy thuật toán Andrew hoạt động như thế nào:



Tài liệu tham khảo

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).

- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>

- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.

- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).
- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).
- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.

- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

Chỉ mục

- 2SAT problem, 168
- 2SUM problem, 84
- 3SAT problem, 170
- 3SUM problem, 85

- adjacency list, 119
- adjacency matrix, 120
- alphabet, 253
- amortized analysis, 83
- ancestor, 171
- and operation, 102
- Andrew's algorithm, 291
- antichain, 203
- arithmetic progression, 11

- backtracking, 56
- Bellman–Ford algorithm, 131
- binary code, 68
- binary indexed tree, 92
- binary search, 34
- binary tree, 147
- Binet's formula, 14
- binomial coefficient, 218
- binomial distribution, 240
- bipartite graph, 118, 128
- birthday paradox, 257
- bit representation, 101
- bit shift, 103
- bitset, 46
- border, 254
- breadth-first search, 125
- bubble sort, 27
- Burnside's lemma, 224

- Catalan number, 220
- Cayley's formula, 225
- child, 141
- Chinese remainder theorem, 215

- closest pair, 289
- codeword, 68
- cofactor, 229
- collision, 256
- coloring, 118, 243
- combinatorics, 217
- comparison function, 34
- comparison operator, 33
- complement, 12
- complete graph, 117
- complex, 276
- complex number, 276
- complexity classes, 22
- component, 116
- component graph, 165
- conditional probability, 237
- conjunction, 13
- connected graph, 116, 127
- constant factor, 23
- constant-time algorithm, 22
- coprime, 211
- counting sort, 30
- cross product, 278
- cubic algorithm, 22
- cut, 192
- cycle, 116, 128, 157, 163
- cycle detection, 163

- data compression, 68
- data structure, 39
- De Bruijn sequence, 188
- degree, 117
- depth-first search, 123
- deque, 47
- derangement, 223
- determinant, 229
- diameter, 143

difference, 12
 difference array, 100
 Dijkstra's algorithm, 134, 161
 Dilworth's theorem, 203
 Diophantine equation, 214
 Dirac's theorem, 188
 directed graph, 116
 disjunction, 13
 distance function, 283
 distribution, 239
 divisibility, 207
 divisor, 207
 dynamic array, 39
 dynamic programming, 71
 dynamic segment tree, 271

 edge, 115
 edge list, 121
 edit distance, 80
 Edmonds–Karp algorithm, 195
 equivalence, 13
 Euclid's algorithm, 210
 Euclid's formula, 216
 Euclidean distance, 283
 Euler tour technique, 176
 Euler's theorem, 212
 Euler's totient function, 211
 Eulerian circuit, 184
 Eulerian path, 183
 expected value, 238
 extended Euclid's algorithm, 214

 factor, 207
 factorial, 14
 Faulhaber's formula, 10
 Fenwick tree, 92
 Fermat's theorem, 212
 Fibonacci number, 14, 216, 230
 floating point number, 7
 flow, 191
 Floyd's algorithm, 164
 Floyd–Warshall algorithm, 137
 Ford–Fulkerson algorithm, 192
 Freivalds' algorithm, 242
 functional graph, 162

 geometric distribution, 240
 geometric progression, 11
 geometry, 275
 Goldbach's conjecture, 209
 graph, 115
 greatest common divisor, 210
 greedy algorithm, 63
 Grundy number, 249
 Grundy's game, 251

 Hall's theorem, 199
 Hamiltonian circuit, 187
 Hamiltonian path, 187
 Hamming distance, 106
 harmonic sum, 11, 210
 hash value, 255
 hashing, 255
 heap, 48
 Heron's formula, 275
 heuristic, 190
 Hierholzer's algorithm, 185
 Huffman coding, 69

 identity matrix, 228
 implication, 13
 in-order, 147
 inclusion-exclusion, 222
 indegree, 117
 independence, 238
 independent set, 200
 index compression, 99
 input and output, 4
 integer, 6
 intersection, 12
 intersection point, 288
 inverse matrix, 230
 inversion, 28
 iterator, 43

 Kadane's algorithm, 25
 Kirchhoff's theorem, 233
 knapsack, 78
 knight's tour, 189
 Kosaraju's algorithm, 166
 Kruskal's algorithm, 150
 König's theorem, 200

 Lagrange's theorem, 215
 Laplacean matrix, 234

Las Vegas algorithm, 241
 lazy propagation, 268
 lazy segment tree, 268
 leaf, 141
 least common multiple, 210
 Legendre's conjecture, 209
 Levenshtein distance, 80
 lexicographical order, 254
 line segment intersection, 279
 linear algorithm, 22
 linear recurrence, 230
 logarithm, 14
 logarithmic algorithm, 22
 logic, 13
 longest increasing subsequence, 76
 losing state, 245
 lowest common ancestor, 175

 macro, 9
 Manhattan distance, 283
 map, 42
 Markov chain, 240
 matching, 198
 matrix, 227
 matrix multiplication, 228, 242
 matrix power, 229
 maximum flow, 191
 maximum independent set, 200
 maximum matching, 198
 maximum query, 89
 maximum spanning tree, 150
 maximum subarray sum, 24
 meet in the middle, 60
 memoization, 73
 merge sort, 29
 mex function, 249
 minimum cut, 192, 195
 minimum node cover, 200
 minimum query, 89
 minimum spanning tree, 149
 misère game, 248
 Mo's algorithm, 265
 modular arithmetic, 6, 211
 modular inverse, 213
 Monte Carlo algorithm, 241
 multinomial coefficient, 220

 natural logarithm, 15
 nearest smaller elements, 85
 negation, 13
 negative cycle, 133
 neighbor, 117
 next_permutation, 55
 nim game, 247
 nim sum, 247
 node, 115
 node cover, 200
 not operation, 103
 NP-hard problem, 23
 number theory, 207

 or operation, 102
 order statistic, 242
 Ore's theorem, 188
 outdegree, 117

 pair, 33
 parent, 141
 parenthesis expression, 221
 Pascal's triangle, 219
 path, 115
 path cover, 201
 pattern matching, 253
 perfect matching, 199
 perfect number, 208
 period, 254
 permutation, 55
 persistent segment tree, 272
 Pick's theorem, 282
 point, 276
 polynomial algorithm, 23
 polynomial hashing, 255
 post-order, 147
 pre-order, 147
 predicate, 13
 prefix, 253
 prefix sum array, 90
 Prim's algorithm, 155
 prime, 207
 prime decomposition, 207
 priority queue, 48
 probability, 235
 programming language, 3
 Prüfer code, 226

- Pythagorean triple, 216
- quadratic algorithm, 22
- quantifier, 13
- queen problem, 56
- queue, 48
- quickselect, 242
- quicksort, 242
- random variable, 238
- random_shuffle, 43
- randomized algorithm, 241
- range query, 89
- regular graph, 117
- remainder, 6
- reverse, 43
- root, 141
- rooted tree, 141
- rotation, 253
- scaling algorithm, 195
- segment tree, 95, 267
- set, 12, 41
- set theory, 12
- shoelace formula, 281
- shortest path, 131
- sieve of Eratosthenes, 210
- simple graph, 119
- sliding window, 87
- sliding window minimum, 87
- sort, 32, 43
- sorting, 27
- spanning tree, 149, 233
- sparse segment tree, 271
- sparse table, 91
- SPFA algorithm, 134
- Sprague–Grundy theorem, 248
- square matrix, 227
- square root algorithm, 261
- stack, 47
- string, 40, 253
- string hashing, 255
- strongly connected component, 165
- strongly connected graph, 165
- subsequence, 253
- subset, 12, 53
- substring, 253
- subtree, 141
- successor graph, 162
- suffix, 253
- sum query, 89
- sweep line, 287
- time complexity, 19
- topological sorting, 157
- transpose, 227
- tree, 116, 141
- tree query, 171
- tree traversal array, 172
- trie, 254
- tuple, 33
- typedef, 8
- twin prime, 209
- two pointers method, 83
- two-dimensional segment tree, 274
- uniform distribution, 239
- union, 12
- union-find structure, 153
- universal set, 12
- vector, 39, 227, 276
- Warnsdorf’s rule, 190
- weighted graph, 117
- Wilson’s theorem, 216
- winning state, 245
- xor operation, 103
- Z-algorithm, 258
- Z-array, 258
- Zeckendorf’s theorem, 216