

Competitive Programmer's Handbook

Antti Laaksonen (deanqkhanh dịch)

Draft Ngày 17 tháng 9 năm 2025

Mục lục

Lời nói đầu	vii
I Basic techniques	1
1 Giới thiệu	2
1.1 Ngôn ngữ lập trình	2
1.2 Nhập và xuất	3
1.3 Làm việc với số	4
1.4 Viết code ngắn gọn	6
1.5 Toán học	8
1.6 Các cuộc thi và tài nguyên	12
2 Độ phức tạp thời gian	15
2.1 Quy tắc tính toán	15
2.2 Các lớp độ phức tạp	17
2.3 Ước tính hiệu quả	18
2.4 Tổng con lớn nhất	19
3 Sắp xếp	22
3.1 Lý thuyết sắp xếp	22
3.2 Sắp xếp trong C++	26
3.3 Tìm kiếm nhị phân	28
4 Cấu trúc dữ liệu	31
4.1 Mảng động	31
4.2 Cấu trúc tập hợp	32
4.3 Cấu trúc map	34
4.4 Iterator và khoảng	34
4.5 Các cấu trúc khác	36
4.6 So sánh với sắp xếp	40
5 Tìm kiếm toàn diện	42
5.1 Tạo tập con	42
5.2 Tạo hoán vị	43
5.3 Quay lui	44
5.4 Cắt tỉa tìm kiếm	46
5.5 Gặp nhau ở giữa	49

6 Thuật toán tham lam	50
6.1 Bài toán đổi tiền	50
6.2 Lập lịch	51
6.3 Nhiệm vụ và thời hạn	53
6.4 Giảm thiểu tổng	54
6.5 Nén dữ liệu	54
7 Quy hoạch động	57
7.1 Bài toán đồng xu	57
7.2 Dây con tăng dài nhất	62
7.3 Đường đi trong lưới	63
7.4 Bài toán cái túi	64
7.5 Khoảng cách chỉnh sửa	65
7.6 Đếm số cách lát gạch	66
8 Phân tích trừ dần	68
8.1 Phương pháp hai con trỏ	68
8.2 Phần tử nhỏ hơn gần nhất	70
8.3 Giá trị nhỏ nhất trong cửa sổ trượt	71
9 Truy vấn đoạn	73
9.1 Truy vấn trên mảng tĩnh	73
9.2 Cây chỉ số nhị phân (Binary indexed tree)	76
9.3 Cây đoạn (Segment tree)	78
9.4 Các kỹ thuật bổ sung	82
10 Thao tác bit	84
10.1 Biểu diễn bit	84
10.2 Các phép toán bit	85
10.3 Biểu diễn tập hợp	87
10.4 Tối ưu hóa bit	88
10.5 Quy hoạch động	90
II Graph algorithms	95
11 Những điều cơ bản về đồ thị	96
11.1 Thuật ngữ đồ thị	96
11.2 Biểu diễn đồ thị	99
12 Duyệt đồ thị	103
12.1 Tìm kiếm theo chiều sâu	103
12.2 Tìm kiếm theo chiều rộng	104
12.3 Ứng dụng	106
13 Đường đi ngắn nhất	109
13.1 Thuật toán Bellman–Ford	109
13.2 Thuật toán Dijkstra	112
13.3 Thuật toán Floyd–Warshall	114

14 Các thuật toán trên cây	118
14.1 Duyệt cây	119
14.2 Đường kính	120
14.3 Tất cả các đường đi dài nhất	121
14.4 Cây nhị phân	123
15 Cây khung	125
15.1 Thuật toán Kruskal	126
15.2 Cấu trúc Union-find	129
15.3 Thuật toán Prim	130
16 Đồ thị có hướng	133
16.1 Sắp xếp tô pô (Topological sorting)	133
16.2 Quy hoạch động (Dynamic programming)	135
16.3 Đường đi kế tiếp	137
16.4 Đồ tìm chu trình (Cycle detection)	139
17 Liên thông mạnh (Strong connectivity)	141
17.1 Thuật toán Kosaraju	142
17.2 Bài toán 2SAT	144
18 Truy vấn trên cây	146
18.1 Tìm tổ tiên	146
18.2 Cây con và đường đi	147
18.3 Tổ tiên chung thấp nhất	149
18.4 Thuật toán ngoại tuyến	152
19 Đường đi và chu trình	155
19.1 Đường đi Euler	155
19.2 Đường đi Hamilton	159
19.3 Chuỗi De Bruijn	160
19.4 Hành trình của quân mã	160
20 Luồng và lát cắt (Flows and cuts)	162
20.1 Thuật toán Ford–Fulkerson	163
20.2 Đường đi không giao nhau (Disjoint paths)	166
20.3 Maximum matchings	168
20.4 Path covers	170
III Advanced topics	174
21 Lý thuyết số	175
21.1 Số nguyên tố và ước số	175
21.2 Số học mô-đun	179
21.3 Giải phương trình	181
21.4 Các kết quả khác	183

22 Tổ hợp	185
22.1 Tổ hợp chập (Binomial coefficients)	186
22.2 Số Catalan	188
22.3 Bao hàm - loại trừ	190
22.4 Bổ đề Burnside	192
22.5 Công thức Cayley	192
23 Ma trận	195
23.1 Các phép toán	195
23.2 Hệ thức truy hồi tuyến tính	198
23.3 Đồ thị và ma trận	199
24 Xác suất (Probability)	202
24.1 Tính toán	202
24.2 Biến cố (Events)	203
24.3 Biến ngẫu nhiên (Random variables)	205
24.4 Chuỗi Markov (Markov chains)	207
24.5 Thuật toán ngẫu nhiên (Randomized algorithms)	208
25 Lý thuyết trò chơi (Game theory)	211
25.1 Trạng thái trò chơi (Game states)	211
25.2 Trò chơi Nim (Nim game)	213
25.3 Định lý Sprague–Grundy (Sprague–Grundy theorem)	214
26 Các thuật toán chuỗi	218
26.1 Thuật ngữ chuỗi	218
26.2 Cấu trúc Trie	219
26.3 Băm chuỗi (String hashing)	220
26.4 Thuật toán Z (Z-algorithm)	222
27 Thuật toán căn bậc hai	226
27.1 Kết hợp các thuật toán	227
27.2 Phân hoạch số nguyên	228
27.3 Thuật toán của Mo (Mo's algorithm)	230
28 Cây phân đoạn nâng cao	232
28.1 Truyền lười (Lazy propagation)	233
28.2 Cây động	236
28.3 Cấu trúc dữ liệu	238
28.4 Hai chiều	238
29 Hình học	240
29.1 Số phức	241
29.2 Điểm và đường thẳng	242
29.3 Diện tích đa giác	245
29.4 Hàm khoảng cách	246
30 Thuật toán đường quét	249
30.1 Giao điểm	250
30.2 Bài toán cặp điểm gần nhất	251
30.3 Bài toán bao lồi	252

Lời nói đầu

Mục đích của cuốn sách này là giúp bạn có một cái nhìn tổng quan về lập trình thi đấu. Chúng tôi giả định rằng bạn đã biết những kiến thức cơ bản về lập trình, nhưng không cần có kinh nghiệm trước đây về lập trình thi đấu.

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

Để trở thành một lập trình viên thi đấu giỏi cần rất nhiều thời gian, nhưng đây cũng là cơ hội để học hỏi rất nhiều điều. Bạn có thể chắc chắn rằng bạn sẽ có được một hiểu biết tổng quát tốt về thuật toán nếu bạn dành thời gian đọc cuốn sách này, giải các bài tập và tham gia các cuộc thi.

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

Helsinki, tháng 8 năm 2019
Antti Laaksonen

Phần I

Basic techniques

Chương 1

Giới thiệu

Lập trình thi đấu kết hợp hai chủ đề: (1) thiết kế thuật toán và (2) cài đặt thuật toán.

Thiết kế thuật toán bao gồm việc giải quyết vấn đề và tư duy toán học. Cần có kỹ năng phân tích vấn đề và giải quyết chúng một cách sáng tạo. Một thuật toán để giải quyết một vấn đề phải vừa đúng vừa hiệu quả, và trọng tâm của vấn đề thường là về việc tìm ra một thuật toán hiệu quả.

Kiến thức lý thuyết về thuật toán rất quan trọng đối với các lập trình viên thi đấu. Thông thường, lời giải cho một bài toán là sự kết hợp của các kỹ thuật đã biết và những hiểu biết mới. Các kỹ thuật xuất hiện trong lập trình thi đấu cũng tạo nên nền tảng cho nghiên cứu khoa học về thuật toán.

Cài đặt thuật toán đòi hỏi kỹ năng lập trình tốt. Trong lập trình thi đấu, các lời giải được chấm điểm bằng cách kiểm tra thuật toán đã cài đặt bằng một tập các test case. Do đó, không chỉ cần ý tưởng của thuật toán đúng, mà việc cài đặt cũng phải chính xác.

Một phong cách code tốt trong các cuộc thi là đơn giản và súc tích. Chương trình cần được viết nhanh, vì không có nhiều thời gian. Khác với kỹ thuật phần mềm truyền thống, các chương trình ngắn gọn (thường nhiều nhất là vài trăm dòng code), và không cần phải bảo trì sau cuộc thi.

1.1 Ngôn ngữ lập trình

Hiện tại, các ngôn ngữ lập trình phổ biến nhất được sử dụng trong các cuộc thi là C++, Python và Java. Ví dụ, trong Google Code Jam 2017, trong số 3.000 người tham gia giỏi nhất, 79% sử dụng C++, 16% sử dụng Python và 8% sử dụng Java [29]. Một số người tham gia cũng sử dụng nhiều ngôn ngữ.

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

Mặt khác, việc thành thạo nhiều ngôn ngữ và hiểu rõ điểm mạnh của chúng là điều tốt. Ví dụ, nếu cần xử lý số nguyên lớn trong bài toán, Python có thể là một lựa chọn tốt, vì nó có sẵn các phép toán tích hợp để tính toán với số nguyên lớn. Tuy nhiên, hầu hết các bài toán trong các cuộc thi lập trình được thiết kế sao cho việc sử dụng một ngôn ngữ lập trình cụ thể không tạo ra lợi thế không công bằng.

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

Template code C++

Một template code C++ điển hình cho lập trình thi đấu trông như thế này:

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

using namespace std;

int main() {
    // phan giai bai toan o day
}
```

Dòng `#include` ở đầu mã nguồn là một tính năng của trình biên dịch `g++` cho phép chúng ta include toàn bộ thư viện chuẩn. Do đó, không cần phải include riêng lẻ các thư viện như `iostream`, `vector` và `algorithm`, mà chúng được sử dụng tự động.

Dòng `using` khai báo rằng các lớp và hàm của thư viện chuẩn có thể được sử dụng trực tiếp trong mã nguồn. Nếu không có dòng `using`, chúng ta sẽ phải viết, ví dụ, `std::cout`, nhưng bây giờ chỉ cần viết `cout`.

Mã nguồn có thể được biên dịch bằng lệnh sau:

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

Lệnh này tạo ra một file nhị phân `test` từ mã nguồn `test.cpp`. Trình biên dịch tuân theo chuẩn `C++11` (`-std=c++11`), tối ưu hóa mã nguồn (`-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, luồng chuẩn được sử dụng để đọc dữ liệu vào và ghi kết quả ra. Trong C++, các luồng chuẩn là `cin` cho nhập và `cout` cho xuất. Ngoài ra, các hàm `C` `scanf` và `printf` cũng có thể được sử dụng.

Dữ liệu vào cho chương trình thường bao gồm các số và chuỗi được phân tách bằng khoảng trắng và dòng mới. Chúng có thể được đọc từ luồng `cin` như sau:

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

Kiểu mã nguồn này luôn hoạt động, giả sử rằng có ít nhất một khoảng trắng hoặc dòng mới giữa các phần tử trong dữ liệu vào. Ví dụ, mã nguồn trên có thể đọc cả hai dữ liệu vào sau:

```
123 456 monkey
```

```
123    456
monkey
```

Luồng `cout` được sử dụng cho đầu ra 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 trong chương trình. Các dòng sau đây ở đầu mã nguồn giúp việc nhập và xuất hiệu quả hơn:

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

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

Các hàm C scanf và printf là một lựa chọn thay thế cho các luồng chuẩn C++. Chúng thường nhanh hơn một chút, nhưng cũng khó sử dụng hơn. Mã nguồn sau đây đọc hai số nguyên từ đầu vào:

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

Mã nguồn sau đây 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 đầu vào, có thể chứa các khoảng trắng. Điều này có thể thực hiện được bằng cách sử dụng hàm getline:

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

Nếu không biết lượng dữ liệu là bao nhiêu, vòng lặp sau đây rất hữu ích:

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

Vòng lặp này đọc các phần tử từ đầu vào lần lượt từng cái một, cho đến khi không còn dữ liệu nào trong đầu vào.

Trong một số hệ thống thi, file được sử dụng cho đầu vào và đầu ra. Một giải pháp đơn giản là viết mã nguồn như bình thường sử dụng luồng chuẩn, nhưng thêm các dòng sau vào đầu mã nguồn:

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

Sau đó, chương trình sẽ đọc dữ liệu vào từ file "input.txt" và ghi kết quả ra file "output.txt".

1.3 Làm việc với số

Số nguyên

Kiểu số nguyên được sử dụng nhiều nhất trong lập trình thi đấu là int, đây là kiểu 32-bit với phạm vi giá trị từ $-2^{31} \dots 2^{31} - 1$ hoặc khoảng $-2 \cdot 10^9 \dots 2 \cdot 10^9$. Nếu kiểu int không đủ,

kiểu 64-bit long long có thể được sử dụng. Nó có phạm vi giá trị từ $-2^{63} \dots 2^{63} - 1$ hoặc khoảng $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

Mã nguồn sau đây 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 sử dụng kiểu long long là kiểu int vẫn được sử dụng ở đâu đó trong mã nguồn. Ví dụ, mã nguồn sau chứa một lỗi tinh vi:

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

Mặc 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 điều này, biến b sẽ chứa một kết quả sai. Vấn đề có thể được giải quyết bằng cách thay đổi kiểu của a thành long long hoặc bằng cách thay đổi biểu thức thành $(\text{long long})a*a$.

Thông thường các bài toán trong cuộc thi được thiết lập sao cho kiểu long long là đủ. Tuy nhiên, tốt để biết rằng trình biên dịch g++ cũng cung cấp một kiểu 128-bit `__int128_t` với phạm vi giá trị từ $-2^{127} \dots 2^{127} - 1$ hoặc khoảng $-10^{38} \dots 10^{38}$. Tuy nhiên, kiểu này không có sẵn trong tất cả các hệ thống thi.

Số học modulo

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

Đôi khi, câu trả lời của một bài toán là một số rất lớn nhưng chỉ cần xuất ra nó "modulo m ", tức là, phần dư khi câu trả lời được chia cho m (ví dụ, "modulo $10^9 + 7$ "). Ý tưởng là kể cả khi câu trả lời thực tế rất lớn, chỉ cần sử dụng các kiểu int và long long là đủ.

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

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

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

Ví dụ, mã nguồn 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 chúng 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 là 0 hoặc số âm. Một cách đơn giản để đảm bảo không có phần dư âm là trước tiên tính phần dư như bình thường và sau đó cộng thêm m nếu kết quả là âm:

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

Tuy nhiên, điều này chỉ cần thiết khi có phép trừ trong code và phần còn lại có thể trở thành số âm.

Số thực dấu phẩy động

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

Độ chính xác yêu cầu của câu trả lời thường được cho trong đề bài. Một cách dễ dàng để xuất câu trả lời là sử dụng hàm printf và chỉ định số chữ số thập phân trong chuỗi định dạng. Ví dụ, mã nguồn sau in giá trị của x với 9 chữ số thập phân:

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

Một khó khăn khi sử dụng số thực dấu phẩy động là một số giá trị không thể được biểu diễn chính xác dưới dạng số thực dấu phẩy động, và sẽ có các lỗi làm tròn. Ví dụ, kết quả của mã nguồn sau khá bất ngờ:

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

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

Việc so sánh các số thực dấu phẩy động bằng toán tử `==` là rất nguy hiểm, bởi vì có thể các giá trị đáng lẽ phải bằng nhau nhưng lại không bằng do lỗi độ chính xác. Một cách tốt hơn để so sánh số thực dấu phẩy động là giả định rằng hai số bằng nhau nếu hiệu của chúng nhỏ hơn ϵ , trong đó ϵ là một số nhỏ.

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

```
if (abs(a-b) < 1e-9) {
    // a và b bằng nhau
}
```

Lưu ý rằng mặc dù số thực dấu phẩy động không chính xác, các số nguyên cho đến một giới hạn nhất định vẫn có thể được biểu diễn chính xác. Ví dụ, sử dụng double, có thể biểu diễn chính xác tất cả các số nguyên có giá trị tuyệt đối tối đa là 2^{53} .

1.4 Viết code ngắn gọn

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

Tên kiểu dữ liệu

Sử dụng 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 code

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

có thể được viết ngắn gọn như sau:

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

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

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

Macro

Một cách khác để viết code ngắn gọn là định nghĩa **macro**. Macro có nghĩa là một số chuỗi trong code 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 code

```
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 viết ngắn gọn như sau:

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

Một macro cũng có thể có tham số điều này cho phép viết ngắn gọn các 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 code

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

```
}
```

có thể được viết ngắn gọn như sau:

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

Đôi khi macro có thể gây ra lỗi khó phát hiện. Ví dụ, xem xét macro sau để 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 code

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

tương ứng với đoạn code

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

Một phiên bản tốt hơn của macro là:

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

Bây giờ đoạn code

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

tương ứng với đoạn code

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

1.5 Toán học

Toán học đóng một vai trò quan trọng trong lập trình thi đấu, và không thể trở thành một lập trình viên thi đấu thành công mà không có kỹ năng toán học tốt. Phần này thảo luận về một số khái niệm và công thức toán học quan trọng sẽ được sử dụng sau này trong cuốn sách.

Các 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, đều có một công thức dạng đóng là một đa thức bậc $k+1$. Ví dụ¹,

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

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

và

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

Cấp số cộng là một dãy số trong đó hiệu giữa hai số liên tiếp bất kỳ là một hằng số. Ví dụ,

$$3, 7, 11, 15$$

là một cấp số cộng với công sai là 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 hạng. Ví dụ,

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

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

Cấp số nhân là một dãy số trong đó tỷ số giữa hai số liên tiếp bất kỳ là một hằng số. Ví dụ,

$$3, 6, 12, 24$$

là một cấp số nhân với công bội là 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 tỷ số giữa các 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. Hãy đặt

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

Bằng cách nhân cả hai vế với k , ta được

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

và giải phương trình

$$kS - S = bk - a$$

ta được công thức trên.

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 là một tổng có dạng

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

Một cận trên của tổng điều hòa là $\log_2(n) + 1$. Cụ thể, ta có thể sửa đổi mỗi số hạng $1/k$ 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 bao gồm $\log_2(n) + 1$ phần ($1, 2 \cdot 1/2, 4 \cdot 1/4$, v.v.), và giá trị của mỗi phần không quá 1.

Lý thuyết tập hợp

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

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

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

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

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

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

- **Giao** $A \cap B$ bao gồm các phần tử có mặt trong cả A và B . Ví dụ, nếu $A = \{1, 2, 5\}$ và $B = \{2, 4\}$, thì $A \cap B = \{2\}$.
- **Hợp** $A \cup B$ bao gồm các phần tử có mặt 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** \bar{A} bao gồm các phần tử không có trong A . Cách hiểu về phần bù phụ thuộc vào **tập vũ trụ**, là tập chứa tất cả các phần tử có thể có. 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** $A \setminus B = A \cap \bar{B}$ bao gồm các phần tử có trong A nhưng không có trong B . Lưu ý rằng B có thể chứa các phần tử không có 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** của S , ký hiệu là $A \subset S$. Một tập hợp S luôn có $2^{|S|}$ tập con, bao gồm cả tập rỗng. Ví dụ, các tập con của tập hợp $\{2, 4, 7\}$ là

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

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

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

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

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

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

chứa tất cả các số nguyên chẵn.

Logic

Giá trị của một biểu thức logic là **đúng** (1) hoặc **sai** (0). Các toán tử logic quan trọng nhất là \neg (**phủ định**), \wedge (**hội**), \vee (**tuyển**), \Rightarrow (**kéo theo**) và \Leftrightarrow (**tương đương**). Bảng sau thể hiện 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 ngược lại 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 đều đúng hoặc đều sai.

Vị từ là một biểu thức có giá trị đúng hoặc sai phụ thuộc vào các tham số của nó. Vị từ thường được ký hiệu bằng các chữ cái in hoa. Ví dụ, ta có thể định nghĩa một 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 kết nối một biểu thức logic với các phần tử của một tập hợp. Các lượng từ quan trọng nhất là \forall (**với mọi**) và \exists (**tồn tại**). Ví dụ,

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

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

Sử dụng 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 là số nguyên tố, thì tồn tại các số a và b lớn hơn 1 và tích của chúng bằng x . Mệnh đề này đúng trong tập số nguyên.

Các hàm

Hàm $[x]$ làm tròn số x xuống số nguyên gần nhất, và hàm $\lceil x \rceil$ làm tròn số x lên số nguyên gần nhất. 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 $n!$ có thể được định nghĩa

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

hoặc đệ quy

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

Dãy số Fibonacci 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 dạng đóng để tính các số Fibonacci, đôi khi được gọi là **công thức Binet**:

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

Logarit

Logarit của một số x được ký hiệu là $\log_k(x)$, trong đó k là cơ số của logarit. Theo định nghĩa, $\log_k(x) = a$ 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 được 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 sử dụng trong phân tích thuật toán, vì nhiều thuật toán hiệu quả chia đôi một cái gì đó ở mỗi bước. Do đó, ta có thể ước tính hiệu quả của những thuật toán như vậy bằng cách sử dụng 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 sử dụng công thức này, có thể tính logarit với bất kỳ cơ số nào nếu có cách để tính logarit với một cơ số cố định nào đó.

Logarit tự nhiên $\ln(x)$ của một số x là logarit có cơ số là $e \approx 2.71828$. 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 Các cuộc thi và tài nguyên

IOI

Olympic Tin học Quốc tế (International Olympiad in Informatics - IOI) là một cuộc thi lập trình thường niên dành cho học sinh trung học. Mỗi quốc gia được phép gửi một đội gồm bốn học sinh tham gia cuộc thi. Thường có khoảng 300 thí sinh từ 80 quốc gia.

IOI bao gồm hai kỳ thi kéo dài năm giờ. Trong cả hai kỳ thi, thí sinh được yêu cầu giải ba bài toán thuật toán với độ khó khác nhau. Các bài toán được chia thành các phần nhỏ, mỗi phần có một điểm số được ấn định. Mặc dù thí sinh được chia thành các đội, họ thi đấu với tư cách cá nhân.

Chương trình IOI [41] quy định các chủ đề có thể xuất hiện trong các bài thi IOI. Hầu hết các chủ đề trong chương trình IOI đều được đề cập trong cuốn sách này.

Thí sinh tham dự IOI được chọn thông qua các cuộc thi quốc gia. Trước IOI, nhiều cuộc thi khu vực được tổ chức, như Olympic Tin học Baltic (BOI), Olympic Tin học Trung Âu (CEOI) và Olympic Tin học Châu Á - Thái Bình Dương (APIO).

Một số quốc gia tổ chức các cuộc thi luyện tập trực tuyến cho các thí sinh dự định tham gia IOI, như Cuộc thi Tin học Mở Croatia [11] và Olympic Tin học Hoa Kỳ [68]. Ngoài ra, một bộ sưu tập lớn các bài toán từ các cuộc thi Ba Lan có sẵn trực tuyến [60].

ICPC

Kỳ thi Lập trình Sinh viên Quốc tế (ICPC) là một cuộc thi lập trình thường niên dành cho sinh viên đại học. Mỗi đội trong cuộc thi bao gồm ba sinh viên, và khác với IOI, các sinh viên làm việc cùng nhau; chỉ có một máy tính được cung cấp cho mỗi đội.

ICPC bao gồm nhiều vòng, và cuối cùng các đội xuất sắc nhất được mời tham dự Chung kết Thế giới. Trong khi có hàng chục nghìn thí sinh tham gia cuộc thi, chỉ có một số nhỏ² suất chung kết, vì vậy ngay cả việc vào được 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. Một lời giải chỉ được chấp nhận nếu nó giải tất cả các test case 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 xem kết quả của các lần nộp bài cuối cùng.

Các chủ đề có thể xuất hiện trong ICPC không được quy định rõ ràng như trong IOI. Trong mọi trường hợp, rõ ràng là cần nhiều kiến thức hơn trong ICPC, đặc biệt là kỹ năng toán học.

Các cuộc thi trực tuyến

Cũng có nhiều cuộc thi trực tuyến mở cho tất cả mọi người. Hiện tại, trang web thi đấu tích cực nhất là Codeforces, tổ chức các cuộc thi khoảng hàng tuần. Trong Codeforces, thí sinh được chia thành hai bảng: người mới bắt đầu thi đấu ở Div2 và lập trình viên có kinh nghiệm hơn ở Div1. Các trang web thi đấu khác bao gồm AtCoder, CS Academy, HackerRank và Topcoder.

Một số công ty tổ chức các cuộc thi trực tuyến với vòng chung kết tại chỗ. Ví dụ về các cuộc thi như vậy là Facebook Hacker Cup, Google Code Jam và Yandex.Algorithm. Tất nhiên, các công ty cũng sử dụng những cuộc thi này cho việc tuyển dụng: thể hiện tốt trong một cuộc thi là cách tốt để chứng minh kỹ năng của mình.

Sách

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

- S. S. Skiena và M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim và F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]

²Số lượng suất tham dự chung kết chính xác thay đổi từng năm; năm 2017, có 133 suất chung kết.

- K. Diks và các tác giả khác: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

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

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

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

Chương 2

Độ phức tạp thời gian

Hiệu quả của các thuật toán là rất quan trọng trong lập trình thi đấu. Thông thường, việc thiết kế một thuật toán giải quyết bài toán một cách chậm chạp là dễ dàng, nhưng thách thức thực sự là phát minh ra một thuật toán nhanh. Nếu thuật toán quá chậm, nó sẽ chỉ nhận được điểm một phần hoặc không có điểm nào cả.

Độ phức tạp thời gian (time complexity) của một thuật toán ước tính thời gian mà thuật toán sẽ sử dụng cho một đầu vào nào đó. Ý tưởng là biểu diễn hiệu quả dưới dạng một hàm có tham số là kích thước của đầu vào. Bằng cách tính toán độ phức tạp thời gian, chúng ta có thể biết được thuật toán có đủ nhanh hay không mà không cần phải triển khai 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 đại diện cho một hàm nào đó. Thông thường, biến n biểu thị kích thước đầu vào. Ví dụ, nếu đầu vào là một mảng các số, n sẽ là kích thước của mảng, và nếu đầu vào là một chuỗi, n sẽ là độ dài của chuỗi.

Vòng lặp

Một lý do phổ biến khiến một thuật toán bị chậm là nó chứa nhiều vòng lặp duyệt qua đầu vào. Thuật toán càng chứa nhiều vòng lặp lồng nhau, nó 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  
}
```

Và 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

Một độ phức tạp thời gian không cho chúng ta biết số lần chính xác mà đoạn mã bên trong một vòng lặp được thực thi, mà nó chỉ cho thấy 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 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 giai đoạn

Nếu thuật toán bao gồm các giai đoạn liên tiếp, tổng độ phức tạp thời gian là độ phức tạp thời gian lớn nhất của một giai đoạn đơn lẻ. Lý do cho điều này là giai đoạn chậm nhất thường là nút thắt cổ chai của đoạn mã.

Ví dụ, đoạn mã sau bao gồm ba giai đoạn với các độ phức tạp thời gian $O(n)$, $O(n^2)$ và $O(n)$. Do đó, tổng độ phức tạp thời gian 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 số

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. Tổng độ phức tạp thời gian là tích của các giá trị này.

Ví dụ, xem xét hàm sau:

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

Lệnh gọi $f(n)$ gây ra n lần gọi hàm, và độ phức tạp thời gian của mỗi lần gọi là $O(1)$. Do đó, tổng độ phức tạp thời gian là $O(n)$.

Một ví dụ khác, xem xét hàm sau:

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

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

lệnh gọi hàm	số lần gọi
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	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 Các lớp độ phức tạp

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

- $O(1)$ Thời gian chạy của một thuật toán **thời gian 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 toán câu trả lời.
- $O(\log n)$ Một thuật toán **logarit** thường giảm một nửa kích thước đầu vào ở mỗi bước. Thời gian chạy của một thuật toán như vậy là $\log_2 n$ bằng số lần n phải được chia cho 2 để được 1.
- $O(\sqrt{n})$ Một thuật toán **căn bậc hai** chậm hơn $O(\log n)$ nhưng nhanh hơn $O(n)$. Một thuộc tính đặc biệt của căn bậc hai là $\sqrt{n} = n/\sqrt{n}$, vì vậy 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 không đổi. Đây thường là độ phức tạp thời gian tốt nhất có thể, bởi vì thường cần phải truy cập từng phần tử đầu vào ít nhất một lần trước khi báo cáo câu trả lời.
- $O(n \log n)$ Độ phức tạp thời gian này thường chỉ ra rằng thuật toán sắp xếp đầu vào, bởi 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 sử dụng một cấu trúc dữ liệu trong đó mỗi thao tác mất thời gian $O(\log n)$.
- $O(n^2)$ Một thuật toán **bậc hai** thường chứa hai vòng lặp lồng nhau. Có thể duyệt qua tất cả các cặp phần tử đầu vào trong thời gian $O(n^2)$.
- $O(n^3)$ Một thuật toán **bậc ba** thường chứa ba vòng lặp lồng nhau. Có thể duyệt qua tất cả các bộ ba phần tử đầu vào trong thời gian $O(n^3)$.
- $O(2^n)$ Độ phức tạp thời gian này thường chỉ ra rằng thuật toán lặp qua tất cả các tập hợp con của các phần tử đầu vào. Ví dụ, các tập hợp 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 chỉ ra rằng thuật toán lặp qua tất cả các 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 trên ngoại trừ $O(2^n)$ và $O(n!)$ là đa thức. Trong thực tế, hằng số k thường nhỏ, và do đó, một độ phức tạp thời gian đa thức có nghĩa gần đúng là thuật toán đó *hiệu quả*.

Hầu hết các thuật toán trong cuốn sách này là đa thức. Tuy nhiên, có nhiều bài toán quan trọng mà không có thuật toán đa thức nào được biết đến, tức là, không ai biết cách giải quyết chúng một cách hiệu quả. Các bài toán **NP-khó** là một tập hợp quan trọng các bài toán mà không có thuật toán đa thức nào được biết đến¹.

2.3 Ước tính hiệu quả

Bằng cách tính toán độ phức tạp thời gian của một thuật toán, có thể kiểm tra, trước khi triển khai thuật toán, rằng nó đủ hiệu quả cho bài toán. Điểm khởi đầu cho các ước tính là thực tế rằng một máy tính hiện đại có thể thực hiện khoảng vài trăm triệu phép toán trong một giây.

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

Ví dụ, giả sử giới hạn thời gian cho 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, vì vậy thuật toán có vẻ quá chậm để giải quyết bài toán.

Mặt khác, với kích thước đầu vào cho trước, chúng ta có thể thử *đoán* độ phức tạp thời gian cần thiết của thuật toán để giải quyết bài toán. Bảng sau đây chứa một số ước tính hữu ích giả sử giới hạn thời gian là một giây.

kích thước đầu vào	độ phức tạp thời gian yêu cầu
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ 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 ta mong đợi rằng 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 dàng hơn, bởi vì nó loại bỏ các phương pháp tiếp cận sẽ mang lại một thuật toán có độ phức tạp thời gian tồi tệ hơn.

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

2.4 Tổng con lớn nhất

Thường có nhiều thuật toán khả dĩ để giải quyết một bài toán sao cho độ phức tạp thời gian của chúng khác nhau. Phần này thảo luận về một bài toán kinh điển có một giải pháp $O(n^3)$ đơn giản. Tuy nhiên, bằng cách thiết kế một thuật toán tốt hơn, có thể giải quyết bài toán trong thời gian $O(n^2)$ và thậm chí trong thời gian $O(n)$.

Cho một mảng gồm n số, nhiệm vụ của chúng ta là tính toán **tổng 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 này thú vị khi có thể có các giá trị âm trong mảng. Ví dụ, trong mảng

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

mảng con sau đây tạo ra tổng lớn nhất là 10:

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

Chúng ta giả định rằng một mảng con rỗng được cho phép, vì vậy tổng con lớn nhất luôn ít nhất là 0.

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

Thuật toán 1

Một cách đơn giản để giải quyết bài toán là duyệt qua tất cả các mảng con có thể, tính tổng các giá trị trong mỗi mảng con và duy trì tổng lớn nhất. Đoạn mã sau đây triển khai 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` xác định chỉ số đầu và cuối của mảng con, và tổng các giá trị được tính vào biến `sum`. Biến `best` chứa tổng lớn nhất được tìm thấy trong quá trình tìm kiếm.

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

Thuật toán 2

Dễ dàng làm cho Thuật toán 1 hiệu quả hơn bằng cách loại bỏ một vòng lặp khỏi nó. Điều này có thể thực hiện được bằng cách tính tổng tại cùng thời điểm khi đầu cuối bên phải của mảng 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 quyết bài toán trong thời gian $O(n)^3$. Điều này có nghĩa là chỉ cần một vòng lặp là đủ. Ý tưởng là tính toán, cho mỗi vị trí trong mảng, tổng lớn nhất của một mảng con kết thúc tại vị trí đó. Sau đó, câu trả lời cho bài toán là giá trị lớn nhất trong số các tổng đó.

Hãy xem xét bài toán con là tìm mảng con có tổng lớn nhất kết thúc tại vị trí k . Có hai khả năng:

³Trong [8], thuật toán thời gian tuyến tính này được cho là của J. B. Kadane, và thuật toán này đôi khi được gọi là **Thuật toán của Kadane (Kadane's algorithm)**.

1. Mảng con chỉ chứa phần tử tại vị trí k .
2. Mảng con bao gồm một mảng con kết thúc tại vị trí $k - 1$, theo sau là phần tử tại vị trí k .

Trong trường hợp thứ hai, vì chúng ta muốn tìm một mảng con có tổng lớn nhất, mảng con kết thúc tại vị trí $k - 1$ cũng phải có tổng lớn nhất. Do đó, chúng ta có thể giải quyết bài toán một cách hiệu quả bằng cách tính tổng con lớn nhất cho mỗi vị trí kết thúc từ trái sang phải.

Đoạn mã sau đây triển khai thuật toán này:

```
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, vì vậy độ phức tạp thời gian là $O(n)$. Đây cũng là độ phức tạp thời gian tốt nhất có thể, bởi vì bất kỳ thuật toán nào cho bài toán này cũng phải kiểm tra tất cả các phần tử của mảng ít nhất một lần.

So sánh hiệu quả

Thật thú vị khi nghiên cứu hiệu quả của các thuật toán trong thực tế. Bảng sau đây cho thấy thời gian chạy của các thuật toán trên cho các giá trị n khác nhau trên một máy tính hiện đại.

Trong mỗi bài kiểm tra, đầu vào được tạo ngẫu nhiên. Thời gian cần thiết 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 rằng tất cả các thuật toán đều hiệu quả khi kích thước đầu vào nhỏ, nhưng các đầu vào lớn hơn làm nổi bật 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 có Thuật toán 3 mới có thể xử lý ngay cả những đầu vào lớn nhất một cách tức thì.

Chương 3

Sắp xếp

Sắp xếp là một bài toán thiết kế thuật toán cơ bản. Nhiều thuật toán hiệu quả sử dụng sắp xếp như một chương trình con, bởi vì việc xử lý dữ liệu thường dễ dàng hơn nếu các phần tử đã ở theo thứ tự được sắp xếp.

Ví dụ, bài toán "một mảng có chứa hai phần tử bằng nhau không?" rất dễ giải quyết bằng cách sử dụng sắp xếp. Nếu mảng chứa hai phần tử bằng nhau, chúng sẽ nằm cạnh nhau sau khi sắp xếp, vì vậy rất dễ để tìm thấy chúng. Ngoài ra, bài toán "phần tử nào xuất hiện thường xuyên nhất trong một mảng?" cũng có thể được giải quyết tương tự.

Có rất nhiều thuật toán để sắp xếp, và chúng cũng là những ví dụ điển hình về cách áp dụng các kỹ thuật thiết kế thuật toán khác nhau. Các thuật toán sắp xếp tổng quát hiệu quả hoạt động trong thời gian $O(n \log n)$, và nhiều thuật toán sử dụng sắp xếp như một chương trình 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
---	---	---	---	---	---	---	---

sẽ trở thành như sau sau khi sắp xếp:

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

Các thuật toán $O(n^2)$

Các thuật toán đơn giản để sắp xếp một mảng hoạt động trong thời gian $O(n^2)$. Các thuật toán như vậy thường ngắn và bao gồm hai vòng lặp lồng nhau. Một thuật toán sắp xếp thời gian $O(n^2)$ nổi tiếng là **sắp xếp nổi bọt** (bubble sort), trong đó các phần tử "nổi bọt" trong mảng theo giá trị của chúng.

Sắp xếp nổi bọt bao gồm n vòng. Ở mỗi vòng, thuật toán lặp qua các phần tử của mảng. Bất cứ khi nào tìm thấy hai phần tử liên tiếp không đúng thứ tự, thuật toán sẽ hoán đổi 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 của chúng. Do đó, 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 sắp xếp nổi bọt sẽ hoán đổi các phần tử như sau:

1	3	2	8	9	2	5	6
↖ ↗							
1	3	2	8	2	9	5	6
↖ ↗							
1	3	2	8	2	5	9	6
↖ ↗							
1	3	2	8	2	5	6	9
↖ ↗							

Nghịch thế

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

Một khái niệm hữu ích khi phân tích các thuật toán sắp xếp là **ngịch thế**: một cặp phần tử của mảng ($array[a], array[b]$) sao cho $a < b$ và $array[a] > array[b]$, 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 lượng nghịch thế cho biết cần bao nhiêu công sức để sắp xếp mảng. Một mảng được sắp xếp hoàn toàn khi không có nghịch thế nào. Mặt khác, nếu các phần tử mảng được sắp xếp theo thứ tự ngược lại, số lượng nghịch thế là lớn nhất có thể:

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

Hoán đổi một cặp phần tử liên tiếp đang ở sai thứ tự sẽ loại bỏ chính xác một nghịch thế khỏi mảng. Do đó, nếu một thuật toán sắp xếp chỉ có thể hoán đổi các phần tử liên tiếp, mỗi lần hoán đổi sẽ 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 $O(n \log n)$

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

Sắp xếp trộn sắp xếp một mảng con $\text{array}[a \dots b]$ như sau:

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

Sắp xếp trộn là một thuật toán hiệu quả vì nó chia đôi kích thước của mảng con ở mỗi bước. Quá trình đệ quy bao gồm $O(\log n)$ cấp, và việc xử lý mỗi cấp mất thời gian $O(n)$. Việc trộn các mảng con $\text{array}[a \dots k]$ và $\text{array}[k + 1 \dots b]$ có thể thực hiện trong thời gian tuyến tính, vì chúng đã được sắp xếp.

Ví dụ, hãy xem 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:

1	2	3	6
2	5	8	9

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

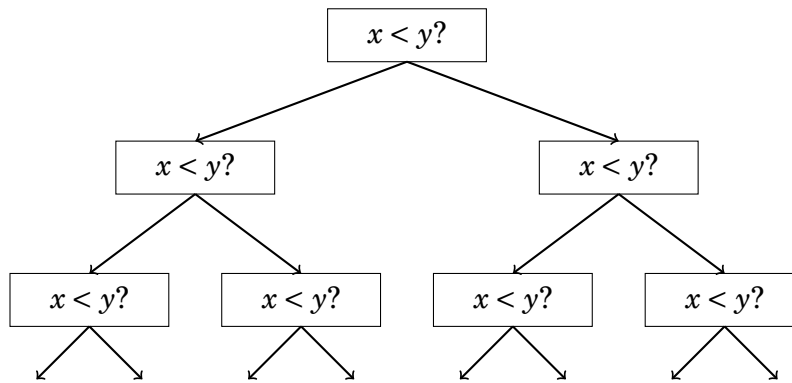
1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Cận dưới của sắp xếp

Liệu có thể sắp xếp một mảng nhanh hơn thời gian $O(n \log n)$ không? Hóa ra điều này là *không* thể khi chúng ta chỉ 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 việc sắp xếp như một quá trình trong đó mỗi phép so sánh hai phần tử cung cấp thêm thông tin về nội dung của mảng. Quá trình này tạo ra cây sau:

¹Theo [47], sắp xếp trộn được phát minh bởi J. von Neumann vào năm 1945.



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

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

Chúng ta có được một cận dưới cho tổng này bằng cách chọn $n/2$ phần tử cuối cùng và thay đổi giá trị của mỗi phần tử thành $\log_2(n/2)$. Điều này mang lại một ước tính

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

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

Sắp xếp đếm phân phối

Cận dưới $n \log n$ không áp dụng cho các thuật toán không so sánh các phần tử mảng mà sử dụng một số thông tin khác. Một ví dụ về thuật toán như vậy là **sắp xếp đếm phân phối** (counting sort) sắp xếp một mảng trong thời gian $O(n)$ với giả định rằng mọi phần tử trong mảng là một số nguyên trong khoảng $0 \dots c$ và $c = O(n)$.

Thuật toán tạo ra một mảng *đánh dấu*, mà các chỉ số của nó là các phần tử của mảng ban đầu. Thuật toán lặp qua mảng ban đầu và tính toán số lần mỗi phần tử xuất hiện trong mảng.

Ví dụ, mảng

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

tương ứng với mảng đánh dấu sau:

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

Ví dụ, giá trị tại vị trí 3 trong mảng đánh dấu là 2, bởi vì phần tử 3 xuất hiện 2 lần trong mảng ban đầu.

Việc xây dựng mảng đánh dấu mất thời gian $O(n)$. Sau đó, mảng đã sắp xếp có thể được tạo ra trong thời gian $O(n)$ vì số lần xuất hiện của mỗi phần tử có thể được lấy ra từ mảng đánh dấu. Do đó, tổng độ phức tạp thời gian của sắp xếp đếm phân phối là $O(n)$.

Sắp xếp đếm phân phối là một thuật toán rất hiệu quả nhưng nó chỉ có thể được sử dụng khi hằng số c đủ nhỏ, để các phần tử mảng có thể được sử dụng làm chỉ số trong mảng đánh dấu.

3.2 Sắp xếp trong C++

Hầu như không bao giờ là một ý tưởng tốt khi sử dụng một thuật toán sắp xếp tự viết trong một cuộc thi, bởi vì có những cài đặt tốt có sẵn trong các ngôn ngữ lập trình. Ví dụ, thư viện chuẩn C++ chứa hàm `sort` có thể dễ dàng được sử dụng để sắp xếp mảng và các cấu trúc dữ liệu khác.

Có nhiều lợi ích trong việc sử dụng một hàm thư viện. Thứ nhất, nó tiết kiệm thời gian vì không cần phải cài đặt hàm. Thứ hai, việc cài đặt của thư viện chắc chắn là đúng và hiệu quả: không có khả năng một hàm sắp xếp tự viết sẽ tốt hơn.

Trong phần này, chúng ta sẽ xem cách sử dụng 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 theo thứ tự ngược lại như sau:

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

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

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

Đoạn mã sau sắp xếp chuỗi s:

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

Sắp xếp một chuỗi có nghĩa là các ký tự của chuỗi được sắp xếp. Ví dụ, chuỗi "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 sử dụng bất cứ khi nào cần phải biết thứ tự của hai phần tử.

Hầu hết các kiểu dữ liệu C++ đều có một toán tử so sánh tích hợp, và các phần tử của các kiểu đó có thể được sắp xếp tự động. Ví dụ, các số được sắp xếp theo giá trị của chúng và các chuỗi được sắp xếp theo thứ tự chữ cái.

Cặp (pair) được sắp xếp chủ yếu theo phần tử đầu tiên của chúng (first). Tuy nhiên, nếu phần tử đầu tiên của hai cặp bằng nhau, chúng được sắp xếp theo phần tử thứ hai của chúng (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, bộ (tuple) được sắp xếp chủ yếu theo phần tử đầu tiên, thứ yếu 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 bộ là (1, 5, 3), (2, 1, 3) và (2, 1, 4).

Struct do người dùng định nghĩa

Struct do người dùng định nghĩa không có toán tử so sánh tự động. Toán tử này 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 hiện tại nhỏ hơn tham số, và `false` trong trường hợp ngược lại.

Ví dụ, struct `P` sau đây chứa tọa độ `x` và `y` của một điểm. Toán tử so sánh được định nghĩa sao cho các điểm được sắp xếp chủ yếu theo tọa độ `x` và thứ yếu 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 cung cấp một **hàm so sánh** bên ngoài cho hàm `sort` dưới dạng một hàm callback. Ví dụ, hàm so sánh `comp` sau đây sắp xếp các chuỗi chủ yếu theo độ dài và thứ yếu 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 chuỗi có thể được sắp xếp như sau:

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

²Lưu ý rằng trong một số trình biên dịch cũ hơn, hàm `make_tuple` phải được sử dụng để tạo một tuple thay vì dấu ngoặc nhọn (ví dụ, `make_tuple(2,1,4)` thay vì `{2,1,4}`).

3.3 Tìm kiếm nhị phân

Một phương pháp chung để tìm kiếm một phần tử trong một mảng là sử dụng một vòng lặp `for` lặp qua các phần tử của mảng. Ví dụ, đoạn mã sau tìm kiếm một phần tử x trong một mảng:

```
for (int i = 0; i < n; i++) {  
    if (array[i] == x) {  
        // tìm thấy x tại chỉ số i  
    }  
}
```

Độ phức tạp thời gian của phương pháp này là $O(n)$, bởi vì trong trường hợp xấu nhất, cần phải kiểm tra tất cả các phần tử của mảng. Nếu thứ tự của các phần tử là tùy ý, đây cũng là phương pháp tốt nhất có thể, bởi vì không có thông tin bổ sung nào cho biết chúng ta nên tìm kiếm 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, bởi vì thứ tự của các phần tử trong mảng sẽ hướng dẫn việc tìm kiếm. Thuật toán **tìm kiếm nhị phân** sau đây tìm kiếm một phần tử trong một mảng đã sắp xếp một cách hiệu quả trong thời gian $O(\log n)$.

Phương pháp 1

Cách thông thường để cài đặt tìm kiếm nhị phân giống như tìm một từ trong từ điển. Việc tìm kiếm duy trì một vùng hoạt động trong mảng, ban đầu chứa tất cả các phần tử của mảng. Sau đó, một số bước được thực hiện, mỗi bước giảm một nửa kích thước của vùng.

Ở mỗi bước, việc tìm kiếm kiểm tra phần tử ở giữa của vùng hoạt động. Nếu phần tử ở giữa là phần tử mục tiêu, việc tìm kiếm kết thúc. Ngược lại, việc tìm kiếm tiếp tục đệ quy sang nửa bên trái hoặc bên phải của vùng, tùy thuộc vào giá trị của phần tử ở 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) {  
        // tìm thấy x tại chỉ số k  
    }  
    if (array[k] > x) b = k-1;  
    else a = k+1;  
}
```

Trong cài đặt này, vùng hoạt động là $a \dots b$, và ban đầu vùng là $0 \dots n-1$. Thuật toán giảm một nửa kích thước của vùng ở mỗi bước, vì vậy 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 lặp qua các phần tử của mảng. Ý tưởng là thực hiện các bước nhảy và giảm tốc độ khi chúng ta đến gần phần tử mục tiêu.

Việc 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 giảm một nửa: đầu tiên là $n/4$, sau đó là $n/8$, $n/16$, 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 chúng 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) {
    // tìm thay x tại chỉ số k
}
```

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

Các hàm C++

Thư viện chuẩn C++ chứa các hàm sau đây dựa trên tìm kiếm nhị phân và hoạt động trong thời gian logarit:

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

Các hàm này giả định rằng mảng đã được sắp xếp. Nếu không có phần tử như vậy, con trỏ trỏ đến phần tử sau phần tử mảng cuối cùng. Ví dụ, đoạn mã sau tìm hiểu xem một mảng có chứa một phần tử có giá trị x hay không:

```
auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // tìm thay x tại chỉ số k
}
```

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

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

Sử dụng `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 để tìm vị trí mà giá trị của một hàm thay đổi. Giả sử chúng ta muốn tìm giá trị nhỏ nhất k là một nghiệm hợp lệ cho một bài toán. Chúng ta được cho một hàm $ok(x)$ trả về `true` nếu x là một nghiệm hợp lệ và `false`

trong trường hợp ngược lại. Ngoài ra, chúng ta biết rằng $ok(x)$ là false khi $x < k$ và true khi $x \geq k$. Tình huống trông như sau:

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	false	false	...	false	true	true	...

Bây giờ, giá trị của k có thể được tìm thấy 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;
```

Việc tìm kiếm tìm giá trị lớn nhất của x mà $ok(x)$ là false. Do đó, giá trị tiếp theo $k = x + 1$ là giá trị nhỏ nhất có thể mà $ok(k)$ là true. Độ dài bước nhảy ban đầu z phải đủ lớn, ví dụ một giá trị nào đó mà chúng ta biết trước rằng $ok(z)$ là true.

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

Tìm giá trị lớn nhất

Tìm kiếm nhị phân cũng có thể được sử dụng để tìm giá trị lớn nhất cho một hàm ban đầu tăng và sau đó giảm. Nhiệm vụ của chúng 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 sử dụng tìm kiếm nhị phân để tìm giá trị lớn nhất của x mà $f(x) < f(x+1)$. Điều này ngụ ý rằng $k = x + 1$ bởi vì $f(x+1) > f(x+2)$. Đoạn mã sau cài đặt việc 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 không giống như trong tìm kiếm nhị phân thông thường, ở đây không được phép các giá trị liên tiếp của hàm bằng nhau. Trong trường hợp đó sẽ không thể biết cách tiếp tục tìm kiếm.

Chương 4

Cấu trúc dữ liệu

Cấu trúc dữ liệu là cách để 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, bởi vì mỗi cấu trúc dữ liệu có những ưu điểm và nhược điểm riêng. Câu hỏi quan trọng là: những thao tác nào hiệu quả trong cấu trúc dữ liệu đã chọn?

Chương này giới thiệu những cấu trúc dữ liệu quan trọng nhất trong thư viện chuẩn C++. Sử dụng thư viện chuẩn bất cứ khi nào có thể là một ý tưởng tốt, bởi vì nó sẽ tiết kiệm rất nhiều thời gian. Sau này trong cuốn sách chúng ta sẽ học về các cấu trúc dữ liệu phức tạp hơn mà không có sẵn trong thư viện chuẩn.

4.1 Mảng động

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

Đoạn code 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, các phần tử có thể được truy cập 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 code sau duyệt qua vector và in tất cả các phần tử trong đó:

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

Một cách ngắn gọn hơn để duyệt qua vector 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 code sau tạo một vector với năm phần tử:

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

Một cách khác để tạo vector là chỉ định số lượng phần tử và giá trị ban đầu cho mỗi phần tử:

```
// kích thước 10, giá trị ban đầu 0  
vector<int> v(10);
```

```
// kích thước 10, giá trị ban đầu 5  
vector<int> v(10, 5);
```

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

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

Đoạn code sau trình bày một số thao tác với chuỗi:

```
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

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

Thư viện chuẩn C++ chứa hai cách cài đặt tập hợp: Cấu trúc `set` được dựa trên cây nhị phân cân bằng và các thao tác của nó hoạt động trong thời gian $O(\log n)$. Cấu trúc `unordered_set` sử dụng bảng băm, và các thao tác của nó hoạt động trong thời gian trung

binh $O(1)$.

Việc chọn cài đặt tập hợp nào để sử dụng 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 sẵn trong unordered_set. Mặt khác, unordered_set có thể hiệu quả hơn.

Đoạn code sau tạo một tập hợp chứa các số nguyên, và cho thấy 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 sử dụng gần như một vector, nhưng không thể truy cập các phần tử bằng ký hiệu []. Đoạn code sau tạo một tập hợp, in số lượng phần tử trong nó, và sau đó duyệt qua tất cả các 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 các phần tử của chúng đều *khác nhau*. Do đó, 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 nó đã có ở đó. Đoạn code 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 như set và unordered_set nhưng chúng có thể chứa nhiều phiên bản của một phần tử. Ví dụ, trong đoạn code sau, cả ba phiên bản của số 5 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 các phiên bản của một phần tử khỏi multiset:

```
s.erase(5);
```

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

Thường thì, chỉ một phiên bản cần được xóa, điều này có thể được thực hiện như sau:

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

4.3 Cấu trúc map

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

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

Đoạn code sau tạo một map trong đó các khóa là các chuỗi và các giá trị là các 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 nó, khóa sẽ tự động được thêm vào map với một giá trị mặc định. Ví dụ, trong đoạn code sau, khóa "aybabbtu" với giá trị 0 được thêm vào map.

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

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

```
if (m.count("aybabbtu")) {  
    // khóa tồn tại  
}
```

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

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

4.4 Iterator và khoảng

Nhiều hàm trong thư viện chuẩn C++ hoạt động với iterator. **Iterator** là một biến trỏ đến một phần tử trong cấu trúc dữ liệu.

Các iterator thường được sử dụng `begin` và `end` định nghĩa một khoảng chứa tất cả các phần tử trong một cấu trúc dữ liệu. Iterator `begin` trở đến phần tử đầu tiên trong cấu trúc dữ liệu, và iterator `end` trở đến vị trí *sau* phần tử cuối cùng. Tình huống như sau:

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

Lưu ý sự bất đối xứng trong các iterator: `s.begin()` trở đến một phần tử trong cấu trúc dữ liệu, trong khi `s.end()` trở ra ngoài cấu trúc dữ liệu. Do đó, khoảng được định nghĩa bởi các iterator là *nửa mở*.

Làm việc với khoảng

Iterator được sử dụng trong các hàm thư viện chuẩn C++ nhận một khoảng các phần tử trong một cấu trúc dữ liệu. Thường thì, chúng ta muốn xử lý tất cả các phần tử trong một cấu trúc dữ liệu, nên các iterator `begin` và `end` được truyền cho hàm.

Ví dụ, đoạn code 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 sử dụng với một mảng thông thường. Trong trường hợp này, các hàm nhận các con trỏ đến mảng thay vì iterator:

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

Iterator trong tập hợp

Iterator thường được sử dụng để truy cập các phần tử của một tập hợp. Đoạn code sau tạo một iterator `it` trở đến phần tử nhỏ nhất trong một tập hợp:

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

Một cách ngắn gọn hơn để viết code như sau:

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

Phần tử mà iterator trở đến có thể được truy cập bằng ký hiệu `*`. Ví dụ, đoạn code sau in phần tử đầu tiên trong tập hợp:

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

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

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

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

Đoạn code sau in phần tử lớn nhất trong tập hợp:

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

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

```
auto it = s.find(x);
if (it == s.end()) {
    // không tìm thấy x
}
```

Hàm `lower_bound(x)` trả về một iterator đến phần tử nhỏ nhất trong tập hợp có giá trị *ít nhất là* x , và hàm `upper_bound(x)` trả về một iterator đến phần tử nhỏ nhất trong tập hợp 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 hỗ trợ bởi cấu trúc `unordered_set` vì nó không duy trì thứ tự của các phần tử.

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

```
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 code giả định rằng tập hợp không rỗng, và xử lý tất cả các trường hợp có thể bằng cách sử dụng một iterator `it`. Đầu tiên, iterator trỏ đến phần tử nhỏ nhất có giá trị ít nhất là x . Nếu `it` bằng `begin`, phần tử tương ứng là gần nhất với x . Nếu `it` bằng `end`, phần tử lớn nhất trong tập hợp là gần nhất với x . Nếu không phải các trường hợp trên, phần tử gần nhất với x là phần tử tương ứng với `it` hoặc phần tử trước đó.

4.5 Các cấu trúc khác

Bitset

Bitset là một mảng mà mỗi giá trị của nó là 0 hoặc 1. Ví dụ, đoạn code 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 sử dụng bitset là chúng yêu cầu ít bộ nhớ hơn các mảng thông thường, bởi vì mỗi phần tử trong một bitset chỉ sử dụng một bit bộ nhớ. Ví dụ, nếu n bit được lưu trữ trong một mảng `int`, $32n$ bit bộ nhớ sẽ được sử dụng, nhưng một bitset tương ứng chỉ cần n bit bộ nhớ. Ngoài ra, các giá trị của một bitset có thể được thao tác hiệu quả bằng cách sử dụng các toán tử bit, điều này làm cho việc tối ưu hóa các thuật toán sử dụng bitset trở nên khả thi.

Đoạn code sau cho thấy một cách khác để tạo bitset trên:

```
bitset<10> s(string("0010011010")); // tu phải sang trái
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Hàm `count` trả về số lượng số một trong bitset:

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

Đoạn code sau cho thấy ví dụ về việc sử dụng các phép toán 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

Deque là một mảng động có 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 bao gồm các hàm `push_front` và `pop_front` không có sẵn trong `vector`.

Một deque có thể được sử dụng 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 một deque phức tạp hơn so với `vector`, và vì lý do này, deque chậm hơn `vector`. Tuy nhiên, cả việc thêm và xóa phần tử đều mất thời gian trung bình $O(1)$ ở cả hai đầu.

Stack

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 từ đỉnh. Chỉ có thể truy cập phần tử ở đỉnh của stack.

Đoạn code sau cho thấy cách sử dụng stack:

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

Queue

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 hàng đợi.

Đoạn code sau cho thấy cách sử dụng hàng đợi:

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

Priority queue

Priority queue duy trì một tập các phần tử. Các thao tác được hỗ trợ là chèn và, tùy thuộc vào loại hàng đợi, truy xuất và xóa hoặc phần tử nhỏ nhất hoặc lớn nhất. Chèn và xóa mất thời gian $O(\log n)$, và truy xuất mất thời gian $O(1)$.

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

Mặc định, các phần tử trong C++ priority queue được sắp xếp theo thứ tự giảm dần, và có thể tìm và xóa phần tử lớn nhất trong hàng đợi. Đoạn code 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 chúng ta muốn tạo một priority queue hỗ trợ việc 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 dựa trên policy

Trình biên dịch g++ cũng hỗ trợ một số cấu trúc dữ liệu không thuộc thư viện chuẩn C++. Các cấu trúc như vậy được gọi là cấu trúc dữ liệu *dựa trên policy*. Để sử dụng các cấu trúc này, các dòng sau phải được thêm vào code:

```
#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 như `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 tập hợp như sau:

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

Điểm đặc biệt của tập hợp này là ta có thể 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 đến phần tử ở một 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 tập hợp, ta nhận được vị trí mà phần tử sẽ có trong tập hợp:

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

Cả hai hàm đều hoạt động trong thời gian logarithm.

4.6 So sánh với sắp xếp

Thường có thể giải quyết một bài toán bằng cách sử dụng cấu trúc dữ liệu hoặc sắp xếp. Đôi khi có những khác biệt đáng kể về hiệu quả thực tế của các phương pháp này, điều mà có thể bị che giấu trong độ phức tạp thời gian của chúng.

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

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

câu trả lời là 3 vì các số 2, 5 và 9 thuộc cả hai danh sách.

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

Thuật toán 1

Chúng ta xây dựng một tập hợp các phần tử xuất hiện trong A , và sau đó, chúng ta lặp qua các phần tử của B và kiểm tra xem mỗi phần tử có thuộc A hay không. Điều này hiệu quả vì các phần tử của A nằm trong một tập hợp. Sử dụng cấu trúc `set`, độ phức tạp thời gian của thuật toán là $O(n \log n)$.

Thuật toán 2

Không cần thiết phải duy trì một tập hợp có thứ tự, vì vậy thay vì cấu trúc `set` chúng ta cũng có thể sử dụng cấu trúc `unordered_set`. Đây là một cách dễ dàng để làm cho thuật toán hiệu quả hơn, bởi vì chúng ta chỉ cần thay đổi cấu trúc dữ liệu cơ bản. Độ phức tạp thời gian của thuật toán mới là $O(n)$.

Thuật toán 3

Thay vì cấu trúc dữ liệu, chúng ta có thể sử dụng sắp xếp. Đầu tiên, chúng ta sắp xếp cả hai danh sách A và B . Sau đó, chúng ta lặp qua cả hai danh sách cùng một lúc và tìm các phần tử chung. Độ phức tạp thời gian của việc sắp xếp là $O(n \log n)$, và phần còn lại của thuật toán hoạt động trong thời gian $O(n)$, vì vậy tổng độ phức tạp thời gian là $O(n \log n)$.

So sánh hiệu quả

Bảng sau đây 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 hệt nhau ngoại trừ việc chúng sử dụng các cấu trúc tập hợp khác nhau. Trong bài toán này, sự lựa chọn này có ảnh hưởng quan trọng đến thời gian chạy, bởi vì Thuật toán 2 nhanh hơn 4–5 lần so với Thuật toán 1.

Tuy nhiên, thuật toán hiệu quả nhất là Thuật toán 3 sử dụng sắp xếp. Nó chỉ sử dụng 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 mặc dù vậy, Thuật toán 3 nhanh hơn mười lần. Điều này có thể được giải thích bởi thực tế là sắp xếp là một thủ tục đơn giản và nó chỉ được thực hiện một lần ở đầu Thuật toán 3, và phần còn lại của thuật toán hoạt động 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 diện

Tìm kiếm toàn diện là một phương pháp tổng quát có thể được sử dụng để giải quyết hầu hết mọi bài toán thuật toán. Ý tưởng là tạo ra tất cả các giải pháp có thể cho bài toán bằng cách duyệt trâu, và sau đó chọn giải pháp tốt nhất hoặc đếm số lượng giải pháp, tùy thuộc vào bài toán.

Tìm kiếm toàn diện là một kỹ thuật tốt nếu có đủ thời gian để duyệt qua tất cả các giải pháp, bởi vì việc tìm kiếm thường dễ cài đặt và nó luôn cho ra câu trả lời đúng. Nếu tìm kiếm toàn diện quá chậm, các kỹ thuật khác, chẳng hạn như thuật toán tham lam hoặc quy hoạch động, có thể cần thiết.

5.1 Tạo tập con

Đầu tiên, chúng ta xem xét bài toán tạo tất cả các tập con của một tập hợp gồm 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 tạo tập con: chúng ta có thể thực hiện tìm kiếm đệ quy hoặc khai thác biểu diễn bit của các số nguyên.

Phương pháp 1

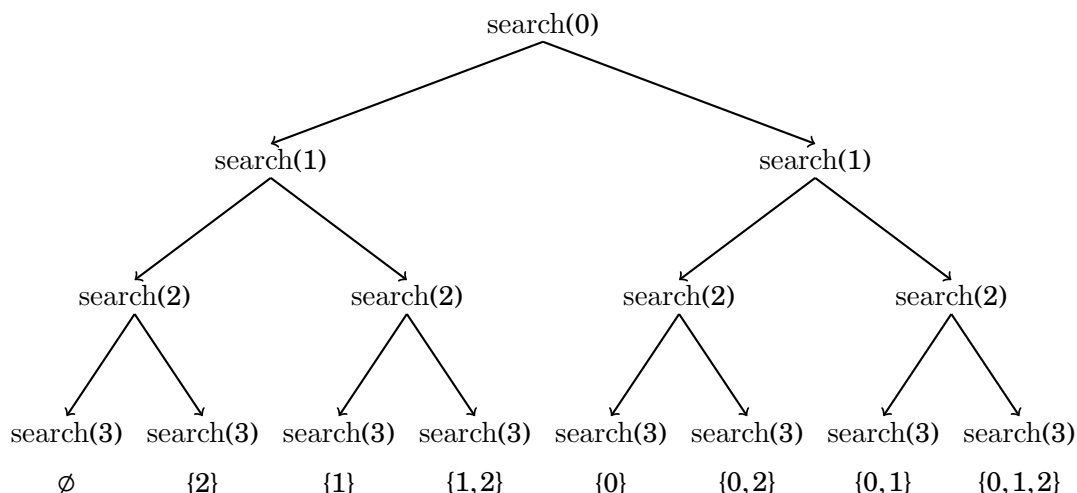
Một cách thanh lịch để duyệt qua tất cả các tập con của một tập hợp là sử dụng đệ quy. Hàm `search` sau đây tạo ra các tập con của tập hợp $\{0, 1, \dots, n-1\}$. Hàm duy trì một vector `subset` sẽ chứa các phần tử của mỗi tập con. Việc tìm kiếm bắt đầu khi hàm được gọi với tham số 0.

```
void search(int k) {
    if (k == n) {
        // xu ly tap con
    } 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 bao gồm phần tử k trong tập con hay không, và trong cả hai trường hợp, sau đó gọi chính nó với tham số $k+1$. Tuy

nhiên, nếu $k = n$, hàm nhận thấy rằng tất cả các phần tử đã được xử lý và một tập con đã được tạo ra.

Cây sau đây minh họa các lệnh gọi hàm khi $n = 3$. Chúng ta luôn có thể chọn nhánh bên trái (k không được bao gồm trong tập con) hoặc nhánh bên phải (k được bao gồm trong tập con).



Phương pháp 2

Một cách khác để tạo tập con là dựa trên biểu diễn bit của các số nguyên. Mỗi tập con của một tập hợp gồm n phần tử có thể được biểu diễn dưới dạng một chuỗi n bit, tương ứng với một số nguyên từ $0 \dots 2^n - 1$. Các bit một trong chuỗi bit cho biết phần tử nào được bao gồm trong tập con.

Quy ước thông thường là bit cuối cùng tương ứng với phần tử 0, bit thứ hai cuối cùng tương ứng với phần tử 1, vân vân. 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 hợp gồm n phần tử

```
for (int b = 0; b < (1<<n); b++) {
    // xử lý tập con
}
```

Đoạn mã sau cho thấy cách chúng ta có thể tìm các phần tử của một tập con tương ứng với một chuỗi bit. Khi xử lý mỗi tập con, đoạn mã xây dựng một vector chứa các phần tử trong tập con.

```
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 Tạo hoán vị

Tiếp theo, chúng ta xem xét bài toán tạo tất cả các hoán vị của một tập hợp gồm 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 cách tiếp cận: chúng ta có thể sử dụng đệ quy hoặc duyệt qua các hoán vị một cách lặp đi lặp lại.

Phương pháp 1

Giống như tập con, hoán vị có thể được tạo ra bằng cách sử dụng đệ quy. Hàm `search` sau đây duyệt qua các hoán vị của tập hợp $\{0, 1, \dots, n-1\}$. Hàm xây dựng một vector `permutation` chứa hoán vị, và việc tìm kiếm bắt đầu khi hàm được gọi mà không có tham số.

```
void search() {
    if (permutation.size() == n) {
        // xử lý hoán vị
    } 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 lệnh gọi hàm thêm một phần tử mới vào `permutation`. Mảng `chosen` cho biết phần tử nào đã được bao gồm trong hoán vị. Nếu kích thước của `permutation` bằng kích thước của tập hợp, một hoán vị đã được tạo ra.

Phương pháp 2

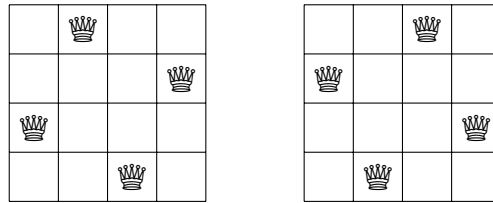
Một phương pháp khác để tạo hoán vị là bắt đầu với hoán vị $\{0, 1, \dots, n-1\}$ và lặp đi lặp lại sử dụng một hàm xây dựng hoán vị tiếp theo theo thứ tự tăng dần. Thư viện chuẩn C++ chứa hàm `next_permutation` có thể được sử dụng cho việc này:

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // xử lý hoán vị
} while (next_permutation(permutation.begin(), permutation.end()));
```

5.3 Quay lui

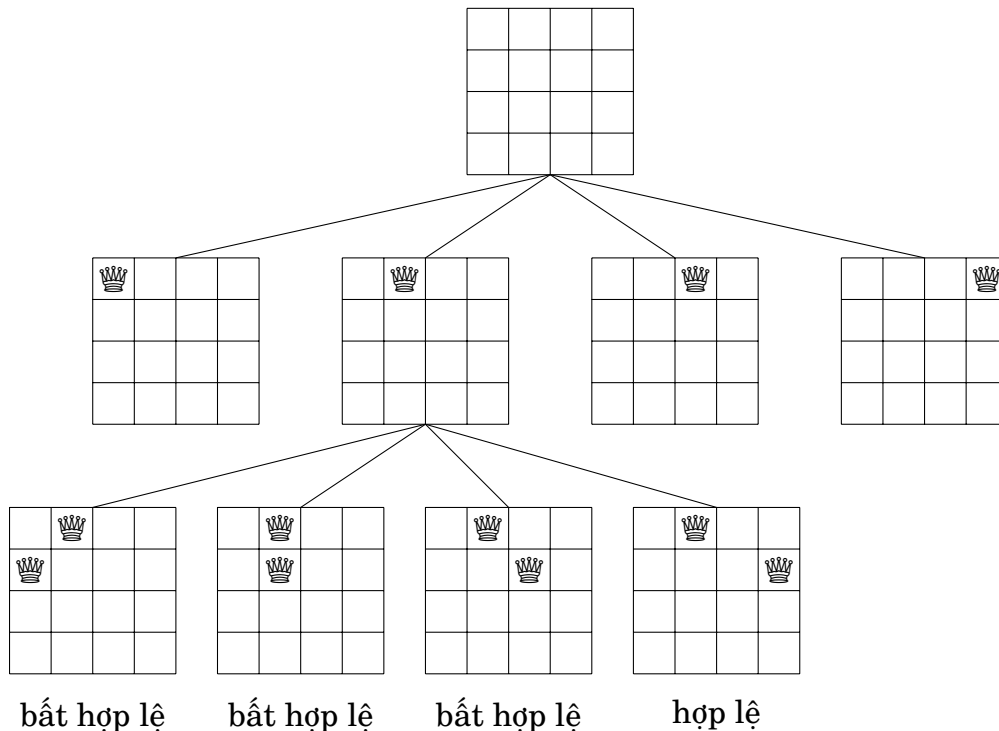
Một thuật toán **quay lui** bắt đầu với một giải pháp rỗng và mở rộng giải pháp từng bước. Việc tìm kiếm đệ quy duyệt qua tất cả các cách khác nhau để xây dựng một giải pháp.

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



Bài toán có thể được giải quyết bằng cách sử dụng quay lui bằng cách đặt các quân hậu lên bàn cờ theo từng hàng. Chính xác hơn, chính xác một quân hậu sẽ được đặt trên mỗi hàng sao cho không có quân hậu nào tấn công bất kỳ quân hậu nào được đặt trước đó. Một giải pháp đã được tìm thấy khi tất cả n quân hậu đã được đặt trên bàn cờ.

Ví dụ, khi $n = 4$, một số giải pháp một phần được tạo ra bởi thuật toán quay lui như sau:



Ở cấp dưới cùng, ba cấu hình đầu tiên là bất hợp lệ, vì các quân hậu tấn công nhau. Tuy nhiên, cấu hình thứ tư là hợp lệ và nó có thể được mở rộng thành một giải pháp hoàn chỉnh bằng cách đặt thêm hai quân hậu nữa 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;
    }
}
```

Việc tìm kiếm bắt đầu bằng cách gọi `search(0)`. Kích thước của bàn cờ là $n \times n$, và đoạn mã tính số lượng giải pháp vào `count`.

Đoạn mã giả định 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 và sau đó gọi chính nó với tham số $y + 1$. Sau đó, nếu $y = n$, một giải pháp đã được tìm thấy và biến `count` được tăng lên một.

Mảng `column` theo dõi các cột chứa một quân hậu, và các mảng `diag1` và `diag2` theo dõi các đường chéo. Không được phép thêm một quân hậu khác vào một cột hoặc đường chéo đã chứa một 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

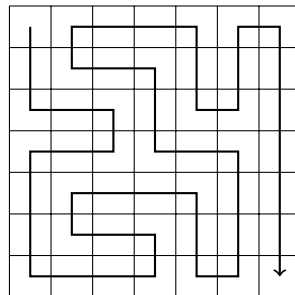
diag2

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

5.4 Cắt tỉa tìm kiếm

Chúng ta thường có thể tối ưu hóa quay lui bằng cách cắt tỉa cây tìm kiếm. Ý tưởng là thêm "trí thông minh" vào thuật toán để nó sẽ nhận ra càng sớm càng tốt nếu một giải pháp một phần không thể được mở rộng thành một giải pháp hoàn chỉnh. Những tối ưu hóa như vậy có thể có một tác động to lớn đến hiệu quả của việc tìm kiếm.

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



Chúng ta tập trung vào trường hợp 7×7 , vì mức độ khó của nó phù hợp với nhu cầu của chúng ta. Chúng ta bắt đầu với một thuật toán quay lui đơn giản, và sau đó tối ưu hóa nó từng bước bằng cách sử dụng các quan sát về cách tìm kiếm có thể được cắt tỉa. Sau mỗi lần tối ưu hóa, chúng ta đo thời gian chạy của thuật toán và số lượng lệnh gọi đệ quy, để chúng ta thấy rõ tác dụng của mỗi lần tối ưu hóa đối với hiệu quả của việc tìm kiếm.

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

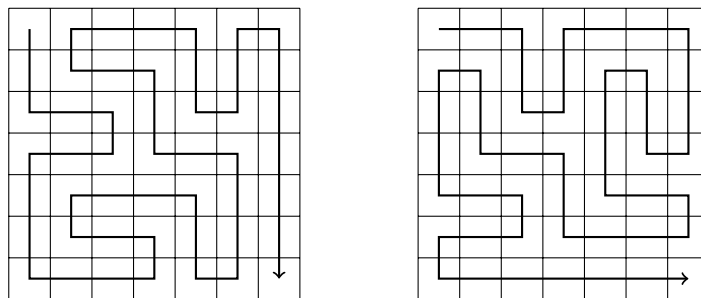
Thuật toán cơ bản

Phiên bản đầu tiên của thuật toán không chứa bất kỳ tối ưu hóa nào. Chúng ta chỉ đơn giản sử dụng quay lui để tạo ra tất cả các đường đi có thể từ góc trên bên trái đến góc dưới bên phải và đếm số lượng các đường đi đó.

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

Tối ưu hóa 1

Trong bất kỳ giải pháp nào, chúng ta trước tiên di chuyển một bước xuống hoặc sang phải. Luôn có hai đường đi đối xứng qua đường chéo của lưới sau bước đầu tiên. Ví dụ, các đường đi sau đây là đối xứng:

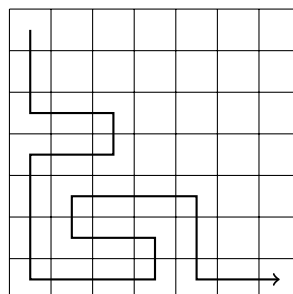


Do đó, chúng ta có thể quyết định rằng chúng ta luôn luôn di chuyển trước một bước xuống (hoặc sang phải), và cuối cùng nhân số lượng giải pháp với hai.

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

Tối ưu hóa 2

Nếu đường đi đến ô vuông dưới cùng bên phải trước khi nó đã thăm tất cả các ô vuông khác của lưới, rõ ràng là sẽ không thể hoàn thành giải pháp. Một ví dụ về điều này là đường đi sau:

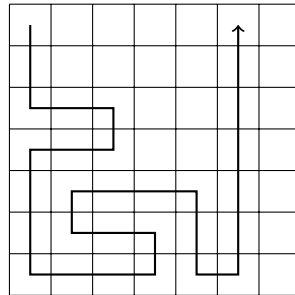


Sử dụng quan sát này, chúng ta có thể kết thúc tìm kiếm ngay lập tức nếu chúng ta đến ô vuông dưới cùng bên phải quá sớm.

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

Tối ưu hóa 3

Nếu đường đi chạm vào một bức tường và có thể rẽ trái hoặc phải, lưới sẽ chia thành hai phần chứa các ô vuông chưa được thăm. Ví dụ, trong tình huống sau, đường đi có thể rẽ trái hoặc phải:

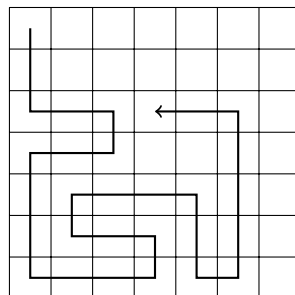


Trong trường hợp này, chúng ta không thể thăm tất cả các ô vuông nữa, vì vậy chúng ta có thể kết thúc tìm kiếm. Tối ưu hóa này rất hữu ích:

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

Tối ưu hóa 4

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



Rõ ràng là chúng ta không thể thăm tất cả các ô vuông nữa, vì vậy chúng ta có thể kết thúc tìm kiếm. Sau khi tối ưu hóa này, việc tìm kiếm trở nên rất hiệu quả:

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

Bây giờ là thời điểm tốt để ngừng tối ưu hóa thuật toán và xem chúng ta đã đạt được gì. Thời gian chạy của thuật toán ban đầu là 483 giây, và bây giờ sau các tối ưu hóa, thời gian chạy chỉ còn 0.6 giây. Do đó, thuật toán đã trở nên nhanh hơn gần 1000 lần sau các tối ưu hóa.

Đây là một hiện tượng thông thường trong quay lui, vì cây tìm kiếm thường lớn và ngay cả những quan sát đơn giản cũng có thể hiệu quả cắt tỉa tìm kiếm. Đặc biệt hữu ích là các tối ưu hóa xảy ra trong các bước đầu tiên của thuật toán, tức là, ở đỉnh của cây tìm kiếm.

5.5 Gặp nhau ở giữa

Gặp nhau ở giữa là một kỹ thuật trong đó không gian tìm kiếm được chia thành hai phần có kích thước gần bằng nhau. Một tìm kiếm riêng biệt được thực hiện cho cả hai phần, và cuối cùng kết quả của các tìm kiếm được kết hợp lại.

Kỹ thuật này có thể được sử dụng nếu có một cách hiệu quả để kết hợp kết quả của các tìm kiếm. Trong tình huống như vậy, hai tìm kiếm có thể yêu cầu ít thời gian hơn một tìm kiếm lớn. Thông thường, chúng ta có thể biến một yếu tố 2^n thành một yếu tố $2^{n/2}$ bằng cách sử dụng kỹ thuật gặp nhau ở giữa.

Ví dụ, hãy xem xét một bài toán trong đó chúng ta được cho một danh sách n số và một số x , và chúng ta muốn tìm hiểu xem có thể chọn một số số từ danh sách sao cho tổng của chúng là x . Ví dụ, cho danh sách $[2, 4, 5, 9]$ và $x = 15$, chúng ta có thể chọn các số $[2, 4, 9]$ để có $2 + 4 + 9 = 15$. Tuy nhiên, nếu $x = 10$ cho cùng một danh sách, không thể tạo thành tổng.

Một thuật toán đơn giản cho bài toán là duyệt qua tất cả các tập con của các phần tử và kiểm tra xem tổng của bất kỳ tập con nào có phải là x không. Thời gian chạy của một thuật toán như vậy là $O(2^n)$, vì có 2^n tập con. Tuy nhiên, bằng cách sử dụng kỹ thuật gặp nhau ở giữa, chúng ta có thể đạt được một 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 chứa khoảng một nửa số lượng số. Tìm kiếm đầu tiên tạo ra tất cả các tập con của A và lưu trữ tổng của chúng vào một danh sách S_A . Tương ứng, tìm kiếm thứ hai tạo ra một 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 . Điều này có thể xảy ra chính xác khi có một cách để tạo thành tổng x bằng cách sử dụng các số của danh sách ban đầu.

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

Chúng ta có thể cài đặt thuật toán sao cho độ phức tạp thời gian của nó là $O(2^{n/2})$. Đầu tiên, chúng ta tạo các danh sách *đã sắp xếp* S_A và S_B , có thể được thực hiện trong thời gian $O(2^{n/2})$ bằng cách sử dụng một kỹ thuật giống như hợp nhất. Sau đó, vì các danh sách đã được sắp xếp, chúng 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 giới thiệu vào năm 1974 bởi E. Horowitz và S. Sahni [39].

Chương 6

Thuật toán tham lam

Một **thuật toán tham lam** xây dựng một giải pháp cho bài toán bằng cách luôn đưa ra một lựa chọn có vẻ tốt nhất tại thời điểm đó. Một thuật toán tham lam không bao giờ rút lại các lựa chọn của mình, mà trực tiếp xây dựng giải pháp cuối cùng. Vì lý do này, các thuật toán tham lam thường rất hiệu quả.

Khó khăn trong việc thiết kế các thuật toán tham lam là tìm ra một chiến lược tham lam luôn tạo ra một giải pháp 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 để chứng minh rằng một thuật toán tham lam hoạt động.

6.1 Bài toán đổi tiền

Ví dụ đầu tiên, chúng ta xem xét một bài toán trong đó chúng ta được cho một tập hợp các đồng xu và nhiệm vụ của chúng ta là tạo thành một tổng tiền n bằng cách sử dụng các đồng xu. Các giá trị của các đồng xu là $\text{coins} = \{c_1, c_2, \dots, c_k\}$, và mỗi đồng xu có thể được sử dụng bao nhiêu lần tùy ý. Số lượng đồng xu tối thiểu cần thiết là bao nhiêu?

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

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

và $n = 520$, chúng ta cần ít nhất bốn đồng xu. Giải pháp 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 tổng tiền yêu cầu được tạo thành. Thuật toán này hoạt động trong trường hợp ví dụ, vì chúng ta trước tiên chọn hai đồng 200 cent, sau đó một đồng 100 cent và cuối cùng là một đồng 20 cent. Nhưng thuật toán này có luôn hoạt động không?

Hóa ra nếu các đồng xu là các đồng euro, thuật toán tham lam *luôn* hoạt động, tức là, nó luôn tạo ra một giải pháp với số lượng đồng xu ít nhất có thể. Tính đúng đắn của thuật toán có thể được chứng minh như sau:

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

Tương tự, các đồng xu 2 và 20 xuất hiện nhiều nhất hai lần trong một giải pháp tối ưu, bởi vì chúng ta có thể thay thế các đồng xu $2 + 2 + 2$ bằng các đồng xu $5 + 1$ và các đồng

xu $20 + 20 + 20$ bằng các đồng xu $50 + 10$. Hơn nữa, một giải pháp tối ưu không thể chứa các đồng xu $2 + 2 + 1$ hoặc $20 + 20 + 10$, bởi vì chúng ta có thể thay thế chúng bằng các đồng xu 5 và 50 .

Sử dụng những quan sát này, chúng ta có thể chứng minh cho mỗi đồng xu x rằng không thể tạo ra một cách tối ưu một tổng x hoặc bất kỳ tổng lớn hơn nào bằng cách chỉ sử dụng các đồng xu nhỏ hơn x . Ví dụ, nếu $x = 100$, tổng tối ưu lớn nhất sử dụng 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 tạo ra giải pháp tối ưu.

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

Trường hợp tổng quát

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

Chúng ta có thể chứng minh rằng một thuật toán tham lam không hoạt động bằng cách đưa ra một phản ví dụ trong đó thuật toán cho ra một câu trả lời sai. Trong bài toán này, chúng ta có thể dễ dàng tìm thấy một phản ví dụ: nếu các đồng xu là $\{1, 3, 4\}$ và tổng mục tiêu là 6 , thuật toán tham lam tạo ra giải pháp $4 + 1 + 1$ trong khi giải pháp tối ưu là $3 + 3$.

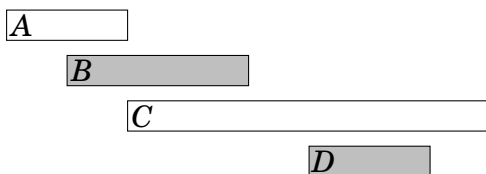
Không biết liệu bài toán đổi tiền tổng quát có thể được giải quyết bằng bất kỳ thuật toán tham lam nào không¹. Tuy nhiên, như chúng ta sẽ thấy trong Chương 7, trong một số trường hợp, bài toán tổng quát có thể được giải quyết hiệu quả bằng cách sử dụng một thuật toán quy hoạch động luôn cho ra câu trả lời đúng.

6.2 Lập lịch

Nhiều bài toán lập lịch có thể được giải quyết bằng thuật toán tham lam. Một bài toán kinh điển như sau: Cho n sự kiện với thời gian bắt đầu và kết thúc của chúng, tìm một lịch trình bao gồm càng nhiều sự kiện càng tốt. Không thể chọn một sự kiện một phần. Ví dụ, hãy xem xét các sự kiện sau:

sự kiện	thời gian bắt đầu	thời gian kết thúc
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

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

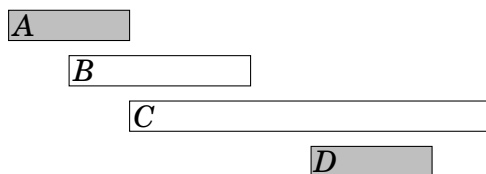


Có thể phát minh ra một số thuật toán tham lam cho bài toán, nhưng thuật toán nào trong số chúng hoạt động trong mọi trường hợp?

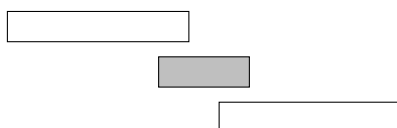
¹Tuy nhiên, có thể *kiểm tra* trong thời gian đa thức xem thuật toán tham lam được trình bày trong chương này có hoạt động cho một tập hợp đồng xu nhất định hay không [53].

Thuật toán 1

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



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



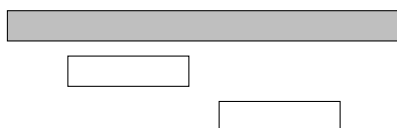
Nếu chúng ta chọn sự kiện ngắn, chúng ta chỉ có thể chọn một sự kiện. Tuy nhiên, có thể chọn cả hai sự kiện dài.

Thuật toán 2

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



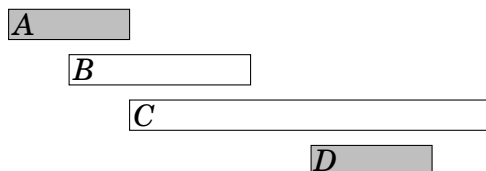
Tuy nhiên, chúng ta cũng có thể tìm thấy một phản ví dụ cho thuật toán này. Ví dụ, trong trường hợp sau, thuật toán chỉ chọn một sự kiện:



Nếu chúng ta chọn sự kiện đầu tiên, không thể chọn bất kỳ 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* càng *sớm* càng tốt. Thuật toán này chọn các sự kiện sau:



Hóa ra thuật toán này *luôn* tạo ra một giải pháp tối ưu. Lý do cho điều này là luôn là một lựa chọn tối ưu để chọn trước một sự kiện kết thúc càng sớm càng tốt. Sau đó, là một lựa chọn tối ưu để chọn sự kiện tiếp theo sử dụng cùng một chiến lược, v.v., cho đến khi chúng ta không thể chọn thêm bất kỳ sự kiện nào.

Một cách để chứng minh rằng thuật toán hoạt động là xem xét điều gì xảy ra nếu chúng ta chọn trước một sự kiện kết thúc muộn hơn sự kiện kết thúc càng sớm càng tốt. Bây giờ, chúng ta sẽ có nhiều nhất là một số lượng lựa chọn bằng nhau về cách chúng ta có thể chọn sự kiện tiếp theo. Do đó, việc chọn một sự kiện kết thúc muộn hơn không bao giờ có thể mang lại một giải pháp tốt hơn, và thuật toán tham lam là đúng.

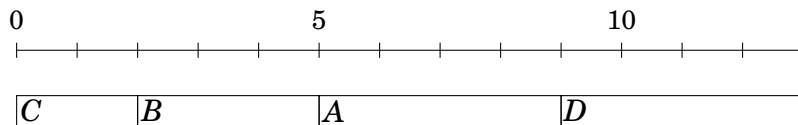
6.3 Nhiệm vụ và thời hạn

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

Ví dụ, giả sử các nhiệm vụ như sau:

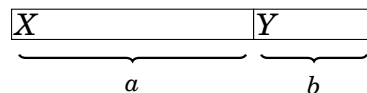
nhiệm vụ	thời lượng	thời hạn
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

Trong trường hợp này, một lịch trình tối ưu cho các nhiệm vụ như sau:

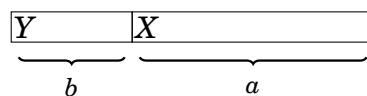


Trong giải pháp này, *C* mang lại 5 điểm, *B* mang lại 0 điểm, *A* mang lại -7 điểm và *D* mang lại -8 điểm, vì vậy tổng số điểm là -10.

Đáng ngạc nhiên, giải pháp tối ưu cho bài toán hoàn toàn không phụ thuộc vào thời hạn, mà một chiến lược tham lam đúng đắn là chỉ đơn giản thực hiện các nhiệm vụ *được sắp xếp theo thời lượng của chúng* theo thứ tự tăng dần. Lý do cho điều này là nếu chúng ta thực hiện hai nhiệm vụ liên tiếp sao cho nhiệm vụ đầu tiên mất nhiều thời gian hơn nhiệm vụ thứ hai, chúng ta có thể có được một giải pháp tốt hơn nếu chúng ta hoán đổi các nhiệm vụ. Ví dụ, hãy xem xét lịch trình sau:



Ở đây $a > b$, vì vậy chúng ta nên hoán đổi các nhiệm vụ:



Bây giờ *X* cho ít hơn b điểm và *Y* cho nhiều hơn a điểm, vì vậy tổng số điểm tăng thêm $a - b > 0$. Trong một giải pháp tối ưu, đối với bất kỳ hai nhiệm vụ liên tiếp nào, phải thỏa mãn rằng nhiệm vụ ngắn hơn đến trước nhiệm vụ dài hơn. Do đó, các nhiệm vụ phải được thực hiện được sắp xếp theo thời lượng của chúng.

6.4 Giảm thiểu tổng

Tiếp theo, chúng ta xem xét một bài toán trong đó chúng ta được cho n số a_1, a_2, \dots, a_n và nhiệm vụ của chúng ta là tìm một giá trị x để giảm thiểu tổng

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

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

Trường hợp $c = 1$

Trong trường hợp này, chúng ta nên giảm thiểu tổng

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

Ví dụ, nếu các số là $[1, 2, 9, 2, 6]$, giải pháp 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 là, 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, vì vậy trung vị là 2.

Trung vị là một lựa chọn tối ưu, bởi vì nếu x nhỏ hơn trung vị, tổng sẽ trở nên nhỏ hơn bằng cách tăng x , và nếu x lớn hơn trung vị, tổng sẽ trở nên nhỏ hơn bằng cách giảm x . Do đó, giải pháp tối ưu là x là trung vị. Nếu n là số chẵn và có hai trung vị, cả hai trung vị và tất cả các giá trị ở giữa chúng đều là các lựa chọn tối ưu.

Trường hợp $c = 2$

Trong trường hợp này, chúng ta nên giảm thiểu tổng

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

Ví dụ, nếu các số là $[1, 2, 9, 2, 6]$, giải pháp 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 biểu diễn 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 cùng không phụ thuộc vào x , vì vậy chúng ta có thể bỏ qua nó. Các phần còn lại tạo thành một hàm $nx^2 - 2xs$ trong đó $s = a_1 + a_2 + \dots + a_n$. Đây là một parabol mở lên trên với các nghiệm $x = 0$ và $x = 2s/n$, và giá trị nhỏ nhất là trung bình của các nghiệm $x = s/n$, tức là, trung bình của các số a_1, a_2, \dots, a_n .

6.5 Nén dữ liệu

Một **mã nhị phân** gán cho mỗi ký tự của một chuỗi một **từ mã** bao gồm các bit. Chúng ta có thể *nén* chuỗi bằng cách sử dụng mã nhị phân bằng cách thay thế mỗi ký tự bằng từ mã tương ứng. Ví dụ, mã nhị phân sau đây gán các từ mã cho các ký tự A–D:

ký tự	từ mã
A	00
B	01
C	10
D	11

Đây là một mã **độ dài không đổi** có nghĩa là độ dài của mỗi từ mã là như nhau. Ví dụ, chúng ta có thể nén chuỗi AABACDACA như sau:

000001001011001000

Sử dụng mã này, độ dài của chuỗi nén là 18 bit. Tuy nhiên, chúng ta có thể nén chuỗi tốt hơn nếu chúng ta sử dụng một mã **độ dài thay đổi** trong đó các từ mã có thể có độ dài khác nhau. Sau đó, chúng ta có thể đưa ra các từ mã ngắn cho các ký tự xuất hiện thường xuyên và các từ mã dài cho các ký tự xuất hiện hiếm khi. Hóa ra một mã **tối ưu** cho chuỗi trên là như sau:

ký tự	từ mã
A	0
B	110
C	10
D	111

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

001100101110100,

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

Chúng tôi yêu cầu không có từ mã nào là tiền tố của một từ mã khác. Ví dụ, không được phép một mã chứa cả hai từ mã 10 và 1011. Lý do cho điều này là chúng ta muốn có thể tạo ra chuỗi gốc từ chuỗi nén. Nếu một từ mã có thể là tiền tố của một từ mã khác, điều này không phải lúc nào cũng có thể. Ví dụ, mã sau đây là *không* hợp lệ:

ký tự	từ mã
A	10
B	11
C	1011
D	111

Sử dụng mã này, sẽ không thể biết liệu chuỗi nén 1011 tương ứng với chuỗi AB hay chuỗi C.

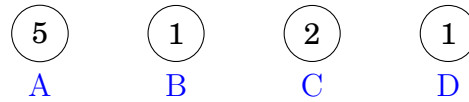
Mã hóa Huffman

Mã hóa Huffman² là một thuật toán tham lam xây dựng một mã tối ưu để nén một chuỗi nhất định. Thuật toán xây dựng một cây nhị phân dựa trên tần suất của các ký tự trong chuỗi, và từ mã của mỗi ký tự có thể được đọc bằng cách đi theo một đường dẫn từ gốc đến nút tương ứng. Một bước di chuyển sang trái tương ứng với bit 0, và một bước di chuyển sang phải tương ứng với bit 1.

²D. A. Huffman đã khám phá ra phương pháp này khi giải một bài tập khóa học đại học và công bố thuật toán vào năm 1952 [40].

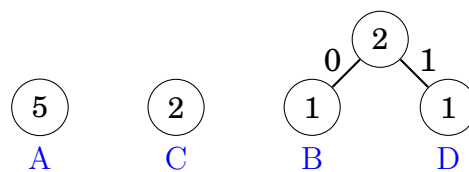
Ban đầu, mỗi ký tự của chuỗi được đại diện bởi một nút có trọng số là số lần ký tự xuất hiện trong chuỗi. Sau đó, ở mỗi bước, hai nút có trọng số nhỏ nhất được kết hợp bằng cách tạo một nút mới có trọng số là tổng trọng số của các nút ban đầu. Quá trình tiếp tục cho đến khi tất cả các nút đã được kết hợp.

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

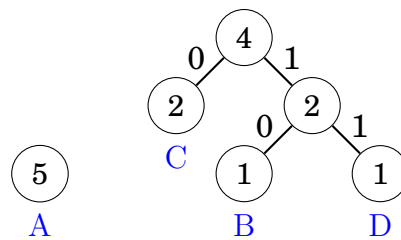


Nút đại diện cho ký tự A có trọng số 5 vì ký tự A xuất hiện 5 lần trong chuỗi. Các trọng số khác đã được tính toán theo cùng một cách.

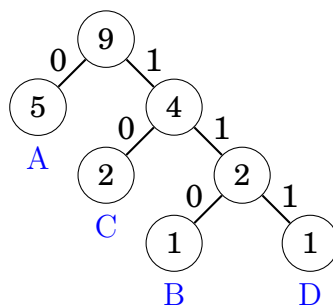
Bước đầu tiên là kết hợp các nút tương ứng với các ký tự B và D, cả hai đều có trọng số 1. Kết quả là:



Sau đó, các nút có trọng số 2 được kết hợp:



Cuối cùng, hai nút còn lại được kết hợp:



Bây giờ tất cả các nút đều nằm trong cây, vì vậy mã đã sẵn sàng. Các từ mã sau đây có thể được đọc từ cây:

ký tự	từ mã
A	0
B	110
C	10
D	111

Chương 7

Quy hoạch động

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

Có hai ứng dụng cho quy hoạch động:

- **Tìm một giải pháp tối ưu:** Chúng ta muốn tìm một giải pháp lớn nhất có thể hoặc nhỏ nhất có thể.
- **Đếm số lượng các giải pháp:** Chúng ta muốn tính tổng số các giải pháp có thể.

Đầu tiên, chúng ta sẽ xem quy hoạch động có thể được sử dụng để tìm một giải pháp tối ưu như thế nào, và sau đó chúng ta sẽ sử dụng cùng một ý tưởng để đếm các giải pháp.

Hiểu được quy hoạch động là một cột mốc quan trọng trong sự nghiệp của mọi lập trình viên thi đấu. Mặc dù ý tưởng cơ bản là đơn giản, thách thức là làm thế nào để áp dụng quy hoạch động vào các bài toán khác nhau. Chương này giới thiệu một tập hợp các bài toán kinh điển là một điểm khởi đầu tốt.

7.1 Bài toán đồng xu

Đầu tiên, chúng ta tập trung vào một bài toán mà chúng ta đã thấy trong Chương 6: Cho một tập hợp các mệnh giá đồng xu $\text{coins} = \{c_1, c_2, \dots, c_k\}$ và một tổng tiền mục tiêu n , nhiệm vụ của chúng ta là tạo thành tổng n bằng cách sử dụng càng ít đồng xu càng tốt.

Trong Chương 6, chúng ta đã giải quyết bài toán bằng cách sử dụng một thuật toán tham lam luôn chọn đồng xu lớn nhất có thể. Thuật toán tham lam hoạt động, ví dụ, khi các đồng xu là các đồng xu euro, nhưng trong trường hợp tổng quát, thuật toán tham lam không nhất thiết tạo ra một giải pháp tối ưu.

Bây giờ là lúc để giải quyết bài toán một cách hiệu quả sử dụng quy hoạch động, sao cho thuật toán hoạt động với bất kỳ bộ đồng xu nào. Thuật toán quy hoạch động dựa trên một hàm đệ quy duyệt qua tất cả các khả năng để tạo thành tổng, giống như một thuật toán duyệt toàn bộ. Tuy nhiên, thuật toán quy hoạch động hiệu quả vì nó sử dụng *ghi nhớ (memoization)* và tính toán câu trả lời cho mỗi bài toán con chỉ một lần.

Công thức đệ quy

Ý tưởng trong quy hoạch động là xây dựng bài toán một cách đệ quy sao cho lời giải của bài toán có thể được tính toán từ các lời giải của các bài toán con nhỏ hơn. Trong bài toán

đồng xu, một bài toán đệ quy tự nhiên như sau: số lượng đồng xu nhỏ nhất cần thiết để tạo thành một tổng x là bao nhiêu?

Gọi $\text{solve}(x)$ biểu thị số lượng đồng xu tối thiểu cần thiết cho một tổng x . Các giá trị của hàm phụ thuộc vào các 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, $\text{solve}(10) = 3$, bởi vì cần ít nhất 3 đồng xu để tạo thành tổng 10. Giải pháp tối ưu là $3 + 3 + 4 = 10$.

Thuộc tính thiết yếu của solve là các giá trị của nó có thể được tính toán một cách đệ quy từ các giá trị nhỏ hơn của nó. Ý tưởng là tập trung vào đồng xu *đầu tiên* mà chúng ta chọn cho tổng. Ví dụ, trong kịch bản trên, đồng xu đầu tiên có thể là 1, 3 hoặc 4. Nếu chúng ta chọn đồng xu 1 đầu tiên, nhiệm vụ còn lại là tạo thành tổng 9 sử dụng số lượng đồng xu tối thiểu, đó là một bài toán con của bài toán ban đầu. Tất nhiên, điều tương tự cũng áp dụng cho các đồng xu 3 và 4. Do đó, chúng ta có thể sử dụng công thức đệ quy sau để tính số lượng 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 $\text{solve}(0) = 0$, bởi vì không cần đồng xu nào để tạo thành một tổng rỗng. Ví dụ,

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

Bây giờ chúng ta sẵn sàng đưa ra một hàm đệ quy tổng quát tính toán số lượng đồng xu tối thiểu cần thiết để tạo thành một tổng x :

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

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

Một khi một hàm đệ quy giải quyết được bài toán đã được tìm thấy, chúng ta có thể triển khai trực tiếp một giải pháp trong C++ (hằng số INF biểu thị vô cùng):

```
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 không hiệu quả, bởi vì có thể có một số lượng mũ các cách để xây dựng tổng. Tuy nhiên, tiếp theo chúng ta sẽ xem cách làm cho hàm này hiệu quả bằng cách sử dụng một kỹ thuật gọi là ghi nhớ.

Sử dụng ghi nhớ (memoization)

Ý tưởng của quy hoạch động là sử dụng **ghi nhớ (memoization)** để tính toán hiệu quả các giá trị của một hàm đệ quy. Điều này có nghĩa là các giá trị của hàm được lưu trữ trong một mảng sau khi tính toán chúng. Đối với mỗi tham số, giá trị của hàm được tính toán đệ quy chỉ một lần, và sau đó, giá trị có thể được lấy trực tiếp từ mảng.

Trong bài toán này, chúng ta sử dụng các mảng

```

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

```

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

Bây giờ hàm có thể được triển khai một cách 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 từ `ready[x]` xem `solve(x)` đã được lưu trữ trong `value[x]` hay chưa, và nếu có, hàm sẽ trả về nó trực tiếp. Ngược lại, hàm sẽ tính giá trị của `solve(x)` một cách đệ quy và lưu nó vào `value[x]`.

Hàm này hoạt động hiệu quả, bởi vì câu trả lời cho mỗi tham số x được tính toán đệ quy chỉ một lần. Sau khi một giá trị của `solve(x)` đã được lưu trữ trong `value[x]`, nó có thể được lấy ra một cách hiệu quả bất cứ khi nào hàm sẽ 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 chúng ta cũng có thể xây dựng mảng `value` một cách *lập* bằng cách sử dụng một vòng lặp chỉ đơn giản là tính toán tất cả các giá trị của `solve` cho các 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 các lập trình viên thi đấu thích cách triển khai này hơn, bởi vì nó ngắn gọn và có các hệ số hằng thấp hơn. Từ bây giờ, chúng ta cũng sử dụng các cách triển khai lặp trong các ví dụ của mình. Tuy nhiên, thường thì việc suy nghĩ về các giải pháp quy hoạch động dưới dạng các hàm đệ quy sẽ dễ dàng hơn.

Xây dựng một giải pháp

Đôi khi chúng ta được yêu cầu vừa tìm giá trị của một giải pháp tối ưu vừa đưa ra một ví dụ về cách một giải pháp như vậy có thể được xây dựng. Trong bài toán đồng xu, ví dụ, chúng ta có thể khai báo một mảng khác chỉ ra cho mỗi tổng tiền đồng xu đầu tiên trong một giải pháp tối ưu:

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

Sau đó, chúng ta có thể sửa đổi 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 sử dụng để in ra các đồng xu xuất hiện trong một giải pháp tối ưu cho tổng n :

```

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

```

Đếm số lượng các giải pháp

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

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

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

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

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

Sau 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 giải pháp nào. Nếu $x = 0$, giá trị là 1, vì chỉ có một cách để tạo thành một tổng rỗng. Ngược lại, chúng ta tính tổng của tất cả các giá trị có dạng $\text{solve}(x - c)$ trong đó c nằm trong coins .

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

```
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 lượng các giải pháp lớn đến mức không cần phải tính số chính xác mà chỉ cần đưa ra câu trả lời theo mô-đun m trong đó, ví dụ, $m = 10^9 + 7$. Điều này có thể được thực hiện bằng cách thay đổi mã sao cho tất cả các tính toán được thực hiện theo mô-đun m . Trong đoạn mã trên, chỉ cần thêm dòng

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

sau dòng

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

Bây giờ chúng ta đã thảo luận về tất cả các ý tưởng cơ bản của quy hoạch động. Vì quy hoạch động có thể được sử dụng trong nhiều tình huống khác nhau, bây giờ chúng ta sẽ xem xét một tập hợp các bài toán để minh họa thêm về các khả năng của quy hoạch động.


7.2 Dãy con tăng dài nhất

Bài toán đầu tiên của chúng ta là tìm **dãy con tăng dài nhất (longest increasing subsequence)** trong một mảng gồm n phần tử. Đây là một dãy có độ dài tối đa gồm các phần tử mảng đi từ trái sang phải, và mỗi phần tử trong dãy lớn hơn phần tử trước đó. 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 chúng ta tính tất cả các giá trị của $\text{length}(k)$ trong đó $0 \leq k \leq n-1$, chúng ta sẽ tìm ra độ dài của dãy con tăng dài nhất. Ví dụ, các giá trị của hàm cho mảng trên như sau:

$\text{length}(0) = 1$
 $\text{length}(1) = 1$
 $\text{length}(2) = 2$
 $\text{length}(3) = 1$
 $\text{length}(4) = 3$
 $\text{length}(5) = 2$
 $\text{length}(6) = 4$
 $\text{length}(7) = 2$

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

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

Vì tất cả các giá trị của hàm có thể được tính từ các giá trị nhỏ hơn của nó, chúng ta có thể sử dụng quy hoạch động. Trong đoạn mã sau, các giá trị của hàm sẽ được lưu trữ trong một 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 hoạt động trong thời gian $O(n^2)$, bởi vì nó bao gồm hai vòng lặp lồng nhau. Tuy nhiên, cũng có thể triển khai việc tính toán quy hoạch động một cách hiệu quả hơn trong thời gian $O(n \log n)$. Bạn có thể tìm ra cách làm điều này không?

7.3 Đường đi trong lưới

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

Hình sau cho thấy một đường đi tối ưu trong một 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 các cột của lưới được đánh số từ 1 đến n , và $\text{value}[y][x]$ bằng giá trị của ô vuông (y, x) . Gọi $\text{sum}(y, x)$ là tổng lớn nhất trên một đường đi từ góc trên bên trái đến ô vuông (y, x) . Bây giờ $\text{sum}(n, n)$ cho chúng ta biết tổng lớn nhất từ góc trên bên trái đến góc dưới bên phải. Ví dụ, trong lưới trên, $\text{sum}(5, 5) = 67$.

Chúng ta có thể tính toán các tổng một cách 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 quan sát rằng một đường đi kết thúc tại ô vuông (y, x) có thể đến từ ô vuông $(y, x-1)$ hoặc ô vuông $(y-1, x)$:

			↓	
		→		

Do đó, chúng ta chọn hướng để tối đa hóa tổng. Chúng ta giả định rằng $\text{sum}(y, x) = 0$ nếu $y = 0$ hoặc $x = 0$ (bởi vì không có đường đi nào như vậy), vì vậy công thức đệ quy cũng hoạt động khi $y = 1$ hoặc $x = 1$.

Vì hàm sum có hai tham số, mảng quy hoạch động cũng có hai chiều. Ví dụ, chúng ta có thể sử dụng một 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 Bài toán cái túi

Thuật ngữ **cái túi (knapsack)** đề cập đến các bài toán trong đó một tập hợp các đồ vật được cho, và các tập hợp con có một số thuộc tính cần được tìm thấy. Các bài toán cái túi thường có thể được giải quyết bằng quy hoạch động.

Trong phần này, chúng ta tập trung vào bài toán sau: Cho một danh sách các trọng lượng $[w_1, w_2, \dots, w_n]$, xác định tất cả các tổng có thể được xây dựng bằng cách sử dụng các trọng lượng. Ví dụ, nếu các trọng lượng là $[1, 3, 3, 5]$, các tổng sau đây 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, tất cả các tổng trong khoảng $0 \dots 12$ đều có thể, ngoại trừ 2 và 10. Ví dụ, tổng 7 là có thể vì chúng ta có thể chọn các trọng lượng $[1, 3, 3]$.

Để giải quyết bài toán, chúng ta tập trung vào các bài toán con trong đó chúng ta chỉ sử dụng k trọng lượng đầu tiên để xây dựng các tổng. Gọi $\text{possible}(x, k) = \text{true}$ nếu chúng ta có thể xây dựng một tổng x sử dụng k trọng lượng đầu tiên, và ngược lại $\text{possible}(x, k) = \text{false}$. Các giá trị của hàm có thể được tính một cách 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 thực tế là chúng ta có thể sử dụng hoặc không sử dụng trọng lượng w_k trong tổng. Nếu chúng ta sử dụng w_k , nhiệm vụ còn lại là tạo thành tổng $x - w_k$ bằng cách sử dụng $k - 1$ trọng lượng đầu tiên, và nếu chúng ta không sử dụng w_k , nhiệm vụ còn lại là tạo thành tổng x bằng cách sử dụng $k - 1$ trọng lượng đầu tiên. Là các trường hợp cơ sở,

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

bởi vì nếu không có trọng lượng nào được sử dụng, chúng ta chỉ có thể tạo thành tổng 0.

Bảng sau cho thấy tất cả các giá trị của hàm cho các trọng lượng $[1, 3, 3, 5]$ (ký hiệu "X" chỉ ra 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 chúng ta biết liệu chúng ta có thể xây dựng một tổng x bằng cách sử dụng *tất cả* các trọng lượng hay không.

Gọi W là tổng trọng lượng của các trọng lượng. Giải pháp quy hoạch động thời gian $O(nW)$ sau 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 cách triển khai tốt hơn chỉ sử dụng một mảng một chiều `possible[x]` chỉ ra liệu chúng ta có thể xây dựng một tập hợp con với tổng x hay không. Bí quyết là cập nhật mảng từ phải sang trái cho 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 chung được trình bày ở đây có thể được sử dụng trong nhiều bài toán cái túi. Ví dụ, nếu chúng ta được cho các đồ vật có trọng lượng và giá trị, chúng ta có thể xác định cho mỗi tổng trọng lượng tổng giá trị tối đa của một tập hợp con.

7.5 Khoảng cách chỉnh sửa

Khoảng cách chỉnh sửa (edit distance) hoặc **khoảng cách Levenshtein (Levenshtein distance)**¹ là số lượng tối thiểu các thao tác chỉnh sửa cần thiết để biến một chuỗi thành một chuỗi khác. Các thao tác chỉnh sửa được phép như sau:

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

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

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

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

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

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

- $distance(a, b-1)$: chèn một ký tự vào cuối x
- $distance(a-1, b)$: xóa ký tự cuối cùng khỏi x
- $distance(a-1, b-1)$: khớp hoặc sửa đổi ký tự cuối cùng của x

¹Khoảng cách được đặt theo tên của V. I. Levenshtein, người đã nghiên cứu nó liên quan đến các mã nhị phân [49].

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

Bảng sau cho thấy các giá trị của distance trong trường hợp 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

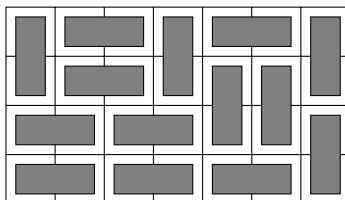
Góc dưới bên phải của bảng cho chúng ta biết rằng khoảng cách chỉnh sửa giữa LOVE và MOVIE là 2. Bảng cũng cho thấy cách xây dựng chuỗi thao tác chỉnh sửa ngắn nhất. Trong trường hợp này, đường đi như sau:

		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 cùng của LOVE và MOVIE bằng nhau, vì vậy khoảng cách chỉnh sửa giữa chúng bằng khoảng cách chỉnh sửa giữa LOV và MOVI. Chúng ta có thể sử dụng một thao tác chỉnh sửa để xóa ký tự I khỏi MOVI. Do đó, khoảng cách chỉnh sửa lớn hơn một so với khoảng cách chỉnh sửa giữa LOV và MOV, v.v.

7.6 Đếm số cách lát gạch

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



và tổng số giải pháp là 781.

Bài toán có thể được giải quyết bằng quy hoạch động bằng cách duyệt qua lưới từng hàng một. Mỗi hàng trong một giải pháp có thể được biểu diễn dưới dạng một chuỗi chứa m ký tự từ tập hợp $\{\square, \sqcup, \sqsubset, \sqsupset\}$. Ví dụ, giải pháp trên bao gồm bốn hàng tương ứng với các chuỗi sau:

- $\square \sqsubset \square \square \sqsubset \square$

- $\sqcup \square \square \sqcup \sqcap \sqcap \sqcup$
- $\square \square \square \sqcup \sqcup \sqcap$
- $\square \square \square \square \square \sqcup$

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

Một giải pháp hợp lệ nếu hàng 1 không chứa ký tự \sqcup , hàng n không chứa ký tự \sqcap , và tất cả các hàng liên tiếp đều *tương thích*. Ví dụ, các hàng $\sqcup \square \square \sqcup \sqcap \sqcap \sqcup$ và $\square \square \square \sqcup \sqcup \sqcap$ là tương thích, trong khi các hàng $\sqcap \square \square \sqcap \square \sqcap \sqcap$ và $\square \square \square \square \square \sqcup$ không tương thích.

Vì một hàng bao gồm m ký tự và có bốn lựa chọn cho mỗi ký tự, số lượng hàng phân biệt nhiều nhất là 4^m . Do đó, độ phức tạp thời gian của giải pháp là $O(n4^{2m})$ bởi vì chúng ta có thể duyệt qua $O(4^m)$ trạng thái có thể có cho mỗi hàng, và đối với mỗi trạng thái, có $O(4^m)$ trạng thái có thể có cho hàng trước đó. Trong thực tế, một ý tưởng tốt là xoay lưới sao cho cạnh ngắn hơn có độ dài m , bởi vì hệ số 4^{2m} chiếm ưu thế trong độ phức tạp thời gian.

Có thể làm cho giải pháp hiệu quả hơn bằng cách sử dụng một biểu diễn gọn hơn cho các hàng. Hóa ra chỉ cần biết các cột nào của hàng trước đó chứa ô vuông phía trên của một viên gạch dọc. Do đó, chúng ta có thể biểu diễn một hàng chỉ bằng cách sử dụng các ký tự \sqcap và \square , trong đó \square là một sự kết hợp của các ký tự \sqcup , \square và \sqcap . Sử dụng biểu diễn này, chỉ có 2^m hàng phân biệt và độ phức tạp thời gian là $O(n2^{2m})$.

Lưu ý cuối cùng, cũng có một công thức trực tiếp đáng ngạc nhiên để tính số lượng các 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, bởi vì nó tính số lượng các cách lát gạch trong thời gian $O(nm)$, nhưng vì câu trả lời là một tích của các số thực, một vấn đề khi sử dụng công thức là làm thế nào để lưu trữ các kết quả trung gian một cách chính xác.

²Đáng ngạc nhiên, công thức này được khám phá vào năm 1961 bởi hai nhóm nghiên cứu [43, 67] làm việc độc lập.

Chương 8

Phân tích trừ dần

Độ phức tạp thời gian của một thuật toán thường dễ dàng phân tích chỉ bằng cách kiểm tra cấu trúc của thuật toán: thuật toán chứa những vòng lặp nào và các vòng lặp được thực hiện bao nhiêu lần. Tuy nhiên, đôi khi một phân tích trực tiếp không mang lại bức tranh chân thực về hiệu quả của thuật toán.

Phân tích trừ dần (Amortized analysis) có thể được sử dụng để phân tích các thuật toán chứa các thao tác mà độ phức tạp thời gian của chúng thay đổi. Ý tưởng là ước tính tổng thời gian được sử dụng cho tất cả các thao tác như vậy trong suốt quá trình thực thi của thuật toán, thay vì tập trung vào các thao tác riêng lẻ.

8.1 Phương pháp hai con trỏ

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

Tổng của mảng con

Ví dụ đầu tiên, hãy xem xét một bài toán mà chúng ta được cho một mảng gồm n số nguyên dương và một tổng mục tiêu x , và chúng ta muốn tìm một mảng con có tổng là x hoặc báo cáo rằng không có mảng con nào như vậy.

Ví dụ, mảng

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

chứa một mảng con có tổng là 8:

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

Bài toán này có thể được giải quyết trong thời gian $O(n)$ bằng cách sử dụng phương pháp hai con trỏ. Ý tưởng là duy trì các con trỏ trỏ đến giá trị đầu tiên và cuối cùng của một mảng con. Ở mỗi lượt, con trỏ trái di chuyển một bước sang phải, và con trỏ phải di chuyển sang phải miễn là tổng của mảng con kết quả không vượt quá x . Nếu tổng trở thành chính xác là x , một giải pháp đã được tìm thấy.

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

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

Mảng con ban đầu chứa các giá trị 1, 3 và 2 có tổng là 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, bởi vì nếu không tổng của mảng 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 của mảng con là $2 + 5 + 1 = 8$, vì vậy một mảng con có tổng là 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. Mặc dù không có giới hạn trên hữu ích nào về số bước mà con trỏ có thể di chuyển trong một lượt *đơn lẻ*, chúng ta biết rằng con trỏ di chuyển *tổng cộng* $O(n)$ bước trong suốt thuật toán, bởi vì nó chỉ di chuyển sang phải.

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

Bài toán 2SUM

Một bài toán khác có thể được giải quyết bằng phương pháp hai con trỏ là bài toán sau, còn được biết đến với tên gọi **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 là x , hoặc báo cáo rằng không tồn tại các giá trị như vậy.

Để giải quyết bài toán, trước tiên chúng ta sắp xếp các giá trị trong mảng theo thứ tự tăng dần. Sau đó, chúng ta lập qua mảng bằng hai con trỏ. Con trỏ trái bắt đầu tại giá trị đầu tiên và di chuyển một bước sang phải ở mỗi lượt. Con trỏ phải bắt đầu tại giá trị cuối cùng và luôn di chuyển sang trái cho đến khi tổng của giá trị ở con trỏ trái và phải không vượt quá x . Nếu tổng chính xác là x , một giải pháp đã được tìm thấy.

Ví dụ, hãy xem 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 một giải pháp $5 + 7 = 12$ đã được tìm thấy.

1	4	5	6	7	9	9	10
		↑		↑			

Thời gian chạy của thuật toán là $O(n \log n)$, bởi vì nó trước tiên 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 quyết bài toán theo một cách khác trong thời gian $O(n \log n)$ bằng cách sử dụng tìm kiếm nhị phân. Trong giải pháp như vậy, chúng ta lặp qua mảng và với mỗi giá trị của mảng, chúng ta cố gắng tìm một giá trị khác tạo ra tổng x . Điều này có thể được thực hiện bằng cách thực hiện n lần tìm kiếm nhị phân, mỗi lần 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 là x . Sử dụng ý tưởng của thuật toán trên, bài toán này có thể được giải quyết trong thời gian $O(n^2)$ ¹. Bạn có thấy cách làm không?

8.2 Phần tử nhỏ hơn gần nhất

Phân tích trừ dần thường được sử dụng để ước tính số lượng các thao tác được thực hiện trên một cấu trúc dữ liệu. Các thao tác có thể được phân bổ không đồng đều sao cho hầu hết các thao tác xảy ra trong một giai đoạn nhất định của thuật toán, nhưng tổng số lượng các thao tác bị giới hạn.

Ví dụ, hãy xem xét bài toán tìm cho mỗi phần tử của mảng **phần tử nhỏ hơn gần nhất**, tức là, phần tử nhỏ hơn đầu tiên đứng trước phần tử đó trong mảng. Có thể không tồn tại phần tử nào như vậy, trong trường hợp đó thuật toán nên báo cáo điều này. Tiếp theo chúng ta sẽ xem làm thế nào bài toán có thể được giải quyết một cách hiệu quả bằng cách sử dụng cấu trúc ngăn xếp.

Chúng ta duyệt qua mảng từ trái sang phải và duy trì một ngăn xếp các phần tử của mảng. Tại mỗi vị trí trong mảng, chúng 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 trống. Sau đó, chúng ta báo cáo rằng phần tử trên đỉnh là phần tử nhỏ hơn gần nhất của phần tử hiện tại, hoặc nếu ngăn xếp trống, không có phần tử nào như vậy. Cuối cùng, chúng ta thêm phần tử hiện tại vào ngăn xếp.

Ví dụ, hãy xem xét mảng sau:

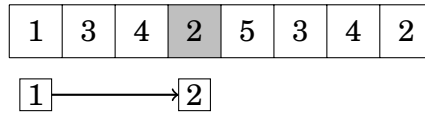
1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

Đầu tiên, các phần tử 1, 3 và 4 được thêm vào ngăn xếp, bởi vì mỗi phần tử 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.

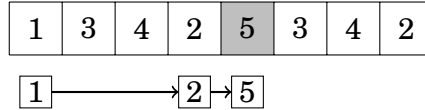
1	3	4	2	5	3	4	2
1	→	3	→	4			

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

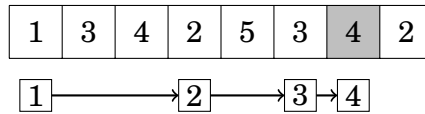
¹Trong một thời gian dài, người ta cho rằng việc giải quyết bài toán 3SUM hiệu quả hơn $O(n^2)$ là không thể. Tuy nhiên, vào năm 2014, hóa ra [30] điều này không đúng.



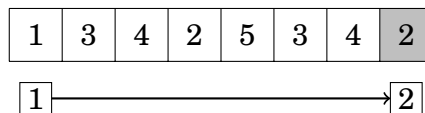
Sau đó, phần tử 5 lớn hơn phần tử 2, vì vậy nó sẽ được thêm vào ngăn xếp, và phần tử nhỏ hơn gần nhất của nó là 2:



Sau đó, phần tử 5 được loại bỏ khỏi ngăn xếp và các phần tử 3 và 4 được thêm vào ngăn xếp:



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



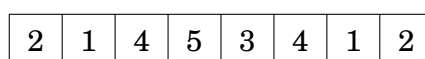
Hiệu quả của thuật toán phụ thuộc vào tổng số lượng các thao tác trên ngăn xếp. Nếu phần tử hiện tại lớn hơn phần tử trên đỉnh trong ngăn xếp, nó được thêm trực tiếp vào ngăn xếp, điều này rất hiệu quả. Tuy nhiên, đôi khi ngăn xếp có thể chứa một vài phần tử lớn hơn và mất thời gian để loại bỏ chúng. Tuy nhiên, mỗi phần tử được thêm *chính xác một lần* vào ngăn xếp và bị loại bỏ *nhieu nhất một lần* khỏi ngăn xếp. Do đó, mỗi phần tử gây ra $O(1)$ thao tác trên ngăn xếp, và thuật toán hoạt động trong thời gian $O(n)$.

8.3 Giá trị nhỏ nhất trong cửa sổ trượt

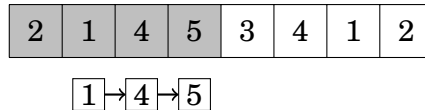
Một **cửa sổ trượt** (sliding window) là một mảng con có kích thước không đổi di chuyển từ trái sang phải qua mảng. Tại mỗi vị trí của cửa sổ, chúng ta muốn tính toán một số thông tin về các phần tử bên trong cửa sổ. Trong phần này, chúng ta tập trung vào bài toán duy trì **giá trị nhỏ nhất trong cửa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

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

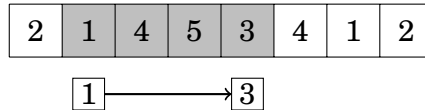
Ví dụ, hãy xem xét mảng sau:



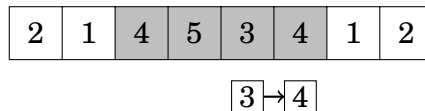
Giả sử kích thước của cửa sổ trượt là 4. Tại vị trí cửa sổ đầu tiên, giá trị nhỏ nhất là 1:



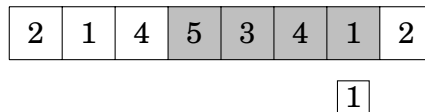
Sau đó cửa sổ di chuyển một bước sang phải. Phần tử mới 3 nhỏ hơn các phần tử 4 và 5 trong hàng đợi, vì vậy các phần tử 4 và 5 bị loại bỏ khỏi hàng đợi và phần tử 3 được thêm vào hàng đợi. Giá trị nhỏ nhất vẫn là 1.



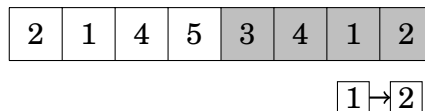
Sau đó, cửa sổ lại di chuyển, và phần tử nhỏ nhất 1 không còn thuộc cửa sổ nữa. Do đó, nó bị loại bỏ khỏi hàng đợi và giá trị nhỏ nhất bây giờ là 3. Phần tử mới 4 cũng được thêm vào hàng đợi.



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



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



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

Chương 9

Truy vấn đoạn

Trong chương này, chúng ta sẽ thảo luận về các cấu trúc dữ liệu cho phép chúng ta xử lý hiệu quả các truy vấn đoạn. Trong một **truy vấn đoạn** (range query), nhiệm vụ của chúng ta là tính toán một giá trị dựa trên một mảng con của một mảng. Các truy vấn đoạn điển hình 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, hãy xem 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 các truy vấn đoạn là sử dụng một vòng lặp duyệt qua tất cả các giá trị của mảng trong đoạn đó. Ví dụ, hàm sau có thể được sử dụng 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 hoạt động trong thời gian $O(n)$, trong đó n là kích thước của mảng. Do đó, chúng ta có thể xử lý q truy vấn trong thời gian $O(nq)$ bằng cách sử dụng hàm này. Tuy nhiên, nếu cả n và q đều lớn, phương pháp này sẽ chậm. May mắn thay, hóa ra có những cách để xử lý các truy vấn đoạn hiệu quả hơn nhiều.

9.1 Truy vấn trên mảng tĩnh

Trước tiên, chúng ta tập trung vào một tình huống mà 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 chúng ta câu trả lời cho bất kỳ truy vấn nào có thể có.

Truy vấn tổng

Chúng ta có thể dễ dàng xử lý các truy vấn tổng trên một mảng tĩnh bằng cách xây dựng một **mảng tổng tiền tố** (prefix sum array). Mỗi giá trị trong mảng tổng tiền tố bằng tổng các giá trị trong mảng ban đầu cho 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, hãy xem 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 $\text{sum}_q(0, k)$, chúng 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, hãy xem 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, chúng ta có thể xây dựng một mảng tổng tiền tố hai chiều có thể được sử dụng 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 ảnh sau đây minh họa ý tưởng:

		D			C		
		B			A		

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

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

trong đó $S(X)$ biểu thị tổng các giá trị trong một mảng con hình chữ nhật từ góc trên bên trái đến vị trí của X .

Truy vấn min

Truy vấn min khó xử lý hơn truy vấn tổng. Tuy nhiên, có một phương pháp tiền xử lý khá đơn giản với thời gian $O(n \log n)$ sau đó chúng ta có thể trả lời bất kỳ truy vấn min nào trong thời gian $O(1)$ ¹. Lưu ý rằng vì các truy vấn min và max có thể được xử lý tương tự nhau, chúng ta có thể tập trung vào các truy vấn min.

Ý tưởng là tính toán 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, đối với mảng

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

các giá trị sau được tính toán:

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						

Số lượng các giá trị được tính toán trước là $O(n \log n)$, bởi vì có $O(\log n)$ độ dài đoạn là lũy thừa của hai. Các giá trị có thể được tính toán hiệu quả bằng công thức đệ quy

$$\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b)),$$

trong đó $b - a + 1$ là một lũy thừa của hai và $w = (b - a + 1)/2$. Tính toán tất cả các giá trị đó mất thời gian $O(n \log n)$.

Sau đó, bất kỳ giá trị nào của $\min_q(a, b)$ có thể được tính trong thời gian $O(1)$ như là giá trị nhỏ nhất của hai giá trị đã được tính toán trước. Gọi k là lũy thừa lớn nhất của hai không vượt quá $b - a + 1$. Chúng 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 là hợp của các đoạn $[a, a + k - 1]$ và $[b - k + 1, b]$, cả hai đều có độ dài k .

Ví dụ, hãy xem 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 các đoạn $[1, 4]$ và $[3, 6]$:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	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] mà thời gian tiền xử lý chỉ là $O(n)$, nhưng những thuật toán như vậy không cần thiết trong lập trình thi đấu.

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

Vì $\min_q(1, 4) = 3$ và $\min_q(3, 6) = 1$, chúng ta kết luận rằng $\min_q(1, 6) = 1$.

9.2 Cây chỉ số nhị phân (Binary indexed tree)

Một **cây chỉ số nhị phân** (binary indexed tree) hay một **cây Fenwick**² có thể được xem như một biến thể động của mảng tổng tiền tố. Nó hỗ trợ hai thao tác thời gian $O(\log n)$ trên một mảng: xử lý một truy vấn tổng đoạn và cập nhật một giá trị.

Ưu điểm của một cây chỉ số nhị phân là nó cho phép chúng ta cập nhật hiệu quả các giá trị của mảng giữa các truy vấn tổng. Điều này sẽ không thể thực hiện được bằng cách sử dụng một mảng tổng tiền tố, bởi vì sau mỗi lần cập nhật, sẽ cần phải xây dựng lại toàn bộ mảng tổng tiền tố trong thời gian $O(n)$.

Cấu trúc

Mặc dù tên của cấu trúc là một *cây* chỉ số nhị phân, nó thường được biểu diễn dưới dạng một mảng. Trong phần này, chúng ta giả định rằng tất cả các mảng đều được đánh chỉ số từ một, bởi vì nó làm cho việc cài đặt dễ dàng hơn.

Gọi $p(k)$ là lũy thừa lớn nhất của hai mà chia hết cho k . Chúng ta lưu trữ một cây chỉ số nhị phân dưới dạng một mảng *tree* sao cho

$$\text{tree}[k] = \text{sum}_q(k - p(k) + 1, k),$$

tức là, mỗi vị trí k chứa tổng các giá trị trong một đoạn của mảng ban đầu có độ dài là $p(k)$ và kết thúc tại vị trí k . Ví dụ, vì $p(6) = 2$, $\text{tree}[6]$ chứa giá trị của $\text{sum}_q(5, 6)$.

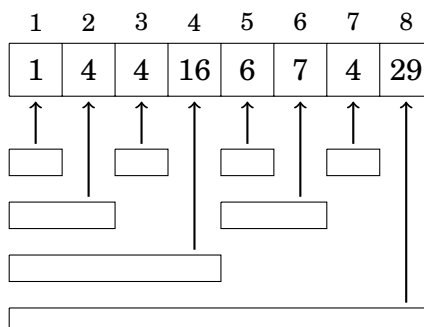
Ví dụ, hãy xem xét mảng sau:

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

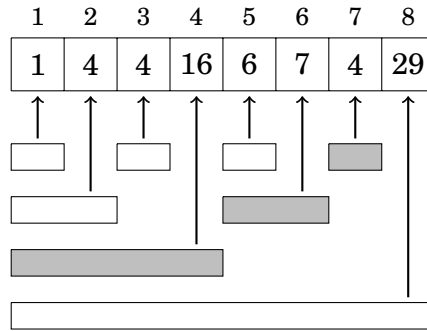
Hình ảnh sau đây cho thấy rõ hơn cách mỗi giá trị trong cây chỉ số nhị phân tương ứng với một đoạn trong mảng ban đầu:



²Cấu trúc cây chỉ số nhị phân được trình bày bởi P. M. Fenwick vào năm 1994 [21].

Sử dụng một cây chỉ số nhị phân, bất kỳ giá trị nào của $\text{sum}_q(1, k)$ đều có thể được tính trong thời gian $O(\log n)$, bởi vì một đoạn $[1, k]$ luôn có thể được chia thành $O(\log n)$ đoạn mà tổng của chúng được lưu trữ trong cây.

Ví dụ, đoạn $[1, 7]$ bao gồm các đoạn sau:



Do đó, chúng ta có thể tính tổng tương ứng như sau:

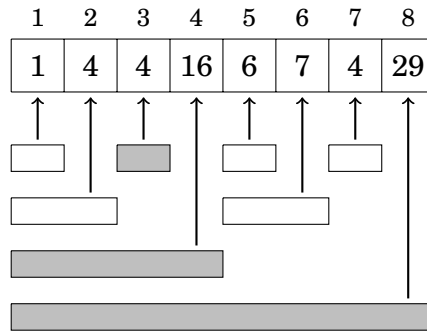
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

Để tính giá trị của $\text{sum}_q(a, b)$ trong đó $a > 1$, chúng ta có thể sử dụng mẹo tương tự mà chúng ta đã sử dụng với mảng tổng tiền tố:

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

Vì chúng ta có thể tính cả $\text{sum}_q(1, b)$ và $\text{sum}_q(1, a - 1)$ trong thời gian $O(\log n)$, tổng độ phức tạp thời gian là $O(\log n)$.

Sau đó, sau khi cập nhật một giá trị trong mảng ban đầu, một vài giá trị trong cây chỉ số nhị phân cần được cập nhật. Ví dụ, nếu giá trị tại vị trí 3 thay đổi, tổng của các đoạn sau sẽ thay đổi:



Vì mỗi phần tử của mảng thuộc về $O(\log n)$ đoạn trong cây chỉ số nhị phân, chỉ cần cập nhật $O(\log n)$ giá trị trong cây.

Cài đặt

Các thao tác của một cây chỉ số nhị phân có thể được cài đặt hiệu quả bằng cách sử dụng các phép toán bit. Sự thật quan trọng cần thiết là chúng 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 lên x (x có thể là 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 $O(\log n)$, bởi vì các hàm truy cập $O(\log n)$ giá trị trong cây chỉ số nhị phân, và mỗi lần di chuyển đến vị trí tiếp theo mất thời gian $O(1)$.

9.3 Cây đoạn (Segment tree)

Một **cây đoạn** (segment tree)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 các truy vấn tổng, truy vấn min và max và nhiều truy vấn khác sao cho cả hai thao tác đều hoạt động trong thời gian $O(\log n)$.

So với một 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 các truy vấn tổng⁴, cây đoạn cũng hỗ trợ các truy vấn khác. Mặt khác, một cây đoạn yêu cầu nhiều bộ nhớ hơn và khó cài đặt hơn một chút.

Cấu trúc

Một cây đoạn là một cây nhị phân sao cho các nút ở tầng dưới cùng của cây tương ứng với các phần tử của mảng, và các nút khác chứa thông tin cần thiết để xử lý các truy vấn đoạn.

Trong phần này, chúng ta giả định rằng kích thước của mảng là một lũy thừa của hai và sử dụng chỉ số dựa trên không, bởi vì việc xây dựng một cây đoạn cho một mảng như vậy rất thuận tiện. Nếu kích thước của mảng không phải là một lũy thừa của hai, chúng ta luôn có thể thêm các phần tử phụ vào nó.

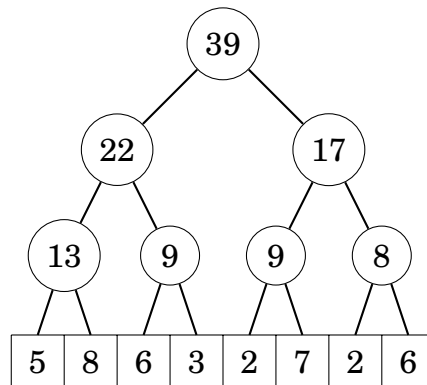
Đầu tiên chúng ta sẽ thảo luận về cây đoạn hỗ trợ các truy vấn tổng. Ví dụ, hãy xem xét mảng sau:

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

³Việc cài đặt từ dưới lên trong chương này tương ứng với cách trong [62]. Các cấu trúc tương tự đã được sử dụng vào cuối những năm 1970 để giải quyết các bài toán hình học [9].

⁴Thực tế, bằng cách sử dụng *hai* cây chỉ số nhị phân, có thể hỗ trợ các truy vấn min [16], nhưng điều này phức tạp hơn so với việc sử dụng một cây đoạn.

Cây đoạn tương ứng như sau:

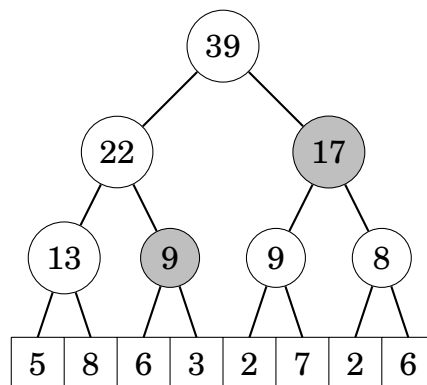


Mỗi nút trong cây tương ứng với một đoạn của mảng có kích thước là một lũy thừa của hai. Trong cây trên, giá trị của mỗi nút trong là tổng của các giá trị mảng tương ứng, và nó có thể được tính bằng tổng của giá trị của nút con trái và phải của nó.

Hóa ra bất kỳ đoạn nào $[a, b]$ cũng có thể được chia thành $O(\log n)$ đoạn mà giá trị của chúng được lưu trữ trong các nút của cây. Ví dụ, hãy xem 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 đây tương ứng với đoạn đó:

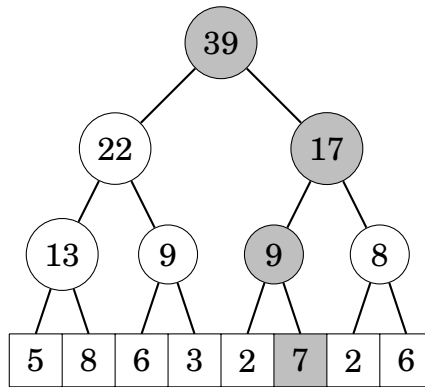


Do đó, một cách khác để tính tổng là $9 + 17 = 26$.

Khi tổng được tính bằng cách sử dụng các nút nằm càng cao càng tốt trong cây, cần nhiều nhất hai nút ở mỗi tầng của cây. Do đó, tổng số nút là $O(\log n)$.

Sau khi cập nhật mảng, chúng ta nên cập nhật tất cả các nút mà giá trị của chúng phụ thuộc vào giá trị được cập nhật. Điều này có thể được thực hiện bằng cách đi theo đường từ phần tử mảng được cập nhật đến nút gốc và cập nhật các nút dọc theo đường đi.

Hình ảnh sau đây cho thấy những nút cây nào thay đổi nếu giá trị 7 của mảng thay đổi:

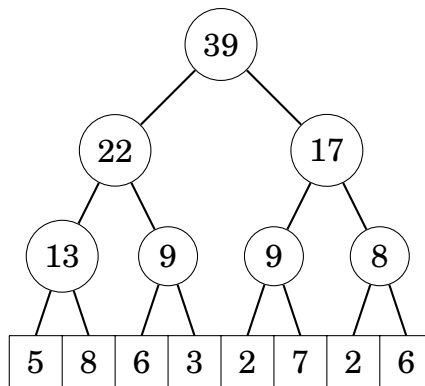


Đường đi từ dưới lên trên luôn bao gồm $O(\log n)$ nút, vì vậy mỗi lần cập nhật thay đổi $O(\log n)$ nút trong cây.

Cài đặt

Chúng ta lưu trữ một 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 của cây được lưu trữ từ trên xuống dưới: $\text{tree}[1]$ là nút gốc, $\text{tree}[2]$ và $\text{tree}[3]$ là các con của nó, và cứ thế. 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 tầng dưới cùng của cây.

Ví dụ, cây đoạn



được lưu trữ 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

Sử dụng biểu diễn này, cha của $\text{tree}[k]$ là $\text{tree}[\lceil k/2 \rceil]$, và các con của nó là $\text{tree}[2k]$ và $\text{tree}[2k + 1]$. Lưu ý rằng điều này ngụ ý rằng 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 được di chuyển lên một tầng cao hơn trong cây, và trước đó, các giá trị của các nút không thuộc đoạn cao hơn được cộng vào tổng.

Hàm sau tăng giá trị của mảng tại vị trí k lên 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];
    }
}

```

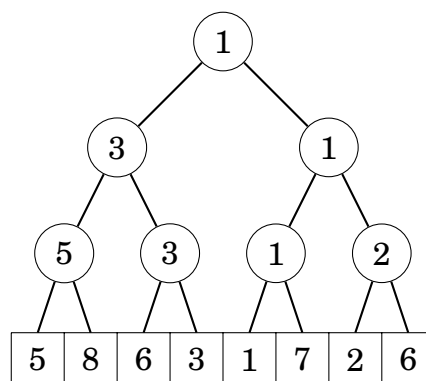
Đầu tiên, hàm cập nhật giá trị ở tầng dưới cùng của cây. Sau đó, hàm cập nhật giá trị của tất cả các nút trong cây, cho đến khi nó đến nút gốc của cây.

Cả hai hàm trên đều hoạt động trong thời gian $O(\log n)$, bởi vì một cây đoạn của n phần tử bao gồm $O(\log n)$ tầng, và các hàm di chuyển lên một tầng cao hơn trong cây ở mỗi bước.

Các truy vấn khác

Cây đoạn có thể hỗ trợ tất cả các truy vấn đoạn mà có thể chia một đoạn thành hai phần, tính toán câu trả lời riêng cho cả hai phần và sau đó kết hợp hiệu quả các câu trả lời. Ví dụ về các truy vấn như vậy là min và max, ước chung lớn nhất, và các phép toán bit and, or và xor.

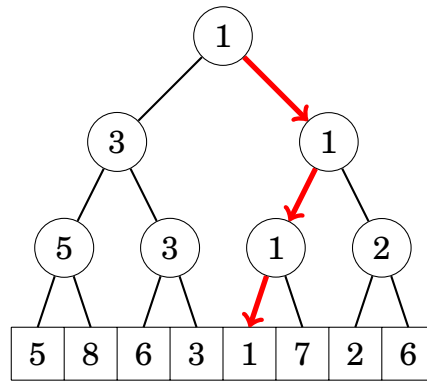
Ví dụ, cây đoạn sau hỗ trợ các truy vấn min:



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 đây, nhưng thay vì tổng, giá trị nhỏ nhất được tính toán.

Cấu trúc của một cây đoạn cũng cho phép chúng ta sử dụng tìm kiếm nhị phân để định vị các phần tử của mảng. Ví dụ, nếu cây hỗ trợ các truy vấn min, chúng 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 là 1 có thể được tìm thấy bằng cách đi theo một đường từ nút gốc xuống dưới:



9.4 Các kỹ thuật bổ sung

Nén chỉ số

Một hạn chế trong các cấu trúc dữ liệu được xây dựng trên một mảng là các phần tử được đánh chỉ số bằng các số nguyên liên tiếp. Khó khăn nảy sinh khi cần các chỉ số lớn. Ví dụ, nếu chúng ta muốn sử dụng chỉ số 10^9 , mảng phải chứa 10^9 phần tử, điều này sẽ đòi hỏi quá nhiều bộ nhớ.

Tuy nhiên, chúng ta thường có thể vượt qua hạn chế này bằng cách sử dụng **nén chỉ số** (index compression), trong đó các chỉ số ban đầu được thay thế bằng các chỉ số 1, 2, 3, v.v. Điều này có thể được thực hiện nếu chúng ta biết trước tất cả các chỉ số cần thiết trong suốt thuật toán.

Ý tưởng là thay thế mỗi chỉ số ban đầu x bằng $c(x)$ trong đó c là một hàm nén các chỉ số. Chúng ta yêu cầu thứ tự của các chỉ số không thay đổi, vì vậy nếu $a < b$, thì $c(a) < c(b)$. Điều này cho phép chúng ta thực hiện các truy vấn một cách thuận tiện ngay cả khi các chỉ số 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, chúng ta đã cài đặt các cấu trúc dữ liệu hỗ trợ các truy vấn đoạn và cập nhật các giá trị đơn lẻ. Bây giờ chúng ta hãy xem xét một tình huống ngược lại, trong đó chúng ta nên cập nhật các đoạn và truy xuất các giá trị đơn lẻ. Chúng ta tập trung vào một thao tác tăng tất cả các phần tử trong một đoạn $[a, b]$ lên x .

Đáng ngạc nhiên là chúng ta cũng có thể sử dụng các cấu trúc dữ liệu được trình bày trong chương này trong tình huống này. Để làm điều này, chúng ta xây dựng một **mảng hiệu** (difference array) mà các giá trị của nó cho biết sự khác biệt giữa các giá trị liên tiếp trong mảng ban đầu. Do đó, mảng ban đầu là mảng tổng tiền tố của mảng hiệu. Ví dụ, hãy xem xét mảng sau:

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

Mảng hiệu cho 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 chúng 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 chúng ta muốn tăng các giá trị của mảng ban đầu giữa các vị trí 1 và 4 lên 5, chỉ cần tăng giá trị của mảng hiệu tại vị trí 1 lên 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 $[a, b]$ lên x , chúng ta tăng giá trị tại vị trí a lên x và giảm giá trị tại vị trí $b + 1$ đi x . Do đó, chỉ cần cập nhật các giá trị đơn lẻ và xử lý các truy vấn tổng, vì vậy chúng ta có thể sử dụng một cây chỉ số nhị phân hoặc một cây đoạn.

Một bài toán khó hơn là hỗ trợ cả truy vấn đoạn và cập nhật đoạn. Trong Chương 28, chúng ta sẽ thấy rằng ngay cả điều này cũng có thể thực hiện được.


```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Nếu một số lớn hơn giới hạn trên của biểu diễn bit, số đó sẽ bị tràn. Trong một biểu diễn có dấu, số tiếp theo sau $2^{n-1} - 1$ là -2^{n-1} , và trong một biểu diễn không dấu, số tiếp theo sau $2^n - 1$ là 0. Ví dụ, hãy xem 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 trữ trong một biến `int`, vì vậy số tiếp theo sau $2^{31} - 1$ là -2^{31} .

10.2 Các phép toán bit

Phép toán 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, $22 \& 26 = 18$, bởi vì

$$\begin{array}{r} 10110 \quad (22) \\ \& \quad 11010 \quad (26) \\ \hline = \quad 10010 \quad (18) \end{array}$$

Sử dụng phép toán **and**, chúng ta có thể kiểm tra xem một số x có phải là số 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 chính xác khi $x \& (2^k - 1) = 0$.

Phép toán 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, $22 \mid 26 = 30$, bởi vì

$$\begin{array}{r} 10110 \quad (22) \\ \mid \quad 11010 \quad (26) \\ \hline = \quad 11110 \quad (30) \end{array}$$

Phép toán Xor

Phép toán **xor** $x \wedge y$ tạo ra một số có các bit một ở những vị trí mà chính xác một trong x và y có bit một. Ví dụ, $22 \wedge 26 = 12$, bởi vì

$$\begin{array}{r} 10110 \quad (22) \\ \wedge \quad 11010 \quad (26) \\ \hline = \quad 01100 \quad (12) \end{array}$$

Phép toán Not

Phép toán **not** $\sim x$ tạo ra một số trong đó tất cả các bit của x đã được đảo ngược. Công thức $\sim x = -x - 1$ đúng, ví dụ, $\sim 29 = -30$.

Kết quả của phép toán not ở cấp độ bit phụ thuộc vào độ dài của biểu diễn bit, bởi vì phép toán đảo ngược tất cả các bit. Ví dụ, nếu các số là các số int 32 bit, kết quả như sau:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000000011101 \\ \sim x & = & -30 \quad 1111111111111111111111111111100010 \end{array}$$

Phép dịch bit

Phép dịch bit trái $x \ll k$ nối thêm k bit không vào số, và phép dịch bit phải $x \gg k$ loại bỏ k bit cuối cùng khỏi số. Ví dụ, $14 \ll 2 = 56$, bởi vì 14 và 56 tương ứng với 1110 và 111000. Tương tự, $49 \gg 3 = 6$, bởi 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 làm tròn xuống thành một số nguyên.

Ứng dụng

Một số có dạng $1 < k$ có một bit một ở vị trí k và tất cả các bit khác là không, vì vậy chúng ta có thể sử dụng các số như vậy để truy cập các bit đơn lẻ của các số. Cụ thể, bit thứ k của một số là một chính xác khi $x \& (1 < k)$ khác không. Đoạn mã sau in ra 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 các bit đơn lẻ của các số bằng cách sử dụng các ý tưởng tương tự. Ví dụ, công thức $x \mid (1 \ll k)$ 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. Cũng lưu ý rằng một số dương x là một lũy thừa của hai chính xác 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 các số không ở đầu số
- `__builtin_ctz(x)`: số lượng các số không ở cuối số
- `__builtin_popcount(x)`: số lượng các số một trong số
- `builtin_parity(x)`: tính chẵn lẻ (chẵn hay lẻ) của số lượng các số một

Các hàm có thể được sử dụng như sau:

```
int x = 5328; // 00000000000000000001010011010000
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 các số int, cũng có các phiên bản long long của các hàm có sẵn với hậu tố ll.

10.3 Biểu diễn tập hợp

Mọi tập hợp con của một tập hợp $\{0, 1, 2, \dots, n-1\}$ có thể được biểu diễn dưới dạng một số nguyên n bit mà các bit một của nó chỉ ra những phần tử nào thuộc về tập hợp con đó. Đây là một cách hiệu quả để biểu diễn các tập hợp, bởi vì mỗi phần tử chỉ yêu cầu một bit bộ nhớ, và các phép toán tập hợp có thể được triển khai dưới dạng các phép toán bit.

Ví dụ, vì int là một kiểu 32 bit, một số int có thể biểu diễn bất kỳ tập hợp con nào của tập hợp $\{0, 1, 2, \dots, 31\}$. Biểu diễn bit của tập hợp $\{1, 3, 4, 8\}$ là

000000000000000000000000100011010,

tương ứng với số $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Triển khai tập hợp

Đoạn mã sau khai báo một biến int x có thể chứa một tập hợp 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 ra 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
```

Sau đó, đoạn mã sau in ra 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
```

Các phép toán tập hợp

Các phép toán tập hợp có thể được triển khai như sau dưới dạng các phép toán bit:

	cú pháp tập hợp	cú pháp bit
giao	$a \cap b$	$a \& b$
hợp	$a \cup b$	$a b$
phần bù	\bar{a}	$\sim a$
hiệu	$a \setminus b$	$a \& (\sim b)$

Ví dụ, đoạn mã sau đầu tiên xây dựng các tập hợp $x = \{1, 3, 4, 8\}$ và $y = \{3, 6, 8, 9\}$, và sau đó xây dựng tập hợp $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```
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
```

Lặp qua các tập hợp con

Đoạn mã sau duyệt qua các tập hợp con của $\{0, 1, \dots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // xử lý tập hợp con b
}
```

Đoạn mã sau duyệt qua các tập hợp con có đúng k phần tử:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // xử lý tập hợp con b
    }
}
```

Đoạn mã sau duyệt qua các tập hợp con của một tập hợp x :

```
int b = 0;
do {
    // xử lý tập hợp con b
} while (b=(b-x)&x);
```

10.4 Tối ưu hóa bit

Nhiều thuật toán có thể được tối ưu hóa bằng cách sử dụng các phép toán bit. Các tối ưu hóa như vậy không làm thay đổi độ phức tạp thời gian của thuật toán, nhưng chúng có thể có tác động lớn đến thời gian chạy thực tế của mã. Trong phần này, chúng ta thảo luận về các ví dụ về các tình huống như vậy.

Khoảng cách Hamming

Khoảng cách Hamming (Hamming distance) $\text{hamming}(a, b)$ giữa hai chuỗi a và b có cùng độ dài là số vị trí mà các chuỗi khác nhau. Ví dụ,

$$\text{hamming}(01101, 11001) = 2.$$

Hãy xem xét bài toán sau: Cho một danh sách n chuỗi bit, mỗi chuỗi có độ dài k , tính khoảng cách Hamming nhỏ nhất giữa hai chuỗi trong danh sách. Ví dụ, câu trả lời cho $[00111, 01101, 11110]$ là 2, bởi vì

- $\text{hamming}(00111, 01101) = 2$,

- `hamming(00111, 11110) = 3`, và
- `hamming(01101, 11110) = 3`.

Một cách đơn giản để giải quyết bài toán là duyệt qua tất cả các cặp chuỗi và tính khoảng cách Hamming của chúng, mang lại một thuật toán thời gian $O(n^2k)$. Hàm sau có thể được sử dụng 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, chúng ta có thể tối ưu hóa mã bằng cách lưu trữ các chuỗi bit dưới dạng số nguyên và tính khoảng cách Hamming bằng các phép toán bit. Cụ thể, nếu $k \leq 32$, chúng ta chỉ cần lưu trữ các chuỗi dưới dạng các giá trị `int` và sử dụng hàm sau để tính khoảng cách:

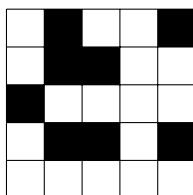
```
int hamming(int a, int b) {
    return __builtin_popcount(a ^ b);
}
```

Trong hàm trên, phép toán xor xây dựng một chuỗi 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`.

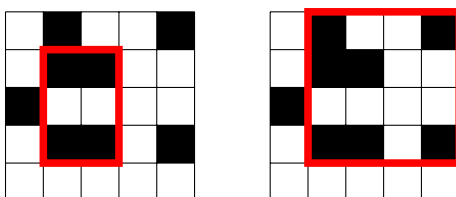
Để so sánh các cách triển khai, chúng tôi đã tạo một danh sách 10000 chuỗi bit ngẫu nhiên có độ dài 30. Sử dụng cách tiếp cận đầu tiên, việc tìm kiếm mất 13.5 giây, và sau khi tối ưu hóa bit, nó chỉ mất 0.5 giây. Do đó, mã được tối ưu hóa bit nhanh hơn gần 30 lần so với mã ban đầu.

Đếm lưới con

Một ví dụ khác, hãy xem xét bài toán sau: Cho một lưới $n \times n$ có mỗi ô là màu đen (1) hoặc màu trắng (0), tính số lượng lưới con mà tất cả các góc của nó đều là màu đen. Ví dụ, lưới



chứa hai lưới con như vậy:



Có một thuật toán thời gian $O(n^3)$ để giải quyết bài toán: duyệt qua tất cả các cặp hàng $O(n^2)$ và cho mỗi cặp (a, b) tính số cột chứa một ô vuông màu 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++;
}
```

Sau đó, những cột đó chiếm $\text{count}(\text{count} - 1)/2$ lưới con có các góc màu đen, bởi vì chúng ta có thể chọn bất kỳ hai trong số chúng để tạo thành một lưới con.

Để tối ưu hóa thuật toán này, chúng ta chia lưới thành các khối cột sao cho mỗi khối bao gồm N cột liên tiếp. Sau đó, mỗi hàng được lưu trữ dưới dạng một danh sách các số N bit mô tả màu sắc của các ô vuông. Bây giờ chúng ta có thể xử lý N cột cùng một lúc sử dụng các phép toán bit. Trong đoạn mã sau, $\text{color}[y][k]$ biểu diễn một khối N màu dưới dạng các bit.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

Thuật toán kết quả hoạt động trong thời gian $O(n^3/N)$.

Chúng tôi đã tạo một lưới ngẫu nhiên có kích thước 2500×2500 và so sánh cách triển khai ban đầu và tối ưu hóa bit. Trong khi mã ban đầu mất 29.6 giây, phiên bản tối ưu hóa 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 thuận tiện để triển khai các thuật toán quy hoạch động mà các trạng thái của chúng chứa các tập hợp con của các phần tử, bởi vì các trạng thái như vậy có thể được lưu trữ dưới dạng các số nguyên. Tiếp theo chúng ta thảo luận về các ví dụ về việc kết hợp các phép toán bit và quy hoạch động.

Lựa chọn tối ưu

Ví dụ đầu tiên, hãy xem xét bài toán sau: Chúng ta được cho giá của k sản phẩm trong n ngày, và chúng ta muốn mua mỗi sản phẩm đúng một lần. Tuy nhiên, chúng ta chỉ được phép mua nhiều nhất một sản phẩm trong một ngày. Tổng giá tối thiểu là bao nhiêu? Ví dụ, hãy xem xét kịch bản 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 kịch bản này, tổng giá tối thiểu 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 kịch bản trên $\text{price}[2][3] = 7$. Sau đó, gọi $\text{total}(S, d)$ là tổng giá tối thiểu để mua một tập hợp con S các sản phẩm trước ngày d . Sử dụng hàm này, lời giải cho bài toán là $\text{total}(\{0 \dots k-1\}, n-1)$.

Đầu tiên, $\text{total}(\emptyset, d) = 0$, bởi vì không mất gì để mua một tập hợp rỗng, và $\text{total}(\{x\}, 0) = \text{price}[x][0]$, bởi vì có một cách để mua một sản phẩm vào ngày đầu tiên. Sau đó, có thể sử dụng công thức đệ quy 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 chúng ta hoặc không mua sản phẩm nào vào ngày d hoặc mua một sản phẩm x thuộc S . Trong trường hợp thứ hai, chúng ta loại bỏ x khỏi S và cộng giá của x vào tổng giá.

Bước tiếp theo là tính các giá trị của hàm sử dụng quy hoạch động. Để lưu trữ các giá trị hàm, chúng ta khai báo một mảng

```
int total[1<<K][N];
```

trong đó K và N là các hằng số đủ lớn. Chiều thứ nhất của mảng tương ứng với một biểu diễn bit của một tập hợp con.

Đầu tiên, 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 đệ quy 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 hợp con

Sử dụng quy hoạch động, thường có thể thay đổi một vòng lặp qua các hoán vị thành một vòng lặp qua các tập hợp con¹. Lợi ích của việc này là $n!$, số lượng các hoán vị, lớn hơn nhiều so với 2^n , số lượng các tập hợp 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 , chúng ta có thể duyệt qua các tập hợp con một cách hiệu quả nhưng không thể duyệt qua các hoán vị.

¹Kỹ thuật này được giới thiệu vào năm 1962 bởi M. Held và R. M. Karp [34].

Ví dụ, hãy xem xét bài toán sau: Có một thang máy với trọng lượng tối đa là x , và n người có trọng lượng đã biết muốn đi từ tầng trệt lên tầng trên cùng. Số chuyến đi tối thiểu cần thiết là bao nhiêu nếu mọi người vào thang máy theo một thứ tự tối ưu?

Ví dụ, giả sử rằng $x = 10$, $n = 5$ và các trọng lượng như sau:

người	trọng lượng
0	2
1	3
2	3
3	5
4	6

Trong trường hợp này, số chuyến đi tối thiểu là 2. Một thứ tự tối ưu là $\{0, 2, 3, 1, 4\}$, chia mọi người thành hai chuyến đi: đầu tiên là $\{0, 2, 3\}$ (tổng trọng lượng 10), và sau đó là $\{1, 4\}$ (tổng trọng lượng 9).

Bài toán có thể được giải quyết dễ dàng trong thời gian $O(n!n)$ bằng cách thử tất cả các hoán vị có thể có của n người. Tuy nhiên, chúng ta có thể sử dụng quy hoạch động để có được một thuật toán hiệu quả hơn với thời gian $O(2^n n)$. Ý tưởng là tính toán cho mỗi tập hợp con của mọi người hai giá trị: số chuyến đi tối thiểu cần thiết và trọng lượng tối thiểu của những người đi trong nhóm cuối cùng.

Gọi $\text{weight}[p]$ là trọng lượng của người p . Chúng ta định nghĩa hai hàm: $\text{rides}(S)$ là số chuyến đi tối thiểu cho một tập hợp con S , và $\text{last}(S)$ là trọng lượng tối thiểu của chuyến đi cuối cùng. Ví dụ, trong kịch bản trên

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{và} \quad \text{last}(\{1, 3, 4\}) = 5,$$

bởi vì các chuyến đi tối ưu là $\{1, 4\}$ và $\{3\}$, và chuyến đi thứ hai có trọng lượng 5. Tất nhiên, mục tiêu cuối cùng của chúng ta là tính toán giá trị của $\text{rides}(\{0 \dots n-1\})$.

Chúng ta có thể tính các giá trị của các hàm một cách đệ quy và sau đó áp dụng quy hoạch động. Ý tưởng là duyệt qua tất cả mọi người thuộc S và chọn một cách tối ưu người cuối cùng p vào thang máy. Mỗi lựa chọn như vậy tạo ra một bài toán con cho một tập hợp con nhỏ hơn của mọi người. Nếu $\text{last}(S \setminus p) + \text{weight}[p] \leq x$, chúng ta có thể thêm p vào chuyến đi cuối cùng. Ngược lại, chúng ta phải dành một chuyến đi mới ban đầu chỉ chứa p .

Để triển khai quy hoạch động, chúng ta khai báo một mảng

```
pair<int,int> best[1<<N];
```

chứa cho mỗi tập hợp con S một cặp $(\text{rides}(S), \text{last}(S))$. Chúng ta đặt giá trị cho một nhóm rỗng như sau:

```
best[0] = {1,0};
```

Sau đó, chúng ta có thể điền vào mảng như sau:

```
for (int s = 1; s < (1<<n); s++) {
    // giá trị ban đầu: cần n+1 chuyến đi
    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) {
                // thêm p vào một chuyến đi đã có
```

```

        option.second += weight[p];
    } else {
        // danh mot chuyen di moi cho 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 hợp con S_1 và S_2 nào sao cho $S_1 \subset S_2$, chúng ta xử lý S_1 trước S_2 . Do đó, các giá trị quy hoạch động được tính toán theo thứ tự đúng.

Đếm tập hợp con

Bài toán cuối cùng của chúng ta trong chương này như sau: Cho $X = \{0 \dots n - 1\}$, và mỗi tập hợp con $S \subset X$ được gán một số nguyên $\text{value}[S]$. Nhiệm vụ của chúng ta là tính toán cho mỗi S

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

tức là, tổng các giá trị của các tập hợp con của S .

Ví dụ, giả sử rằng $n = 3$ 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 hợp con, một giải pháp khả thi là duyệt qua tất cả các cặp tập hợp con trong thời gian $O(2^{2n})$. Tuy nhiên, bằng cách sử dụng quy hoạch động, chúng ta có thể giải quyết bài toán trong thời gian $O(2^n n)$. Ý tưởng là tập trung vào các tổng mà các phần tử có thể bị loại bỏ khỏi S bị hạn chế.

Gọi $\text{partial}(S, k)$ là tổng các giá trị của các tập hợp con của S với hạn chế rằng chỉ có các phần tử $0 \dots k$ mới có thể bị loại bỏ khỏi S . Ví dụ,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

bởi vì chúng ta chỉ có thể loại bỏ các phần tử $0 \dots 1$. Chúng ta có thể tính các giá trị của sum bằng cách sử dụng các giá trị của partial , bởi vì

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Các trường hợp cơ sở cho hàm là

$$\text{partial}(S, -1) = \text{value}[S],$$

bởi vì trong trường hợp này không có phần tử nào có thể bị loại bỏ khỏi S . Sau đó, trong trường hợp tổng quát, chúng ta có thể sử dụng công thức đệ quy 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 chúng ta tập trung vào phần tử k . Nếu $k \in S$, chúng ta có hai lựa chọn: chúng ta có thể giữ k trong S hoặc loại bỏ nó khỏi S .

Có một cách đặc biệt thông minh để triển khai việc tính toán các tổng. Chúng ta có thể khai báo một mảng

```
int sum[1<<N];
```

sẽ chứa tổng của mỗi tập hợp con. Mảng được khởi tạo như sau:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Sau đó, chúng ta có thể điền vào 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 $\text{partial}(S, k)$ cho $k = 0 \dots n-1$ vào mảng `sum`. Vì $\text{partial}(S, k)$ luôn dựa trên $\text{partial}(S, k-1)$, chúng ta có thể tái sử dụng mảng `sum`, mang lại một cách triển khai rất hiệu quả.

Phần II

Graph algorithms

Chương 11

Những điều cơ bản về đồ thị

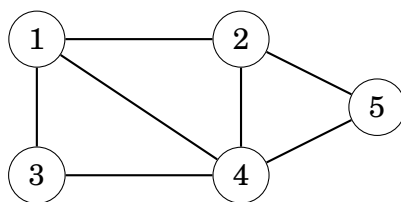
Nhiều bài toán lập trình có thể được giải quyết bằng cách mô hình hóa bài toán thành một bài toán đồ thị và sử dụng một thuật toán đồ thị thích hợp. Một ví dụ điển hình của đồ thị là một mạng lưới đường và thành phố trong một quốc gia. Tuy nhiên, đôi khi, đồ thị bị ẩn trong bài toán và có thể khó phát hiện ra nó.

Phần này của cuốn sách thảo luận về các thuật toán đồ thị, đặc biệt tập trung vào các chủ đề quan trọng trong lập trình thi đấu. Trong chương này, chúng ta sẽ tìm hiểu các khái niệm liên quan đến đồ thị, và nghiên cứu các cách khác nhau để biểu diễn đồ thị trong các thuật toán.

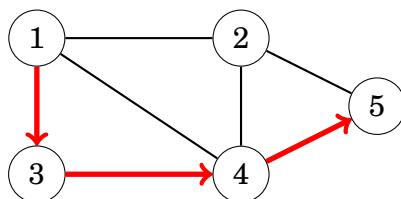
11.1 Thuật ngữ đồ thị

Một **đồ thị** bao gồm các **nút** và **cạnh**. Trong cuốn sách này, biến n biểu thị số lượng nút trong một đồ thị, và biến m biểu thị số lượng cạnh. Các nút được đánh số bằng các số nguyên $1, 2, \dots, n$.

Ví dụ, đồ thị sau bao gồm 5 nút và 7 cạnh:



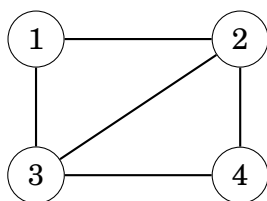
Một **đường đi** dẫn từ nút a đến nút b qua các cạnh của đồ thị. **Độ dài** của một đường đi là số lượng cạnh trong đó. Ví dụ, đồ thị trên chứa một đường đi $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ có độ dài 3 từ nút 1 đến nút 5:



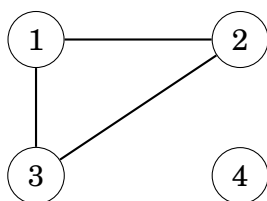
Một đường đi là một **chu trình** nếu nút đầu tiên và cuối cùng giống nhau. Ví dụ, đồ thị trên chứa một chu trình $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. Một đường đi là **đơn giản** nếu mỗi nút xuất hiện nhiều nhất một lần trong đường đi.

Tính liên thông

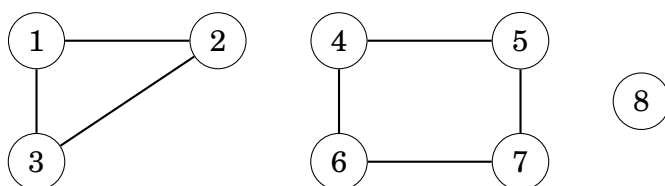
Một đồ thị là **liên thông** nếu có một đường đi giữa bất kỳ hai nút nào. Ví dụ, đồ thị sau là liên thông:



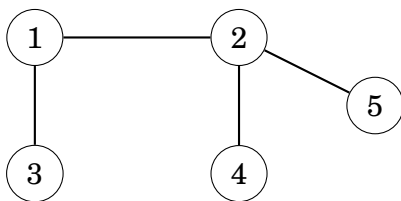
Đồ thị sau không liên thông, vì không thể đi từ nút 4 đến bất kỳ nút nào khác:



Các phần liên thông của một đồ thị được gọi là các **thành phần** của nó. Ví dụ, đồ thị sau chứa ba thành phần: {1, 2, 3}, {4, 5, 6, 7} và {8}.

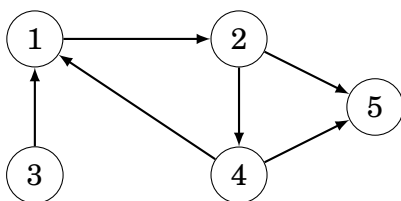


Một **cây** là một đồ thị liên thông bao gồm n nút và $n - 1$ cạnh. Có một đường đi duy nhất giữa bất kỳ hai nút nào của một cây. Ví dụ, đồ thị sau là một cây:



Hướng cạnh

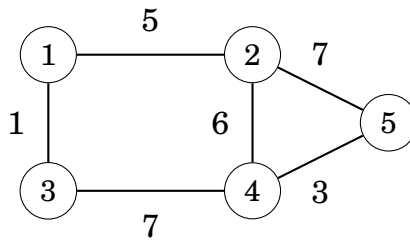
Một đồ thị là **có hướng** nếu các cạnh chỉ có thể được đi qua theo một hướng duy nhất. Ví dụ, đồ thị sau là có hướng:



Đồ thị trên chứa một đường đi $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ từ nút 3 đến nút 5, nhưng không có đường đi từ nút 5 đến nút 3.

Trọng số cạnh

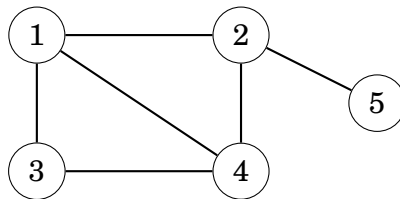
Trong một đồ thị **có trọng số**, mỗi cạnh được gán một **trọng số**. Các trọng số thường được hiểu là độ dài cạnh. Ví dụ, đồ thị sau là có trọng số:



Độ dài của một đường đi trong một đồ thị có trọng số là tổng của các trọng số 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 nút 1 đến nút 5.

Hàng xóm và bậc

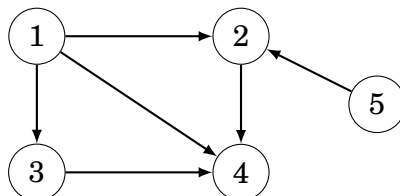
Hai nút là **hàng xóm** hoặc **liên kề** nếu có một cạnh giữa chúng. **Bậc** của một nút là số lượng hàng xóm của nó. Ví dụ, trong đồ thị sau, các hàng xóm của nút 2 là 1, 4 và 5, vì vậy bậc của nó là 3.



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 nút lên một. Vì lý do này, tổng các bậc luôn là số chẵn.

Một đồ thị là **chính quy** nếu bậc của mọi nút là một hằng số d . Một đồ thị là **đầy đủ** nếu bậc của mọi nút là $n - 1$, tức là, đồ thị chứa tất cả các cạnh có thể có giữa các nút.

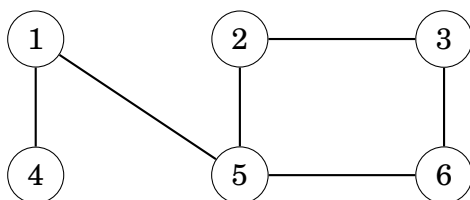
Trong một đồ thị có hướng, **bán bậc vào** của một nút là số cạnh kết thúc tại nút đó, và **bán bậc ra** của một nút là số cạnh bắt đầu từ nút đó. Ví dụ, trong đồ thị sau, bán bậc vào của nút 2 là 2, và bán bậc ra của nút 2 là 1.



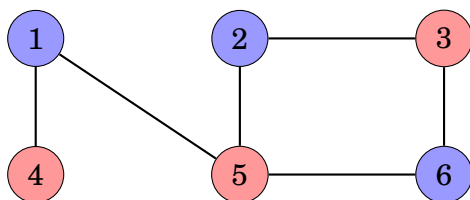
Tô màu

Trong một **tô màu** của một đồ thị, mỗi nút được gán một màu sao cho không có hai nút liên kề nào có cùng màu.

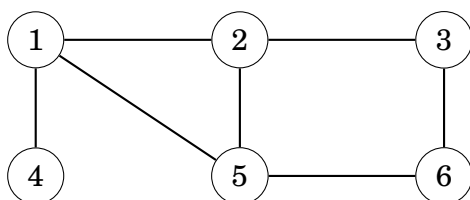
Một đồ thị là **hai phía** nếu có thể tô màu nó bằng hai màu. Hóa ra một đồ thị là hai phía chính xác khi nó không chứa một chu trình với số cạnh lẻ. Ví dụ, đồ thị



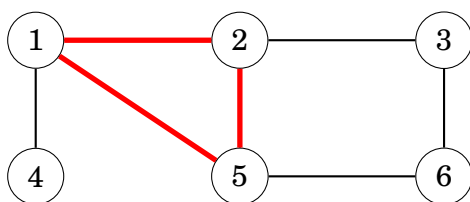
là hai phía, vì nó có thể được tô màu như sau:



Tuy nhiên, đồ thị

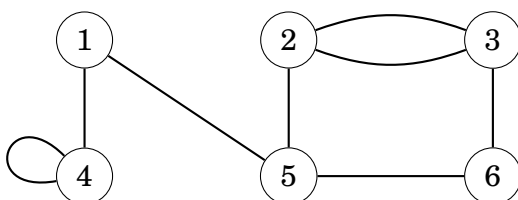


không phải là hai phía, vì không thể tô màu chu trình sau gồm ba nút bằng hai màu:



Tính đơn giản

Một đồ thị là **đơn** nếu không có cạnh nào bắt đầu và kết thúc tại cùng một nút, và không có nhiều cạnh giữa hai nút. Thường thì chúng ta giả định rằng các đồ thị là đơn. Ví dụ, đồ thị sau *không* đơn:



11.2 Biểu diễn đồ thị

Có một số cách để biểu diễn đồ thị trong các thuật toán. Việc lựa chọn một cấu trúc dữ liệu phụ thuộc vào kích thước của đồ thị và cách thuật toán xử lý nó. Tiếp theo, chúng ta sẽ xem xét ba cách biểu diễn phổ biến.

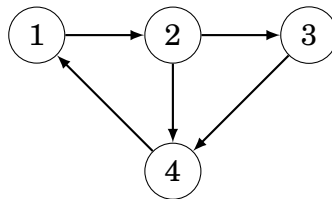
Biểu diễn danh sách kề

Trong biểu diễn danh sách kề, mỗi nút x trong đồ thị được gán một **danh sách kề** bao gồm các nút mà có một cạnh 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 thực hiện hiệu quả bằng cách sử dụng chúng.

Một cách thuận tiện để lưu trữ các danh sách kề là khai báo một mảng các vector như sau:

```
vector<int> adj[N];
```

Hằng số N được chọn sao cho tất cả các danh sách kề có thể được lưu trữ. Ví dụ, đồ thị



có thể được lưu trữ như sau:

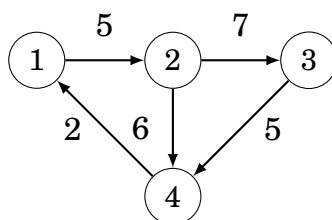
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Nếu đồ thị là vô hướng, nó có thể được lưu trữ theo cách tương tự, nhưng mỗi cạnh được thêm vào theo cả hai hướng.

Đối với một 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 nút a chứa cặp (b, w) luôn luôn khi có một cạnh từ nút a đến nút b với trọng số w . Ví dụ, đồ thị



có thể được lưu trữ 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 sử dụng danh sách kề là chúng ta có thể tìm thấy hiệu quả các nút mà chúng ta có thể di chuyển đến từ một nút đã cho thông qua một cạnh. Ví dụ, vòng lặp sau duyệt qua tất cả các nút mà chúng ta có thể di chuyển đến từ nút s :

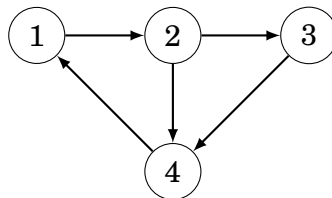
```
for (auto u : adj[s]) {
    // xử lý nút u
}
```

Biểu diễn ma trận kề

Một **ma trận kề** là một mảng hai chiều chỉ ra các cạnh mà đồ thị chứa. Chúng ta có thể kiểm tra hiệu quả từ một ma trận kề nếu có một cạnh giữa hai nút. Ma trận có thể được lưu trữ dưới dạng một mảng

```
int adj[N][N];
```

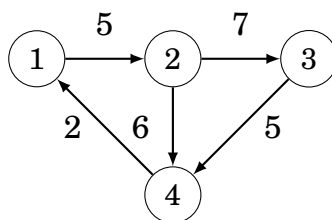
trong đó mỗi giá trị $adj[a][b]$ chỉ ra liệu đồ thị có chứa một cạnh từ nút a đến nút b hay không. Nếu cạnh được bao gồm trong đồ thị, thì $adj[a][b] = 1$, và ngược lại $adj[a][b] = 0$. Ví dụ, đồ thị



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 ma trận kề có thể được mở rộng sao cho ma trận chứa trọng số của cạnh nếu cạnh tồn tại. Sử dụng 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 ma trận kề là ma trận chứa n^2 phần tử, và thường thì hầu hết chúng là không. Vì lý do này, biểu diễn này không thể được sử dụng nếu đồ thị lớn.

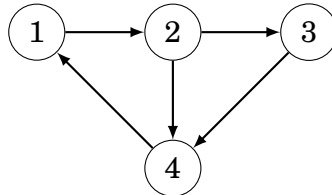
Biểu diễn danh sách cạnh

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 một đồ thị nếu thuật toán xử lý tất cả các cạnh của đồ thị và không cần tìm các cạnh bắt đầu tại một nút đã cho.

Danh sách cạnh có thể được lưu trữ trong một vector

```
vector<pair<int,int>> edges;
```

trong đó mỗi cặp (a, b) biểu thị rằng có một cạnh từ nút a đến nút b . Do đó, đồ thị



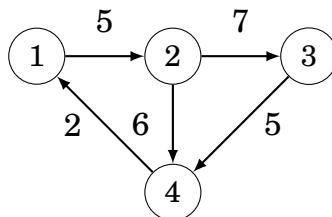
có thể được biểu diễn như sau:

```
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) , có nghĩa là có một cạnh từ nút a đến nút 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, hàm `make_tuple` phải được sử dụng thay vì dấu ngoặc nhọn (ví dụ, `make_tuple(1,2,5)` thay vì `{1,2,5}`).

Chương 12

Duyệt đồ thị

Chương này thảo luận về hai thuật toán đồ thị cơ bản: tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng. Cả hai thuật toán đều được cung cấp một nút bắt đầu trong đồ thị, và chúng duyệt qua tất cả các nút có thể đến được từ nút bắt đầu. Sự khác biệt giữa các thuật toán là thứ tự mà chúng duyệt qua các nút.

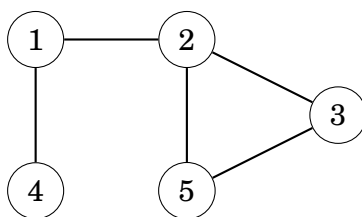
12.1 Tìm kiếm theo chiều sâu

Tìm kiếm theo chiều sâu (DFS) là một kỹ thuật duyệt đồ thị đơn giản. Thuật toán bắt đầu tại một nút xuất phát, và tiếp tục đến tất cả các nút khác có thể đến được từ nút xuất phát bằng cách sử dụng các cạnh của đồ thị.

Tìm kiếm theo chiều sâu luôn đi theo một đường đi duy nhất trong đồ thị miễn là nó tìm thấy các nút mới. Sau đó, nó quay trở lại các nút trước đó và bắt đầu khám phá các phần khác của đồ thị. Thuật toán theo dõi các nút đã được duyệt, để nó chỉ xử lý mỗi nút một lần.

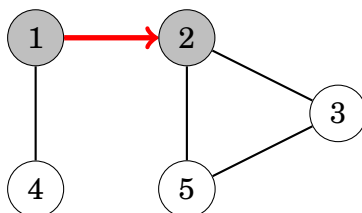
Ví dụ

Chúng ta hãy xem xét cách tìm kiếm theo chiều sâu xử lý đồ thị sau:

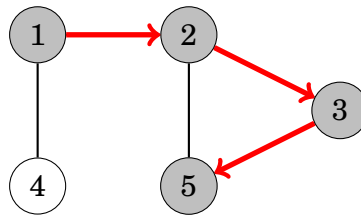


Chúng ta có thể bắt đầu tìm kiếm tại bất kỳ nút nào của đồ thị; bây giờ chúng ta sẽ bắt đầu tìm kiếm tại nút 1.

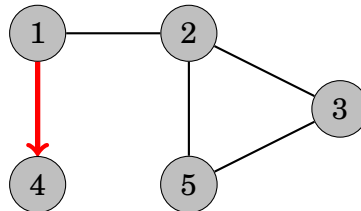
Tìm kiếm đầu tiên tiến đến nút 2:



Sau đó, các nút 3 và 5 sẽ được duyệt:



Các nút lân cận của nút 5 là 2 và 3, nhưng tìm kiếm đã duyệt qua cả hai, vì vậy đã đến lúc quay lại các nút trước đó. Các nút lân cận của nút 3 và 2 cũng đã được duyệt, vì vậy tiếp theo chúng ta di chuyển từ nút 1 đến nút 4:



Sau đó, tìm kiếm kết thúc vì nó đã duyệt tất cả các nút.

Độ phức tạp thời gian của tìm kiếm theo chiều sâu là $O(n + m)$ trong đó n là số nút và m là số cạnh, bởi vì thuật toán xử lý mỗi nút và cạnh một lần.

Cài đặt

Tìm kiếm theo chiều sâu có thể được cài đặt thuận tiện bằng cách sử dụng đệ quy. Hàm dfs sau đây bắt đầu một tìm kiếm theo chiều sâu tại một nút đã cho. Hàm giả định rằng đồ thị được lưu trữ dưới dạng danh sách kề trong một mảng

```
vector<int> adj[N];
```

và cũng duy trì một mảng

```
bool visited[N];
```

để theo dõi các nút đã được duyệt. Ban đầu, mỗi giá trị mảng là false, và khi tìm kiếm đến nút s , giá trị của `visited[s]` trở thành true. Hàm có thể được cài đặt như sau:

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // xử lý nút s
    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

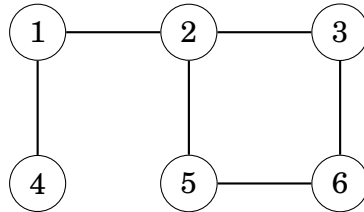
12.2 Tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng (BFS) duyệt các nút theo thứ tự tăng dần khoảng cách của chúng từ nút bắt đầu. Do đó, chúng ta có thể tính khoảng cách từ nút bắt đầu đến tất cả các nút khác bằng cách sử dụng tìm kiếm theo chiều rộng. Tuy nhiên, tìm kiếm theo chiều rộng khó cài đặt hơn so với tìm kiếm theo chiều sâu.

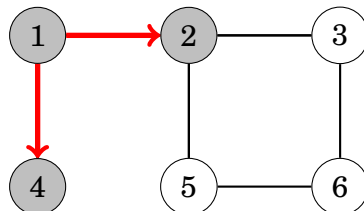
Tìm kiếm theo chiều rộng đi qua các nút theo từng cấp độ một. Đầu tiên, tìm kiếm khám phá các nút có khoảng cách từ nút bắt đầu là 1, sau đó là các nút có khoảng cách là 2, và cứ thế. Quá trình này tiếp tục cho đến khi tất cả các nút đã được duyệt.

Ví dụ

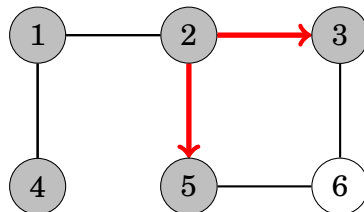
Chúng ta hãy xem xét cách tìm kiếm theo chiều rộng xử lý đồ thị sau:



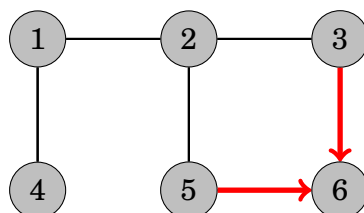
Giả sử rằng tìm kiếm bắt đầu tại nút 1. Đầu tiên, chúng ta xử lý tất cả các nút có thể đến được từ nút 1 bằng một cạnh duy nhất:



Sau đó, chúng ta tiếp tục đến các nút 3 và 5:



Cuối cùng, chúng ta duyệt nút 6:



Bây giờ chúng ta đã tính được khoảng cách từ nút bắt đầu đến tất cả các nút của đồ thị. Các khoảng cách như sau:

nút	khoảng cách
1	0
2	1
3	2
4	1
5	2
6	3

Giống như trong 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 nút và m là số cạnh.

Cài đặt

Tìm kiếm theo chiều rộng khó cài đặt hơn so với tìm kiếm theo chiều sâu, bởi vì thuật toán duyệt các nút ở các phần khác nhau của đồ thị. Một cài đặt điển hình dựa trên một hàng đợi chứa các nút. Tại mỗi bước, nút tiếp theo trong hàng đợi sẽ được xử lý.

Đoạn mã sau giả định rằng đồ thị được lưu trữ 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 nút cần được xử lý theo thứ tự tăng dần khoảng cách của chúng. Các nút mới luôn được thêm vào cuối hàng đợi, và nút ở đầu hàng đợi là nút tiếp theo cần được xử lý. Mảng `visited` chỉ ra những nút nào mà tìm kiếm đã duyệt qua, và mảng `distance` sẽ chứa khoảng cách từ nút bắt đầu đến tất cả các nút của đồ thị.

Tìm kiếm có thể được cài đặt như sau, bắt đầu tại nút x :

```
visited[x] = true;  
distance[x] = 0;  
q.push(x);  
while (!q.empty()) {  
    int s = q.front(); q.pop();  
    // xử lý nút s  
    for (auto u : adj[s]) {  
        if (visited[u]) continue;  
        visited[u] = true;  
        distance[u] = distance[s]+1;  
        q.push(u);  
    }  
}
```

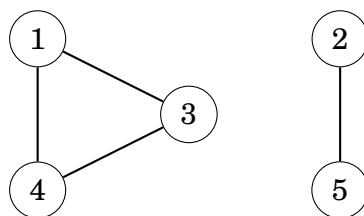
12.3 Ứng dụng

Sử dụng các thuật toán duyệt đồ thị, chúng ta có thể kiểm tra nhiều thuộc tính của đồ thị. Thông thường, cả tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng đều có thể được sử dụng, nhưng trong thực tế, tìm kiếm theo chiều sâu là một lựa chọn tốt hơn, vì nó dễ cài đặt hơn. Trong các ứng dụng sau, chúng ta sẽ giả định rằng đồ thị là vô hướng.

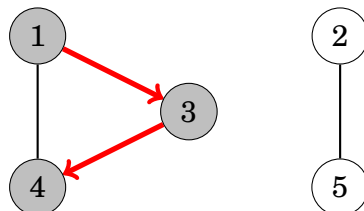
Kiểm tra tính liên thông

Một đồ thị là liên thông nếu có một đường đi giữa hai nút bất kỳ của đồ thị. Do đó, chúng ta có thể kiểm tra xem một đồ thị có liên thông hay không bằng cách bắt đầu tại một nút tùy ý và tìm hiểu xem chúng ta có thể đến được tất cả các nút khác hay không.

Ví dụ, trong đồ thị



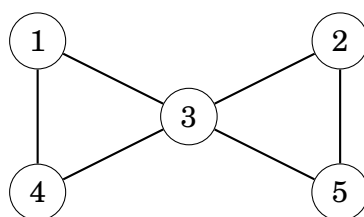
một tìm kiếm theo chiều sâu từ nút 1 duyệt các nút sau:



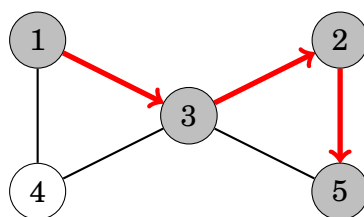
Vì tìm kiếm không duyệt qua tất cả các nút, chúng ta có thể kết luận rằng đồ thị không liên thông. Tương tự, chúng ta cũng có thể tìm thấy tất cả các thành phần liên thông của một đồ thị bằng cách lặp qua các nút và luôn bắt đầu một tìm kiếm theo chiều sâu mới nếu nút hiện tại chưa thuộc về bất kỳ thành phần nào.

Tìm chu trình

Một đồ thị chứa một chu trình nếu trong quá trình duyệt đồ thị, chúng ta tìm thấy một nút có hàng xóm (khác với nút trước đó trong đường đi hiện tại) đã được duyệt. Ví dụ, đồ thị



chứa hai chu trình và chúng ta có thể tìm thấy một trong số chúng như sau:



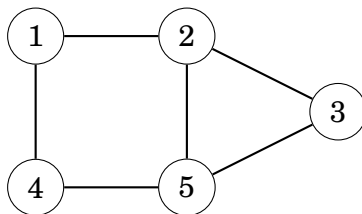
Sau khi di chuyển từ nút 2 đến nút 5, chúng ta nhận thấy rằng hàng xóm 3 của nút 5 đã được duyệt. Do đó, đồ thị chứa một chu trình đi qua nút 3, ví dụ, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

Một cách khác để tìm hiểu xem một đồ thị có chứa chu trình hay không là chỉ cần tính số lượng nút và cạnh trong mọi thành phần. Nếu một thành phần chứa c nút và không có chu trình, nó phải chứa chính xác $c - 1$ cạnh (vì vậy nó phải là một cây). Nếu có c hoặc nhiều cạnh hơn, thành phần đó chắc chắn chứa một chu trình.

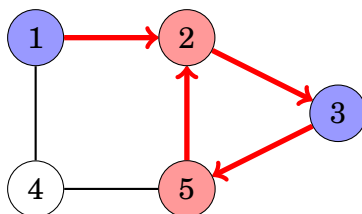
Kiểm tra tính hai phía

Một đồ thị là hai phía nếu các nút của nó có thể được tô màu bằng hai màu sao cho không có các nút kề nhau nào có cùng màu. Thật đáng ngạc nhiên là dễ dàng kiểm tra xem một đồ thị có phải là hai phía hay không bằng cách sử dụng các thuật toán duyệt đồ thị.

Ý tưởng là tô màu nút bắt đầu màu xanh, tất cả các hàng xóm của nó màu đỏ, tất cả các hàng xóm của chúng màu xanh, và cứ thế. Nếu tại một thời điểm nào đó của tìm kiếm, chúng ta nhận thấy rằng hai nút kề nhau có cùng màu, điều này có nghĩa là đồ thị không phải là hai phía. Ngược lại, đồ thị là hai phía và một cách tô màu đã được tìm thấy. Ví dụ, đồ thị



không phải là hai phía, bởi vì một tìm kiếm từ nút 1 tiến hành như sau:



Chúng ta nhận thấy rằng màu của cả hai nút 2 và 5 là màu đỏ, trong khi chúng là các nút kề nhau trong đồ thị. Do đó, đồ thị không phải là hai phía.

Thuật toán này luôn hoạt động, bởi vì khi chỉ có hai màu có sẵn, màu của nút bắt đầu trong một thành phần xác định màu của tất cả các nút khác trong thành phần đó. Không có sự khác biệt nào cho dù nút bắt đầu là màu đỏ hay màu xanh.

Lưu ý rằng trong trường hợp tổng quát, rất khó để tìm ra liệu các nút trong một đồ thị có thể được tô màu bằng k màu sao cho không có nút kề nhau nào có cùng màu hay không. Ngay cả khi $k = 3$, không có thuật toán hiệu quả nào được biết đến mà vấn đề là NP-khó.

Chương 13

Đường đi ngắn nhất

Tìm một đường đi ngắn nhất giữa hai nút của một đồ thị là một bài toán quan trọng có nhiều ứng dụng thực tế. Ví dụ, một bài toán tự nhiên liên quan đến mạng lưới đường bộ là tính toán độ dài ngắn nhất có thể của một tuyến đường giữa hai thành phố, cho biết độ dài của các con đường.

Trong một đồ thị không có trọng số, độ dài của một đường đi bằng số cạnh của nó, và chúng ta có thể đơn giản sử dụng tìm kiếm theo chiều rộng để tìm một đường đi ngắn nhất. Tuy nhiên, trong chương này chúng ta tập trung vào các đồ thị có trọng số nơi cần các thuật toán phức tạp hơn để tìm đường đi ngắn nhất.

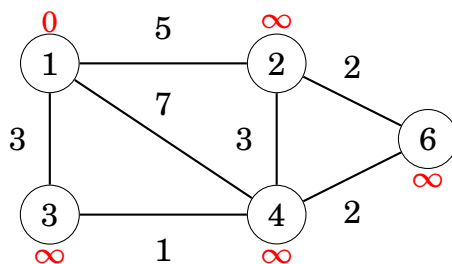
13.1 Thuật toán Bellman–Ford

Thuật toán Bellman–Ford¹ tìm các đường đi ngắn nhất từ một nút bắt đầu đến tất cả các nút của đồ thị. Thuật toán có thể xử lý tất cả các loại đồ thị, miễn là đồ thị không chứa một chu trình có độ dài âm. Nếu đồ thị chứa một chu trình âm, thuật toán có thể phát hiện ra điều này.

Thuật toán theo dõi các khoảng cách từ nút bắt đầu đến tất cả các nút của đồ thị. Ban đầu, khoảng cách đến nút bắt đầu là 0 và khoảng cách đến tất cả các nút khác là vô cùng. Thuật toán giảm các khoảng cách bằng cách tìm các cạnh rút ngắn các đường đi cho đến khi không thể giảm bất kỳ khoảng cách nào nữa.

Ví dụ

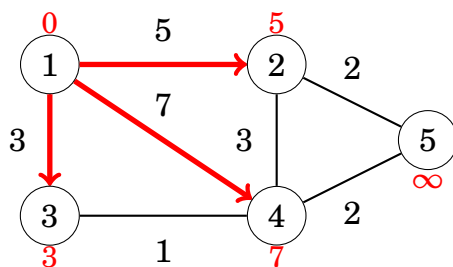
Chúng ta hãy xem xét cách thuật toán Bellman–Ford hoạt động trong đồ thị sau:



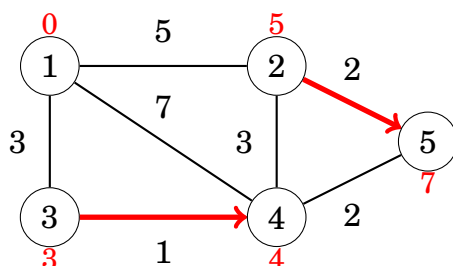
Mỗi nút của đồ thị được gán một khoảng cách. Ban đầu, khoảng cách đến nút bắt đầu là 0, và khoảng cách đến tất cả các nút khác là vô cùng.

¹Thuật toán được đặt theo tên của R. E. Bellman và L. R. Ford, những người đã công bố nó một cách độc lập vào năm 1958 và 1956, tương ứng [5, 24].

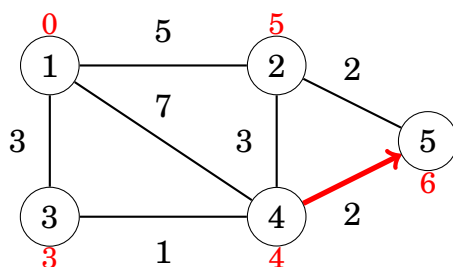
Thuật toán tìm kiếm các cạnh làm giảm khoảng cách. Đầu tiên, tất cả các cạnh từ nút 1 làm giảm khoảng cách:



Sau đó, các cạnh $2 \rightarrow 5$ và $3 \rightarrow 4$ làm giảm khoảng cách:

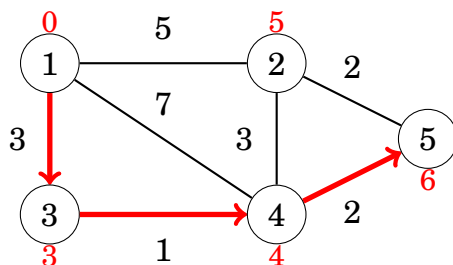


Cuối cùng, có một thay đổi nữa:



Sau đó, không có cạnh nào có thể làm giảm bất kỳ khoảng cách nào. Điều này có nghĩa là các khoảng cách là cuối cùng, và chúng ta đã thành công tính toán các khoảng cách ngắn nhất từ nút bắt đầu đến tất cả các nút của đồ thị.

Ví dụ, khoảng cách ngắn nhất 3 từ nút 1 đến nút 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 nút x đến tất cả các nút của đồ thị. Mã giả định rằng đồ thị được lưu trữ dưới dạng một danh sách cạnh `edges` bao gồm các bộ ba có dạng (a, b, w) , nghĩa là có một cạnh từ nút a đến nút b với trọng số w .

Thuật toán bao gồm $n - 1$ vòng, và ở mỗi vòng, thuật toán đi qua tất cả các cạnh của đồ thị và cố gắng giảm các khoảng cách. Thuật toán xây dựng một mảng `distance` sẽ chứa các khoảng cách từ x đến tất cả các nút của đồ thị. Hằng số INF biểu thị một khoảng cách vô cùng.

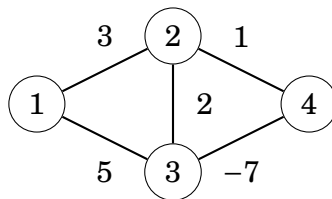
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Độ phức tạp thời gian của thuật toán là $O(nm)$, bởi vì thuật toán bao gồm $n - 1$ vòng và lặp qua tất cả m cạnh trong một vòng. Nếu không có chu trình âm trong đồ thị, tất cả các khoảng cách là cuối cùng sau $n - 1$ vòng, bởi vì mỗi đường đi ngắn nhất có thể chứa nhiều nhất $n - 1$ cạnh.

Trong thực tế, các khoảng cách cuối cùng thường có thể được tìm thấy nhanh hơn so với $n - 1$ vòng. Do đó, một cách khả thi để làm cho thuật toán hiệu quả hơn là dừng thuật toán nếu không có khoảng cách nào có thể được giảm trong một vòng.

Chu trình âm

Thuật toán Bellman–Ford cũng có thể được sử dụng để kiểm tra xem đồ thị có chứa một chu trình có độ dài âm hay không. Ví dụ, đồ thị



chứa một chu trình âm $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ với độ dài -4 .

Nếu đồ thị chứa một chu trình âm, chúng ta có thể rút ngắn vô hạn lần bất kỳ đường đi nào chứa chu trình bằng cách lặp lại chu trình liên tục. Do đó, khái niệm về một đường đi ngắn nhất không có ý nghĩa trong tình huống này.

Một chu trình âm có thể được phát hiện bằng cách sử dụng 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 sử dụng để tìm kiếm một chu trình âm trong toàn bộ đồ thị bất kể nút bắt đầu.

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 ban đầu. Thuật toán SPFA không đi qua tất cả các cạnh trong mỗi vòng, thay vào đó, nó chọn các cạnh để kiểm tra một cách thông minh hơn.

Thuật toán duy trì một hàng đợi các nút có thể được sử dụng để giảm khoảng cách. Đầu tiên, thuật toán thêm nút bắt đầu x vào hàng đợi. Sau đó, thuật toán luôn xử lý nút

đầu tiên trong hàng đợi, và khi một cạnh $a \rightarrow b$ làm giảm khoảng cách, nút b được thêm vào hàng đợi.

Hiệu quả của thuật toán SPFA phụ thuộc vào cấu trúc của đồ thị: thuật toán thường hiệu quả, nhưng độ phức tạp thời gian trường hợp xấu nhất của nó vẫn là $O(nm)$ và có thể tạo ra các đầu vào làm cho thuật toán chậm như thuật toán Bellman–Ford ban đầu.

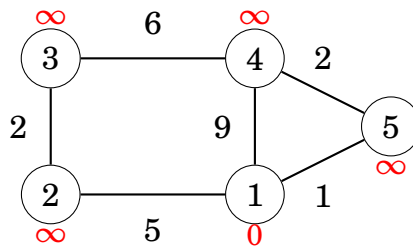
13.2 Thuật toán Dijkstra

Thuật toán Dijkstra² tìm các đường đi ngắn nhất từ nút bắt đầu đến tất cả các nút của đồ thị, giống như thuật toán Bellman–Ford. Lợi ích của thuật toán Dijkstra là nó hiệu quả hơn và có thể được sử dụng để xử lý các đồ thị lớn. Tuy nhiên, thuật toán yêu cầu không có cạnh trọng số âm trong đồ thị.

Giống như thuật toán Bellman–Ford, thuật toán Dijkstra duy trì khoảng cách đến các nút và giảm chúng trong quá trình tìm kiếm. Thuật toán Dijkstra hiệu quả, bởi vì nó chỉ xử lý mỗi cạnh trong đồ thị một lần, sử dụng thực tế rằng không có cạnh âm.

Ví dụ

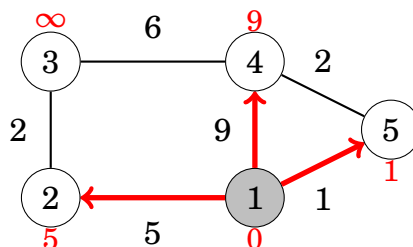
Chúng ta hãy xem xét cách thuật toán Dijkstra hoạt động trong đồ thị sau khi nút bắt đầu là nút 1:



Giống như trong thuật toán Bellman–Ford, ban đầu khoảng cách đến nút bắt đầu là 0 và khoảng cách đến tất cả các nút khác là vô cùng.

Tại mỗi bước, thuật toán Dijkstra chọn một nút chưa được xử lý và có khoảng cách nhỏ nhất có thể. Nút đầu tiên như vậy là nút 1 với khoảng cách 0.

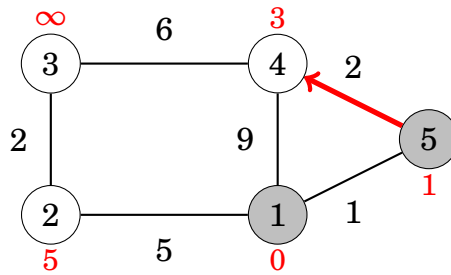
Khi một nút được chọn, thuật toán đi qua tất cả các cạnh bắt đầu tại nút đó và giảm khoảng cách bằng cách sử dụng chúng:



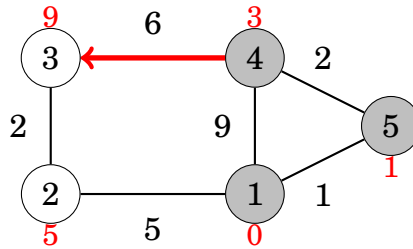
Trong trường hợp này, các cạnh từ nút 1 đã giảm khoảng cách của các nút 2, 4 và 5, có khoảng cách bây giờ là 5, 9 và 1.

Nút tiếp theo được xử lý là nút 5 với khoảng cách 1. Điều này làm giảm khoảng cách đến nút 4 từ 9 xuống 3:

²E. W. Dijkstra đã công bố thuật toán vào năm 1959 [14]; tuy nhiên, bài báo gốc của ông không đề cập đến cách cài đặt thuật toán một cách hiệu quả.

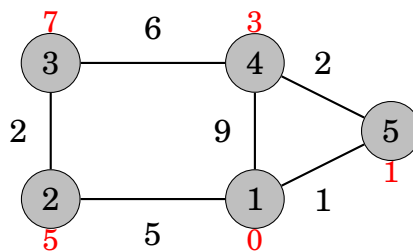


Sau đó, nút tiếp theo là nút 4, làm giảm khoảng cách đến nút 3 xuống 9:



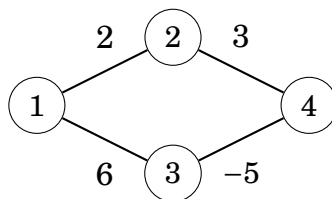
Một thuộc tính đáng chú ý trong thuật toán Dijkstra là bất cứ khi nào một nút được chọn, khoảng cách của nó là cuối cùng. Ví dụ, tại thời điểm này của thuật toán, các khoảng cách 0, 1 và 3 là các khoảng cách cuối cùng đến các nút 1, 5 và 4.

Sau đó, thuật toán xử lý hai nút còn lại, và các khoảng cách cuối cùng như sau:



Cạnh âm

Hiệu quả của thuật toán Dijkstra dựa trên thực tế là đồ thị không chứa các cạnh âm. Nếu có một cạnh âm, thuật toán có thể cho kết quả không chính xác. Ví dụ, hãy xem xét đồ thị sau:



Đường đi ngắn nhất từ nút 1 đến nút 4 là $1 \rightarrow 3 \rightarrow 4$ và độ dài của nó là 1. Tuy nhiên, thuật toán Dijkstra tìm thấy đường đi $1 \rightarrow 2 \rightarrow 4$ bằng cách đi theo các cạnh có trọng số nhỏ nhất. Thuật toán không tính đến việc trên đường đi kia, trọng số -5 bù lại cho trọng số lớn trước đó là 6.

Cài đặt

Cài đặt sau của thuật toán Dijkstra tính toán khoảng cách nhỏ nhất từ một nút x đến các nút khác của đồ thị. Đồ thị được lưu trữ dưới dạng danh sách kề sao cho $\text{adj}[a]$ chứa một cặp (b, w) luôn luôn khi có một cạnh từ nút a đến nút b với trọng số w .

Một cài đặt hiệu quả của thuật toán Dijkstra yêu cầu có thể tìm thấy hiệu quả nút có khoảng cách nhỏ nhất chưa được xử lý. Một cấu trúc dữ liệu thích hợp cho việc này là hàng đợi ưu tiên chứa các nút được sắp xếp theo khoảng cách của chúng. Sử dụng hàng đợi ưu tiên, nút tiếp theo cần xử lý có thể được lấy ra trong thời gian logarit.

Trong đoạn mã sau, hàng đợi ưu tiên q chứa các cặp có dạng $(-d, x)$, nghĩa là khoảng cách hiện tại đến nút x là d . Mảng `distance` chứa khoảng cách đến mỗi nút, và mảng `processed` chỉ ra liệu một nút đã được xử lý hay chưa. Ban đầu khoảng cách là 0 đến x và ∞ đến tất cả các nút khác.

```
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 đến các nút. Lý do cho điều này là phiên bản mặc định của hàng đợi ưu tiên C++ tìm các phần tử lớn nhất, trong khi chúng ta muốn tìm các phần tử nhỏ nhất. Bằng cách sử dụng các khoảng cách âm, chúng ta có thể sử dụng trực tiếp hàng đợi ưu tiên mặc định³. Cũng lưu ý rằng có thể có nhiều phiên bản của cùng một nút trong hàng đợi ưu tiên; tuy nhiên, chỉ có phiên bản có khoảng cách nhỏ nhất sẽ được xử lý.

Độ phức tạp thời gian của cài đặt trên là $O(n + m \log m)$, bởi vì thuật toán đi qua tất cả các nút của đồ thị và thêm cho mỗi cạnh 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 thay thế cho bài toán tìm đường đi ngắn nhất. Không giống như các thuật toán khác trong chương này, nó tìm tất cả các đường đi ngắn nhất giữa các nút trong một lần chạy duy nhất.

Thuật toán duy trì một mảng hai chiều chứa khoảng cách giữa các nút. Đầu tiên, khoảng cách chỉ được tính toán bằng cách sử dụng các cạnh trực tiếp giữa các nút, và sau

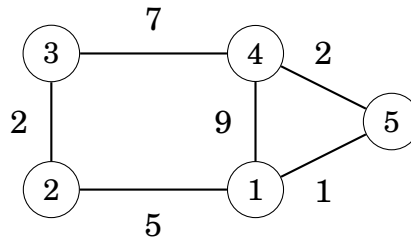
³Tất nhiên, chúng ta cũng có thể khai báo hàng đợi ưu tiên như trong Chương 4.5 và sử dụng các khoảng cách dương, nhưng việc cài đặt sẽ dài hơn một chút.

⁴Thuật toán được đặt theo tên của R. W. Floyd và S. Warshall những người đã công bố nó một cách độc lập vào năm 1962 [23, 70].

đó, thuật toán giảm khoảng cách bằng cách sử dụng các nút trung gian trong các đường đi.

Ví dụ

Chúng ta hãy xem xét cách thuật toán Floyd–Warshall hoạt động trong đồ thị sau:



Ban đầu, khoảng cách từ mỗi nút đến chính nó là 0, và khoảng cách giữa các nút a và b là x nếu có một cạnh giữa các nút a và b với trọng số x . Tất cả các khoảng cách khác là vô cùng.

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 bao gồm các vòng liên tiếp. Trên mỗi vòng, thuật toán chọn một nút mới có thể hoạt động như một nút trung gian trong các đường đi từ bây giờ, và khoảng cách được giảm bằng cách sử dụng nút này.

Ở vòng đầu tiên, nút 1 là nút trung gian mới. Có một đường đi mới giữa các nút 2 và 4 với độ dài 14, vì nút 1 kết nối chúng. Cũng có một đường đi mới giữa các nút 2 và 5 với 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, nút 2 là nút trung gian mới. Điều này tạo ra các đường đi mới giữa các nút 1 và 3 và giữa các nút 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

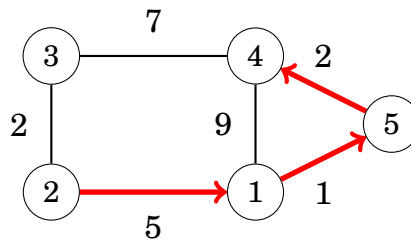
Ở vòng thứ ba, nút 3 là vòng trung gian mới. Có một đường đi mới giữa các nút 2 và 4:

	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 đến khi tất cả các nút đã được chỉ định làm nút trung gian. Sau khi thuật toán kết thúc, mảng chứa khoảng cách tối thiểu giữa bất kỳ hai nút nào:

	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 chúng ta biết rằng khoảng cách ngắn nhất giữa các nút 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 các nút a và b . Đầu tiên, thuật toán khởi tạo distance sử dụng 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 thấy 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 đi qua các nút của đồ thị.

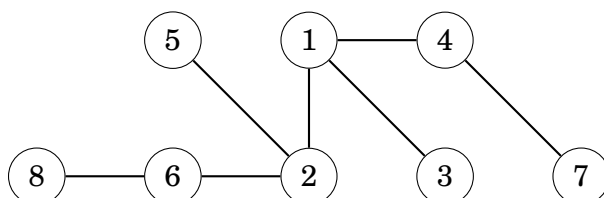
Vì việc cài đặt thuật toán Floyd–Warshall đơn giản, thuật toán có thể là một lựa chọn tốt ngay cả khi chỉ cần tìm một đường đi ngắn nhất duy nhất trong đồ thị. Tuy nhiên, thuật toán chỉ có thể được sử dụng khi đồ thị đủ nhỏ để độ phức tạp thời gian bậc ba đủ nhanh.

Chương 14

Các thuật toán trên cây

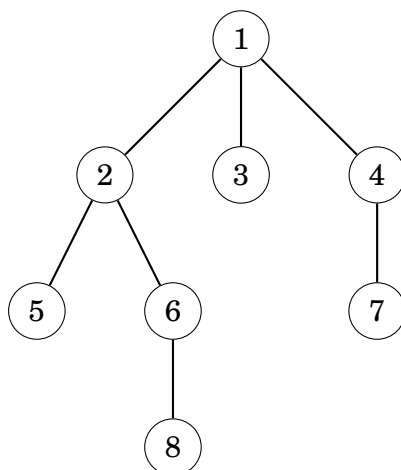
Một **cây** là một đồ thị liên thông, không có chu trình bao gồm n nút và $n - 1$ cạnh. Việc loại bỏ bất kỳ cạnh nào khỏi cây sẽ chia nó thành hai thành phần, và việc thêm bất kỳ cạnh nào vào cây sẽ tạo ra một chu trình. Hơn nữa, luôn có một đường đi duy nhất giữa bất kỳ hai nút nào của một cây.

Ví dụ, cây sau bao gồm 8 nút và 7 cạnh:



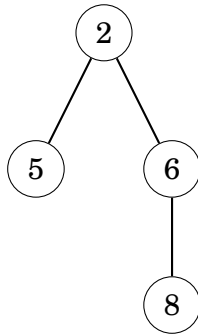
Các **lá** của một cây là các nút có bậc 1, tức là chỉ có một hàng xóm. Ví dụ, các lá của cây trên là các nút 3, 5, 7 và 8.

Trong một cây **có gốc**, một trong các nút được chỉ định làm **gốc** của cây, và tất cả các nút khác được đặt bên dưới gốc. Ví dụ, trong cây sau, nút 1 là nút gốc.



Trong một cây có gốc, các **con** của một nút là các hàng xóm dưới của nó, và **cha** của một nút là hàng xóm trên của nó. Mỗi nút có đúng một cha, ngoại trừ gốc không có cha. Ví dụ, trong cây trên, các con của nút 2 là các nút 5 và 6, và cha của nó là nút 1.

Cấu trúc của một cây có gốc là **đệ quy**: mỗi nút của cây hoạt động như gốc của một **cây con** chứa chính nút đó và tất cả các nút nằm trong các cây con của các con của nó. Ví dụ, trong cây trên, cây con của nút 2 bao gồm các nút 2, 5, 6 và 8:



14.1 Duyệt cây

Các thuật toán duyệt đồ thị tổng quát có thể được sử dụng để duyệt các nút của một cây. Tuy nhiên, việc duyệt một cây dễ cài đặt hơn so với một đồ thị tổng quát, bởi vì không có chu trình trong cây và không thể đến một nút từ nhiều hướng.

Cách điển hình để duyệt một cây là bắt đầu một tìm kiếm theo chiều sâu tại một nút tùy ý. Hàm đệ quy sau có thể được sử dụng:

```
void dfs(int s, int e) {
    // xử lý nút s
    for (auto u : adj[s]) {
        if (u != e) dfs(u, s);
    }
}
```

Hàm được cung cấp hai tham số: nút hiện tại s và nút trước đó e . Mục đích của tham số e là để đảm bảo rằng tìm kiếm chỉ di chuyển đến các nút chưa được thăm.

Lệnh gọi hàm sau bắt đầu tìm kiếm tại nút x :

```
dfs(x, 0);
```

Trong lệnh gọi đầu tiên $e = 0$, bởi vì không có nút trước đó, và được phép tiến đến bất kỳ hướng nào trong cây.

Quy hoạch động

Quy hoạch động có thể được sử dụng để tính toán một số thông tin trong quá trình duyệt cây. Sử dụng quy hoạch động, chúng ta có thể, ví dụ, tính toán trong thời gian $O(n)$ cho mỗi nút của một cây có gốc số lượng nút trong cây con của nó hoặc độ dài của đường đi dài nhất từ nút đó đến một lá.

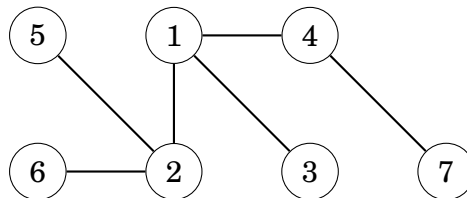
Ví dụ, chúng ta hãy tính toán cho mỗi nút s một giá trị $\text{count}[s]$: số lượng nút trong cây con của nó. Cây con chứa chính nút đó và tất cả các nút trong các cây con của các con của nó, vì vậy chúng ta có thể tính toán số lượng nút một cách đệ quy bằng cách sử dụng đoạn mã sau:

```
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];
    }
}
```

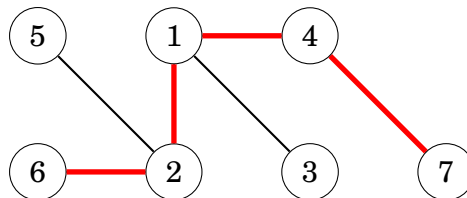
```
}  
}
```

14.2 Đường kính

Đường kính của một cây là độ dài tối đa của một đường đi giữa hai nút. Ví dụ, hãy xem 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 tối đa. Trong đường đi trên, chúng ta có thể thay thế nút 6 bằng nút 5 để có được một đường đi khác có độ dài 4.

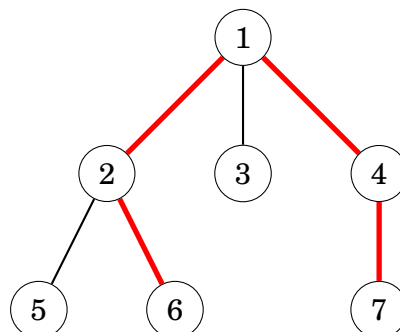
Tiếp theo chúng ta sẽ thảo luận về hai thuật toán thời gian $O(n)$ để tính đường kính của một cây. Thuật toán đầu tiên dựa trên quy hoạch động, và thuật toán thứ hai sử dụng hai lần tìm kiếm theo chiều sâu.

Thuật toán 1

Một cách tiếp cận chung cho nhiều bài toán trên cây là trước tiên gốc hóa cây một cách tùy ý. Sau đó, chúng ta có thể cố gắng giải quyết bài toán một cách riêng biệt cho mỗi cây con. Thuật toán đầu tiên của chúng ta để tính đường kính dựa trên ý tưởng này.

Một quan sát quan trọng là mọi đường đi trong một cây có gốc đều có một *điểm cao nhất*: nút cao nhất thuộc về đường đi. Do đó, chúng ta có thể tính toán cho mỗi nút độ dài của đường đi dài nhất có điểm cao nhất là nút đó. Một trong những đường đi đó tương ứng với đường kính của cây.

Ví dụ, trong cây sau, nút 1 là điểm cao nhất trên đường đi tương ứng với đường kính:



Chúng ta tính toán cho mỗi nút x hai giá trị:

- $\text{toLeaf}(x)$: độ dài tối đa của một đường đi từ x đến bất kỳ lá nào
- $\text{maxLength}(x)$: độ dài tối đa của một đường đi có điểm cao nhất là x

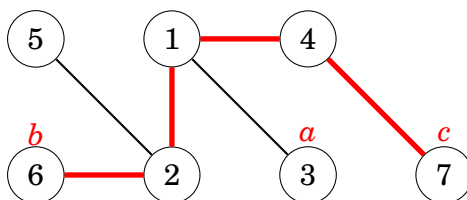
Ví dụ, trong cây trên, $\text{toLeaf}(1) = 2$, vì có một đường đi $1 \rightarrow 2 \rightarrow 6$, và $\text{maxLength}(1) = 4$, vì có một đường đi $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$. Trong trường hợp này, $\text{maxLength}(1)$ bằng với đường kính.

Quy hoạch động có thể được sử dụng để tính toán các giá trị trên cho tất cả các nút trong thời gian $O(n)$. Đầu tiên, để tính $\text{toLeaf}(x)$, chúng ta đi qua các con của x , chọn một con c có $\text{toLeaf}(c)$ lớn nhất và cộng một vào giá trị này. Sau đó, để tính $\text{maxLength}(x)$, chúng ta chọn hai con riêng biệt a và b sao cho tổng $\text{toLeaf}(a) + \text{toLeaf}(b)$ là lớn nhất và cộng hai vào tổng này.

Thuật toán 2

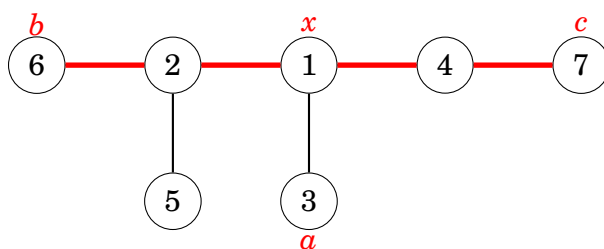
Một cách hiệu quả khác để tính đường kính của một cây là dựa trên hai lần tìm kiếm theo chiều sâu. Đầu tiên, chúng ta chọn một nút tùy ý a trong cây và tìm nút xa nhất b từ a . Sau đó, chúng ta tìm nút xa nhất c từ b . Đường kính của cây là khoảng cách giữa b và c .

Trong đồ thị sau, a , b và c có thể là:



Đây là một phương pháp thanh lịch, nhưng tại sao nó hoạt động?

Sẽ hữu ích nếu vẽ cây theo một cách khác để đường đi tương ứng với đường kính là nằm ngang, và tất cả các nút khác treo trên nó:

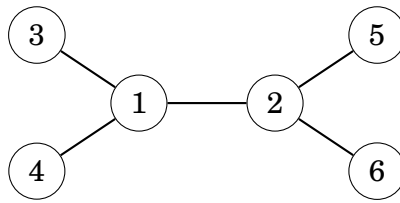


Nút x chỉ ra nơi mà đường đi từ nút a nối với đường đi tương ứng với đường kính. Nút xa nhất từ a là nút b , nút c hoặc một số nút khác ít nhất cũng xa như từ nút x . Do đó, nút này luôn là một lựa chọn hợp lệ cho một điểm cuối của một đường đi tương ứng với đường kính.

14.3 Tất cả các đường đi dài nhất

Bài toán tiếp theo của chúng ta là tính toán cho mọi nút trong cây độ dài tối đa của một đường đi bắt đầu tại nút đó. Điều này có thể được xem như một sự tổng quát hóa của bài toán đường kính cây, bởi vì lớn nhất trong số các độ dài đó bằng với đường kính của cây. Bài toán này cũng có thể được giải quyết trong thời gian $O(n)$.

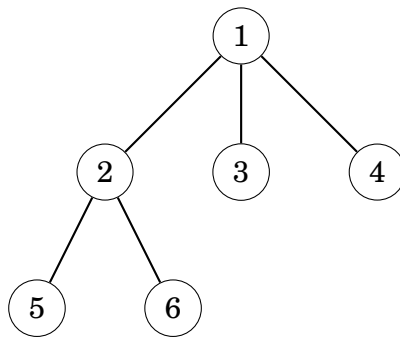
Ví dụ, hãy xem xét cây sau:



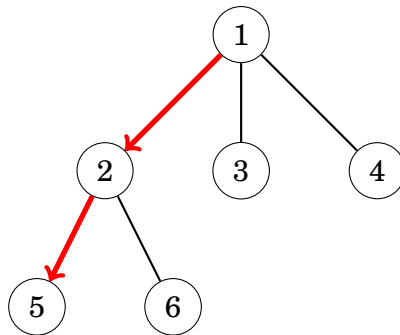
Gọi $\text{maxLength}(x)$ là độ dài tối đa của một đường đi bắt đầu tại nút x . Ví dụ, trong cây trên, $\text{maxLength}(4) = 3$, vì có một đường đi $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Đây là một bảng đầy đủ các giá trị:

nút x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Cũng trong bài toán này, một điểm khởi đầu tốt để giải quyết bài toán là gốc hóa cây một cách tùy ý:

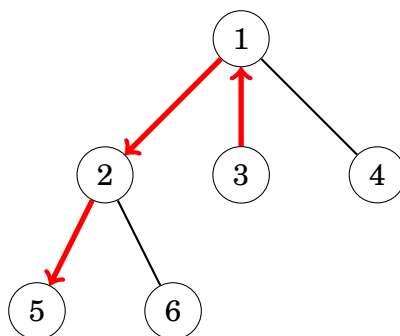


Phần đầu tiên của bài toán là tính toán cho mọi nút x độ dài tối đa của một đường đi đi qua một con của x . Ví dụ, đường đi dài nhất từ nút 1 đi qua con của nó là 2:

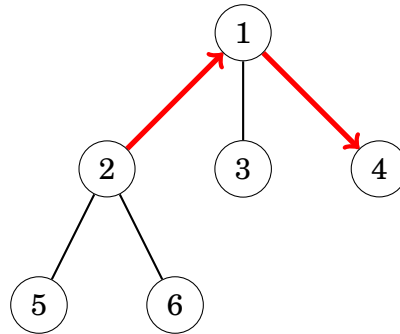


Phần này dễ giải quyết trong thời gian $O(n)$, vì chúng ta có thể sử dụng quy hoạch động như chúng ta đã làm trước đây.

Sau đó, phần thứ hai của bài toán là tính toán cho mọi nút x độ dài tối đa của một đường đi đi qua cha của nó là p . Ví dụ, đường đi dài nhất từ nút 3 đi qua cha của nó là 1:



Thoạt nhìn, có vẻ như chúng ta nên chọn đường đi dài nhất từ p . Tuy nhiên, điều này *không* phải lúc nào cũng hoạt động, bởi vì đường đi dài nhất từ p có thể đi qua x . Đây là một ví dụ về tình huống này:



Tuy nhiên, chúng ta có thể giải quyết phần thứ hai trong thời gian $O(n)$ bằng cách lưu trữ *hai* độ dài tối đa cho mỗi nút x :

- $\text{maxLength}_1(x)$: độ dài tối đa của một đường đi từ x
- $\text{maxLength}_2(x)$: độ dài tối đa của một đường đi từ x theo một hướng khác với đường đi đầu tiên

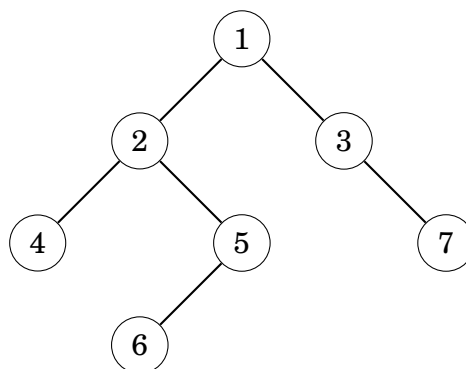
Ví dụ, trong đồ thị trên, $\text{maxLength}_1(1) = 2$ sử dụng đường đi $1 \rightarrow 2 \rightarrow 5$, và $\text{maxLength}_2(1) = 1$ sử dụng đường đi $1 \rightarrow 3$.

Cuối cùng, nếu đường đi tương ứng với $\text{maxLength}_1(p)$ đi qua x , chúng ta kết luận rằng độ dài tối đa là $\text{maxLength}_2(p) + 1$, và ngược lại, độ dài tối đa là $\text{maxLength}_1(p) + 1$.

14.4 Cây nhị phân

Một **cây nhị phân** là một cây có gốc trong đó mỗi nút có một cây con trái và một cây con phải. Có thể một cây con của một nút là rỗng. Do đó, mọi nút trong một cây nhị phân có không, một hoặc hai con.

Ví dụ, cây sau là một cây nhị phân:



Các nút 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:

- **tiền thứ tự**: đầu tiên xử lý gốc, sau đó duyệt cây con trái, sau đó duyệt cây con phải
- **trung thứ tự**: đầu tiên duyệt cây con trái, sau đó xử lý gốc, sau đó duyệt cây con phải
- **hậu thứ tự**: đầu tiên duyệt cây con trái, sau đó duyệt cây con phải, sau đó xử lý gốc

Đối với cây trên, các nút theo tiền thứ tự là $[1, 2, 4, 5, 6, 3, 7]$, theo trung thứ tự là $[4, 2, 6, 5, 1, 3, 7]$ và theo hậu thứ tự là $[4, 6, 5, 2, 7, 3, 1]$.

Nếu chúng ta biết thứ tự tiền thứ tự và trung thứ tự của một cây, chúng ta có thể tái tạo lại cấu trúc chính xác của cây. Ví dụ, cây trên là cây duy nhất có thể có với thứ tự tiền thứ tự là $[1, 2, 4, 5, 6, 3, 7]$ và thứ tự trung thứ tự là $[4, 2, 6, 5, 1, 3, 7]$. Tương tự, thứ tự hậu thứ tự và trung thứ tự cũng xác định cấu trúc của một cây.

Tuy nhiên, tình hình sẽ khác nếu chúng ta chỉ biết thứ tự tiền thứ tự và hậu thứ tự của một cây. Trong trường hợp này, có thể có nhiều hơn một cây khớp với các thứ tự. Ví dụ, trong cả hai cây



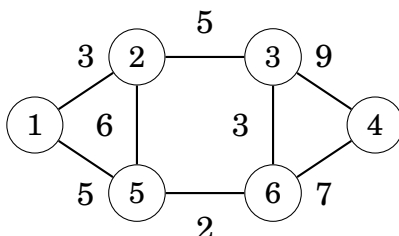
thứ tự tiền thứ tự là $[1, 2]$ và thứ tự hậu thứ tự là $[2, 1]$, nhưng cấu trúc của các cây là khác nhau.

Chương 15

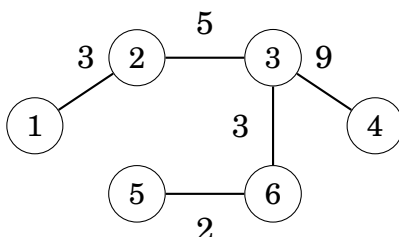
Cây khung

Một **cây khung** của một đồ thị bao gồm tất cả các đỉnh của đồ thị và một số cạnh của đồ thị sao cho có một đường đi giữa hai đỉnh bất kỳ. Giống như cây nói chung, cây khung liên thông và không có chu trình. Thông thường có một vài cách để xây dựng một cây khung.

Ví dụ, xét đồ thị sau:

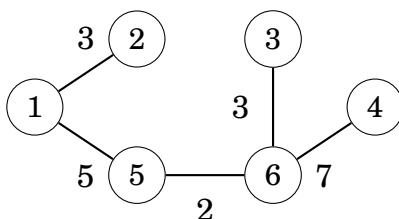


Một cây khung cho đồ thị như sau:

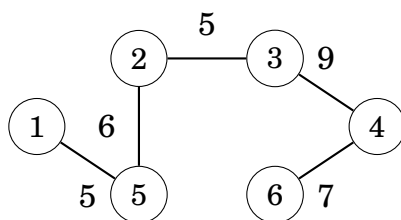


Trọng số của một cây khung là tổng trọng số các cạnh của nó. Ví dụ, trọng số của cây khung trên là $3 + 5 + 9 + 3 + 2 = 22$.

Một **cây khung nhỏ nhất** là một cây khung có trọng số nhỏ nhất có thể. Trọng số của một cây khung nhỏ nhất cho 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 một cây khung có trọng số lớn nhất có thể. Trọng số của một cây khung lớn nhất cho đồ thị ví dụ là 32:



Lưu ý rằng một đồ thị có thể có một vài cây khung nhỏ nhất và lớn nhất, vì vậy các cây không phải là duy nhất.

Hóa ra một số phương pháp tham lam có thể được sử dụng để xây dựng cây khung nhỏ nhất và lớn nhất. Trong chương này, chúng ta thảo luận về hai thuật toán xử lý các cạnh của đồ thị được sắp xếp theo trọng số của chúng. Chúng 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.

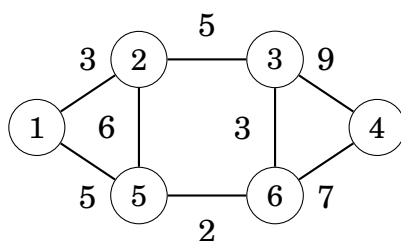
15.1 Thuật toán Kruskal

Trong **thuật toán Kruskal**¹, cây khung ban đầu chỉ chứa các đỉnh của đồ thị và không chứa bất kỳ cạnh nào. Sau đó, thuật toán duyệt qua các cạnh theo thứ tự trọng số của chúng, và luôn thêm một cạnh vào cây nếu nó không tạo ra một chu trình.

Thuật toán duy trì các thành phần của cây. Ban đầu, mỗi đỉnh của đồ thị thuộc về một thành phần riêng biệt. Luôn luôn khi một cạnh được thêm vào cây, hai thành phần được hợp nhất. Cuối cùng, tất cả các đỉnh thuộc về cùng một thành phần, và một cây khung nhỏ nhất đã được tìm thấy.

Ví dụ

Hãy xem xét cách thuật toán Kruskal xử lý đồ thị sau:



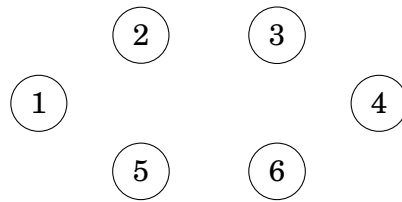
Bước đầu tiên của thuật toán là sắp xếp các cạnh theo thứ tự tăng dần của trọng số của chúng. Kết quả là danh sách sau:

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

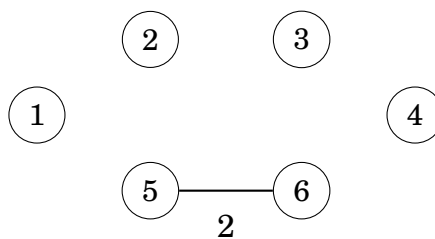
¹Thuật toán được công bố năm 1956 bởi J. B. Kruskal [48].

Sau đó, thuật toán duyệt qua danh sách và thêm mỗi cạnh vào cây nếu nó nối hai thành phần riêng biệt.

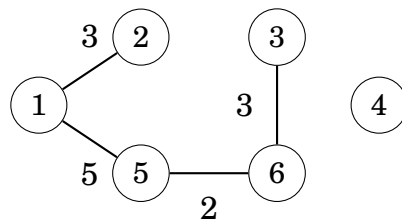
Ban đầu, mỗi đỉnh nằm trong thành phần riêng của nó:



Cạnh đầu tiên được thêm vào cây là cạnh 5–6 tạo ra một thành phần {5,6} bằng cách nối các thành phần {5} và {6}:



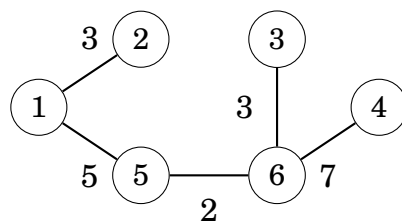
Sau đó, các cạnh 1–2, 3–6 và 1–5 được thêm vào một cách tương tự:



Sau các bước đó, hầu hết các thành phần đã được nối và có hai thành phần trong cây: {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 ở trong cùng một thành phần. Vì lý do tương tự, cạnh 2–5 sẽ không được đưa vào cây.

Cuối cùng, cạnh 4–6 sẽ được đưa vào cây:

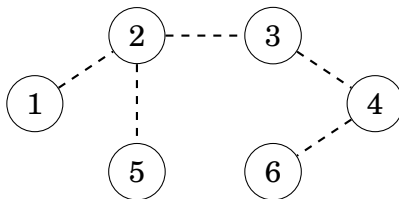


Sau đó, thuật toán sẽ không thêm bất kỳ cạnh mới nào, vì đồ thị đã liên thông và có một đường đi giữa hai đỉnh bất kỳ. Đồ thị kết quả là một cây khung nhỏ nhất với trọng số $2 + 3 + 3 + 5 + 7 = 20$.

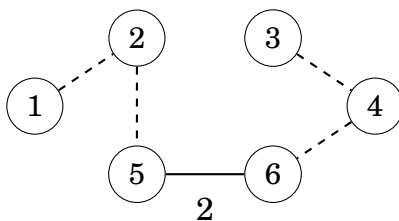
Tại sao thuật toán này hoạt động?

Một câu hỏi hay là tại sao thuật toán Kruskal hoạt động. Tại sao chiến lược tham lam lại đảm bảo rằng chúng ta sẽ tìm thấy một cây khung nhỏ nhất?

Hãy xem điều gì xảy ra nếu cạnh có trọng số nhỏ nhất của đồ thị *không* được bao gồm trong cây khung. Ví dụ, giả sử rằng một cây khung cho đồ thị trước đó sẽ không chứa cạnh có trọng số nhỏ nhất 5–6. Chúng ta không biết cấu trúc chính xác của một cây khung như vậy, nhưng trong mọi trường hợp, nó phải chứa một số cạnh. Giả sử rằng cây sẽ như sau:



Tuy nhiên, không thể nào cây trên lại là một cây khung nhỏ nhất cho đồ thị. Lý do là chúng ta có thể loại bỏ một cạnh khỏi cây và thay thế nó bằng cạnh có trọng số nhỏ nhất 5–6. Điều này tạo ra một cây khung có trọng số *nhỏ hơn*:



Vì lý do này, luôn tối ưu khi bao gồm cạnh có trọng số nhỏ nhất trong cây để tạo ra một cây khung nhỏ nhất. Sử dụng một lập luận tương tự, chúng ta có thể chỉ ra rằng cũng tối ưu khi thêm cạnh tiếp theo theo thứ tự trọng số vào cây, và cứ thế. Do đó, thuật toán Kruskal hoạt động chính xác và luôn tạo ra một cây khung nhỏ nhất.

Cài đặt

Khi cài đặt thuật toán Kruskal, sẽ thuận tiện khi sử dụng biểu diễn danh sách cạnh của đồ thị. Giai đoạn đầu tiên của thuật toán sắp xếp các cạnh trong danh sách theo 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 có hai hàm: hàm `same` xác định liệu a và b có ở trong cùng một thành phần hay không, và hàm `unite` hợp nhất các thành phần chứa a và b .

Vấn đề là làm thế nào để cài đặt hiệu quả các hàm `same` và `unite`. Một khả năng là cài đặt hàm `same` như một phép duyệt đồ thị và kiểm tra xem chúng ta có thể đi từ đỉnh a đến đỉnh b hay không. Tuy nhiên, độ phức tạp thời gian của một hàm như vậy sẽ là $O(n + m)$ và thuật toán kết quả sẽ chậm, bởi vì hàm `same` sẽ được gọi cho mỗi cạnh trong đồ thị.

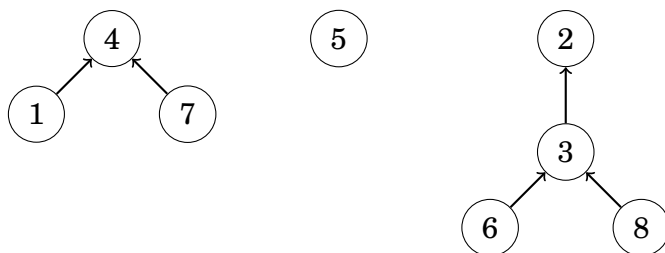
Chúng ta sẽ giải quyết vấn đề bằng cách sử dụng cấu trúc union-find cài đặt cả hai hàm trong thời gian $O(\log n)$. Do đó, độ phức tạp thời gian của thuật toán Kruskal sẽ là $O(m \log n)$ sau khi sắp xếp danh sách cạnh.

15.2 Cấu trúc Union-find

Một **cấu trúc union-find** duy trì một tập hợp các tập hợp. Các tập hợp là rời rạc, vì vậy không có phần tử nào thuộc về nhiều hơn một tập hợp. Hai thao tác thời gian $O(\log n)$ được hỗ trợ: thao tác `unite` hợp nhất hai tập hợp, và thao tác `find` tìm đại diện của tập hợp chứa một phần tử đã cho².

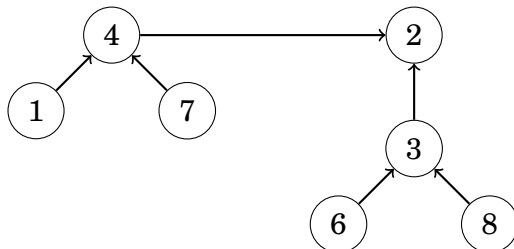
Cấu trúc

Trong một cấu trúc union-find, một phần tử trong mỗi tập hợp là đại diện của tập hợp, và có một chuỗi từ bất kỳ phần tử nào khác của tập hợp đến đại diện. Ví dụ, giả sử các tập hợp là $\{1, 4, 7\}$, $\{5\}$ và $\{2, 3, 6, 8\}$:



Trong trường hợp này, các đại diện của các tập hợp là 4, 5 và 2. Chúng 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 chúng ta đi theo chuỗi $6 \rightarrow 3 \rightarrow 2$. Hai phần tử thuộc cùng một tập hợp chính xác khi đại diện của chúng giống nhau.

Hai tập hợp có thể được hợp nhất bằng cách kết nối đại diện của một tập hợp với đại diện của tập hợp kia. Ví dụ, các tập hợp $\{1, 4, 7\}$ và $\{2, 3, 6, 8\}$ có thể được hợp nhất như sau:



Tập hợp kết quả chứa các phần tử $\{1, 2, 3, 4, 6, 7, 8\}$. Từ đây trở đi, phần tử 2 là đại diện cho toàn bộ tập hợp và đại diện cũ 4 trở đến phần tử 2.

Hiệu quả của cấu trúc union-find phụ thuộc vào cách các tập hợp được hợp nhất. Hóa ra chúng ta có thể tuân theo một chiến lược đơn giản: luôn kết nối đại diện của tập hợp *nhỏ hơn* với đại diện của tập hợp *lớn hơn* (hoặc nếu các tập hợp có kích thước bằng nhau, chúng ta có thể đưa ra một lựa chọn tùy ý). Sử dụng chiến lược này, độ dài của bất kỳ chuỗi nào sẽ là $O(\log n)$, vì vậy chúng 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 mảng. Trong cài đặt sau, mảng `link` chứa cho mỗi phần tử phần tử tiếp theo trong chuỗi hoặc chính phần tử đó nếu nó là một đại diện, và mảng `size` chỉ ra cho mỗi đại diện kích thước của tập hợp tương ứng.

²Cấu trúc được trình bày ở đây được giới thiệu vào năm 1971 bởi J. D. Hopcroft và J. D. Ullman [38]. Sau đó, vào năm 1975, R. E. Tarjan đã nghiên cứu một biến thể phức tạp hơn của cấu trúc [64] được thảo luận trong nhiều sách giáo khoa thuật toán ngày nay.

Ban đầu, mỗi phần tử thuộc về một tập hợp riêng biệt:

```
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 cho một phần tử x . Đại diện có thể được tìm thấy 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 liệu các phần tử a và b có thuộc cùng một tập hợp hay không. Điều này có thể dễ dàng được thực hiện bằng cách sử dụng hàm `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

Hàm `unite` hợp nhất các tập hợp chứa các phần tử a và b (các phần tử phải ở trong các tập hợp khác nhau). Hàm đầu tiên tìm các đại diện của các tập hợp và sau đó kết nối tập hợp nhỏ hơn với tập hợp 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 $O(\log n)$ giả sử rằng độ dài của mỗi chuỗi là $O(\log n)$. Trong trường hợp này, các hàm `same` và `unite` cũng hoạt động trong thời gian $O(\log n)$. Hàm `unite` đảm bảo rằng độ dài của mỗi chuỗi là $O(\log n)$ bằng cách kết nối tập hợp nhỏ hơn với tập hợp lớn hơn.

15.3 Thuật toán Prim

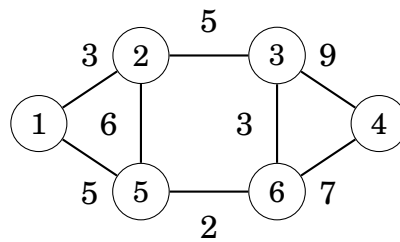
Thuật toán Prim³ là một phương pháp thay thế để tìm một cây khung nhỏ nhất. Thuật toán đầu tiên thêm một đỉnh tùy ý vào cây. Sau đó, thuật toán luôn chọn một cạnh có trọng số nhỏ nhất thêm một đỉnh mới vào cây. Cuối cùng, tất cả các đỉnh đã được thêm vào cây và một cây khung nhỏ nhất đã được tìm thấy.

Thuật toán Prim giống với thuật toán Dijkstra. Sự khác biệt là thuật toán Dijkstra luôn chọn một cạnh có khoảng cách từ đỉnh bắt đầu là nhỏ nhất, nhưng thuật toán Prim chỉ đơn giản là chọn cạnh có trọng số nhỏ nhất để thêm một đỉnh mới vào cây.

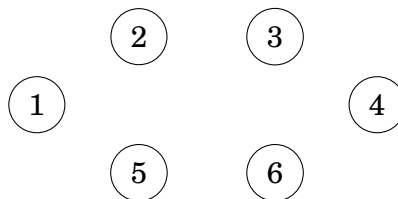
³Thuật toán được đặt theo tên của R. C. Prim, người đã công bố nó vào năm 1957 [54]. Tuy nhiên, thuật toán tương tự đã được phát hiện vào năm 1930 bởi V. Jarník.

Ví dụ

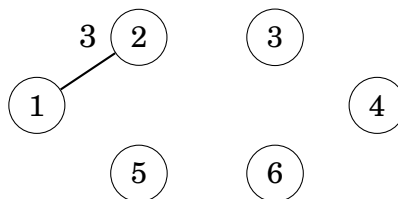
Hãy xem xét cách thuật toán Prim hoạt động trong đồ thị sau:



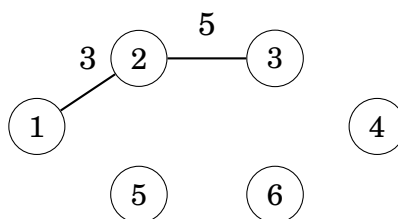
Ban đầu, không có cạnh nào giữa các đỉnh:



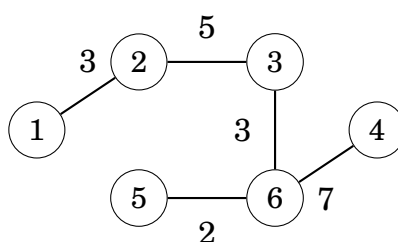
Một đỉnh tùy ý có thể là đỉnh bắt đầu, vì vậy hãy chọn đỉnh 1. Đầu tiên, chúng ta thêm đỉnh 2 được nối bằng một cạnh có trọng số 3:



Sau đó, có hai cạnh có trọng số 5, vì vậy chúng ta có thể thêm đỉnh 3 hoặc đỉnh 5 vào cây. Hãy thêm đỉnh 3 trước:



Quá trình tiếp tục cho đến khi tất cả các đỉnh đã được đưa vào cây:



Cài đặt

Giống như 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 kết nối với thành phần hiện tại bằng một cạnh duy nhất, 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, thuật toán Prim và Kruskal đều hiệu quả, và việc lựa chọn thuật toán là vấn đề sở thích. Tuy nhiên, hầu hết các lập trình viên thi đấu đều sử dụng thuật toán Kruskal.

Chương 16

Đồ thị có hướng

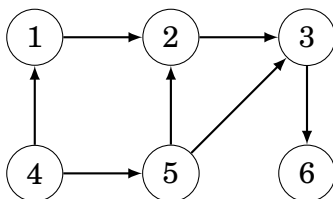
Trong chương này, chúng ta tập trung vào hai loại đồ thị có hướng:

- **Đồ thị không có chu trình (Acyclic graphs):** Không có chu trình nào trong đồ thị, vì vậy không có đường đi từ bất kỳ nút nào đến chính nó¹.
- **Đồ thị kế tiếp (Successor graphs):** Bán bậc ngoài của mỗi nút là 1, vì vậy mỗi nút có một nút kế tiếp duy nhất.

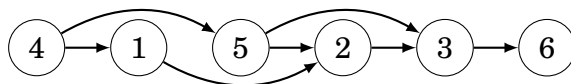
Hóa ra trong cả hai trường hợp, chúng ta có thể thiết kế các thuật toán hiệu quả dựa trên các thuộc tính đặc biệt của các đồ thị này.

16.1 Sắp xếp tô pô (Topological sorting)

Một **sắp xếp tô pô** là một cách sắp xếp thứ tự các nút của một đồ thị có hướng sao cho nếu có một đường đi từ nút a đến nút b , thì nút a xuất hiện trước nút b trong thứ tự đó. Ví dụ, đối với đồ thị



một cách sắp xếp tô pô là [4, 1, 5, 2, 3, 6]:



Một đồ thị không có chu trình luôn có một cách sắp xếp tô pô. Tuy nhiên, nếu đồ thị chứa một chu trình, thì không thể tạo thành một sắp xếp tô pô, bởi vì không có nút nào của chu trình có thể xuất hiện trước các nút khác của chu trình trong thứ tự đó. Hóa ra, tìm kiếm theo chiều sâu (depth-first search) có thể được sử dụng để vừa kiểm tra xem một đồ thị có hướng có chứa chu trình hay không và, nếu nó không chứa chu trình, để xây dựng một sắp xếp tô pô.

¹Đồ thị có hướng không chu trình đôi khi được gọi là DAGs.

Thuật toán

Ý tưởng là duyệt qua các nút của đồ thị và luôn bắt đầu một tìm kiếm theo chiều sâu tại nút hiện tại nếu nó chưa được xử lý. Trong quá trình tìm kiếm, các nút có ba trạng thái khả dĩ:

- trạng thái 0: nút chưa được xử lý (màu trắng)
- trạng thái 1: nút đang được xử lý (màu xám nhạt)
- trạng thái 2: nút đã được xử lý (màu xám đậm)

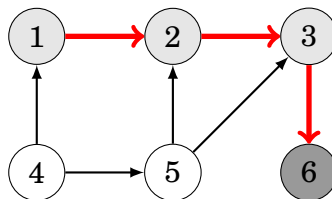
Ban đầu, trạng thái của mỗi nút là 0. Khi một tìm kiếm đến một nút lần đầu tiên, trạng thái của nó trở thành 1. Cuối cùng, sau khi tất cả các nút kế tiếp của nút đó đã được xử lý, trạng thái của nó trở thành 2.

Nếu đồ thị chứa một chu trình, chúng ta sẽ phát hiện ra điều này trong quá trình tìm kiếm, bởi vì sớm hay muộn chúng ta sẽ đến một nút có trạng thái là 1. Trong trường hợp này, không thể xây dựng một sắp xếp tô pô.

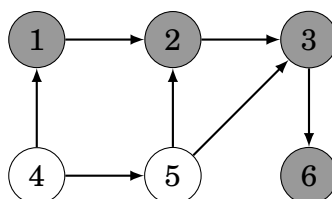
Nếu đồ thị không chứa chu trình, chúng ta có thể xây dựng một sắp xếp tô pô bằng cách thêm mỗi nút vào một danh sách khi trạng thái của nút đó trở thành 2. Danh sách này theo thứ tự ngược lại là một sắp xếp tô pô.

Ví dụ 1

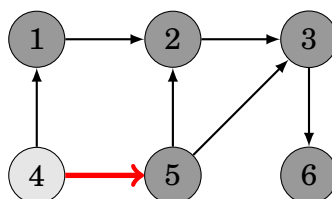
Trong đồ thị ví dụ, việc tìm kiếm đầu tiên tiến hành từ nút 1 đến nút 6:



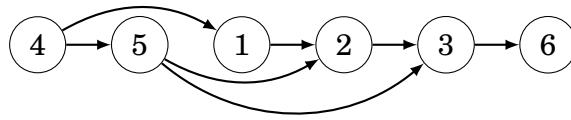
Bây giờ nút 6 đã được xử lý, vì vậy nó được thêm vào danh sách. Sau đó, các nút 3, 2 và 1 cũng được thêm vào danh sách:



Tại thời điểm này, danh sách là [6,3,2,1]. Việc tìm kiếm tiếp theo bắt đầu tại nút 4:



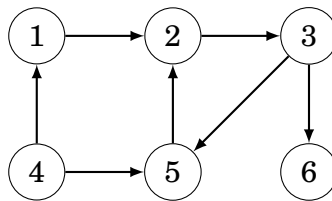
Do đó, danh sách cuối cùng là [6,3,2,1,5,4]. Chúng ta đã xử lý tất cả các nút, vì vậy một sắp xếp tô pô đã được tìm thấy. Sắp xếp tô pô là danh sách đảo ngược [4,5,1,2,3,6]:



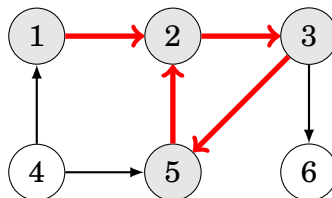
Lưu ý rằng một sắp xếp tô pô không phải là duy nhất, và có thể có nhiều cách sắp xếp tô pô cho một đồ thị.

Ví dụ 2

Bây giờ chúng ta hãy xem xét một đồ thị mà chúng ta không thể xây dựng một sắp xếp tô pô, bởi vì đồ thị chứa một chu trình:



Việc tìm kiếm tiền hành như sau:



Việc tìm kiếm đến nút 2 có trạng thái là 1, điều này có nghĩa là đồ thị chứa một chu trình. Trong ví dụ này, có một chu trình $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

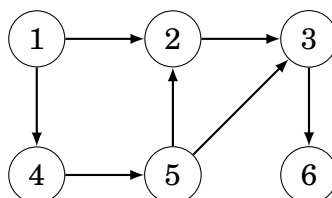
16.2 Quy hoạch động (Dynamic programming)

Nếu một đồ thị có hướng là không có chu trình, quy hoạch động có thể được áp dụng cho nó. Ví dụ, chúng ta có thể giải quyết hiệu quả các vấn đề sau liên quan đến các đường đi từ một nút bắt đầu đến một nút kết thúc:

- có bao nhiêu đường đi khác nhau?
- đường đi ngắn nhất/dài nhất là gì?
- số cạnh tối thiểu/tối đa trong một đường đi là bao nhiêu?
- nút nào chắc chắn xuất hiện trong bất kỳ đường đi nào?

Đếm số lượng đường đi

Ví dụ, chúng ta hãy tính số lượng đường đi từ nút 1 đến nút 6 trong đồ thị sau:



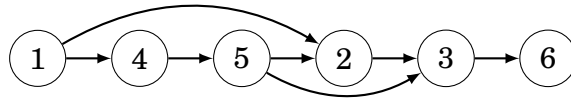
Có tổng cộng ba đường đi như vậy:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

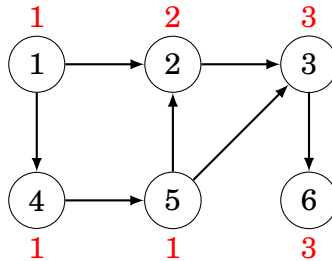
Gọi $\text{paths}(x)$ là số lượng đường đi từ nút 1 đến nút x . Là trường hợp cơ sở, $\text{paths}(1) = 1$. Sau đó, để tính các giá trị khác của $\text{paths}(x)$, chúng ta có thể sử dụng công thức đệ quy

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

trong đó a_1, a_2, \dots, a_k là các nút mà từ đó có một cạnh đến x . Vì đồ thị không có chu trình, các giá trị của $\text{paths}(x)$ có thể được tính theo thứ tự của một sắp xếp tô pô. Một sắp xếp tô pô cho đồ thị trên như sau:



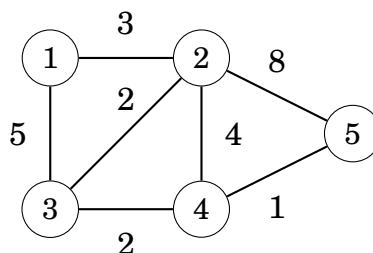
Do đó, số lượng đường đi như sau:



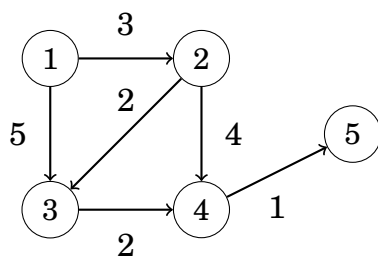
Ví dụ, để tính giá trị của $\text{paths}(3)$, chúng ta có thể sử dụng công thức $\text{paths}(2) + \text{paths}(5)$, bởi vì có các cạnh từ các nút 2 và 5 đến nút 3. Vì $\text{paths}(2) = 2$ và $\text{paths}(5) = 1$, chúng ta kết luận rằng $\text{paths}(3) = 3$.

Mở rộng thuật toán Dijkstra

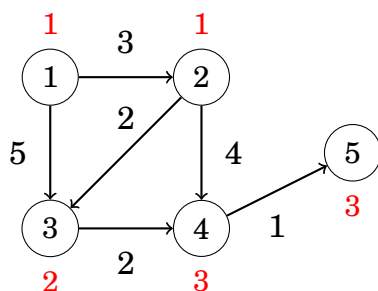
Một sản phẩm phụ của thuật toán Dijkstra là một đồ thị có hướng, không chu trình chỉ ra cho mỗi nút của đồ thị ban đầu các cách khả dĩ để đến nút đó bằng một đường đi ngắn nhất từ nút bắt đầu. Quy hoạch động có thể được áp dụng cho đồ thị đó. Ví dụ, trong đồ thị



các đường đi ngắn nhất từ nút 1 có thể sử dụng các cạnh sau:



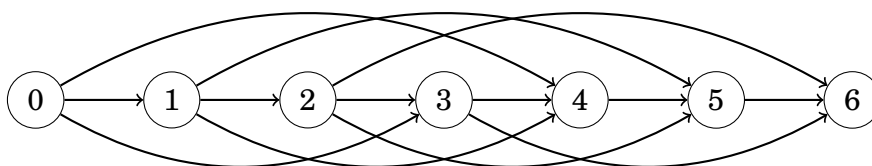
Bây giờ chúng ta có thể, ví dụ, tính số lượng đường đi ngắn nhất từ nút 1 đến nút 5 sử dụng quy hoạch động:



Biểu diễn bài toán dưới dạng đồ thị

Thực ra, bất kỳ bài toán quy hoạch động nào cũng có thể được biểu diễn dưới dạng một đồ thị có hướng, không chu trình. Trong một đồ thị như vậy, mỗi nút tương ứng với một trạng thái quy hoạch động và các cạnh chỉ ra cách các trạng thái phụ thuộc vào nhau.

Ví dụ, xem xét bài toán tạo thành một tổng tiền n sử dụng các đồng xu $\{c_1, c_2, \dots, c_k\}$. Trong bài toán này, chúng ta có thể xây dựng một đồ thị trong đó mỗi nút tương ứng với một tổng tiền, và các cạnh cho thấy cách các đồng xu có thể được chọn. Ví dụ, đối với các đồng xu $\{1, 3, 4\}$ và $n = 6$, đồ thị như sau:



Sử dụng biểu diễn này, đường đi ngắn nhất từ nút 0 đến nút n tương ứng với một giải pháp với số lượng đồng xu tối thiểu, và tổng số đường đi từ nút 0 đến nút n bằng tổng số lời giải.

16.3 Đường đi kế tiếp

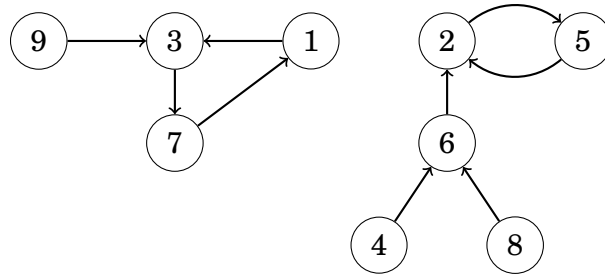
Trong phần còn lại của chương, chúng ta sẽ tập trung vào **đồ thị kế tiếp (successor graphs)**. Trong các đồ thị đó, bán bậc ngoài của mỗi nút là 1, tức là, chính xác một cạnh bắt đầu tại mỗi nút. Một đồ thị kế tiếp bao gồm một hoặc nhiều thành phần, mỗi thành phần chứa một chu trình và một số đường đi dẫn đến nó.

Đồ thị kế tiếp đôi khi được gọi là **đồ thị hàm (functional graphs)**. Lý do cho điều này là bất kỳ đồ thị kế tiếp nào cũng tương ứng với một hàm xác định các cạnh của đồ thị. Tham số cho hàm là một nút của đồ thị, và hàm trả về nút kế tiếp của nút đó.

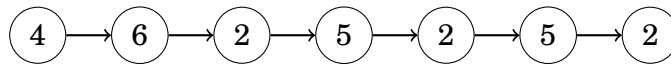
Ví dụ, hàm

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

xác định đồ thị sau:



Vì mỗi nút của một đồ thị kế tiếp có một nút kế tiếp duy nhất, chúng ta cũng có thể định nghĩa một hàm $\text{succ}(x, k)$ trả về nút mà chúng ta sẽ đến nếu chúng ta bắt đầu tại nút x và đi k bước về phía trước. Ví dụ, trong đồ thị trên $\text{succ}(4, 6) = 2$, bởi vì chúng ta sẽ đến nút 2 bằng cách đi 6 bước từ nút 4:



Một cách đơn giản để tính một giá trị của $\text{succ}(x, k)$ là bắt đầu tại nút x và đi k bước về phía trước, mất thời gian $O(k)$. Tuy nhiên, bằng cách sử dụng tiền xử lý, bất kỳ giá trị nào của $\text{succ}(x, k)$ cũng có thể được tính chỉ trong thời gian $O(\log k)$.

Ý tưởng là tính trước tất cả các giá trị của $\text{succ}(x, k)$ trong đó k là một lũy thừa của hai và tối đa là u , trong đó u là số bước tối đa chúng ta sẽ đi. Điều này có thể được thực hiện hiệu quả, bởi vì chúng ta có thể sử dụng công thức đệ quy 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 toán trước các giá trị mất thời gian $O(n \log u)$, bởi vì $O(\log u)$ giá trị được tính cho mỗi nút. 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, bất kỳ giá trị nào của $\text{succ}(x, k)$ cũng có thể được tính toán bằng cách biểu diễn số bước k dưới dạng tổng các lũy thừa của hai. Ví dụ, nếu chúng ta muốn tính giá trị của $\text{succ}(x, 11)$, chúng ta trước tiên tạo biểu diễn $11 = 8 + 2 + 1$. Sử dụng điều đó,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Ví dụ, trong đồ thị trước đó

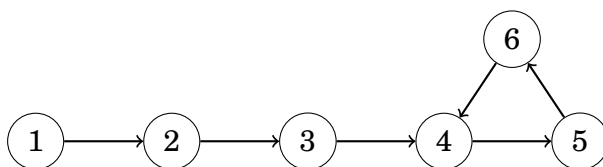
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Một biểu diễn như vậy luôn bao gồm $O(\log k)$ phần, vì vậy việc tính một giá trị của $\text{succ}(x, k)$ mất thời gian $O(\log k)$.

16.4 Dò tìm chu trình (Cycle detection)

Xem xét một đồ thị kế tiếp chỉ chứa một đường đi kết thúc bằng một chu trình. Chúng ta có thể đặt ra các câu hỏi sau: nếu chúng ta bắt đầu đi từ nút xuất phát, nút đầu tiên trong chu trình là gì và chu trình chứa bao nhiêu nút?

Ví dụ, trong đồ thị



chúng ta bắt đầu đi tại nút 1, nút đầu tiên thuộc về chu trình là nút 4, và chu trình bao gồm ba nút (4, 5 và 6).

Một cách đơn giản để phát hiện chu trình là đi trong đồ thị và theo dõi tất cả các nút đã được thăm. Một khi một nút được thăm lần thứ hai, chúng ta có thể kết luận rằng nút đó là nút đầu tiên trong chu trình. Phương pháp này hoạt động trong thời gian $O(n)$ và cũng sử dụng bộ nhớ $O(n)$.

Tuy nhiên, có những thuật toán tốt hơn để dò tìm chu trình. Độ phức tạp thời gian của các thuật toán như vậy vẫn là $O(n)$, nhưng chúng chỉ sử dụng bộ nhớ $O(1)$. Đây là một cải tiến quan trọng nếu n lớn. Tiếp theo chúng ta sẽ thảo luận về thuật toán Floyd đạt được các thuộc tính này.

Thuật toán Floyd

Thuật toán Floyd (Floyd's algorithm)² đi về phía trước trong đồ thị bằng cách sử dụng hai con trỏ a và b . Cả hai con trỏ đều bắt đầu tại một nút x là nút xuất phát của đồ thị. Sau đó, ở mỗi lượt, con trỏ a đi một bước về phía trước và con trỏ b đi hai bước về phía trước. Quá trình tiếp tục cho đến khi các 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 được k bước và con trỏ b đã đi được $2k$ bước, vì vậy độ dài của chu trình là ước của k . Do đó, nút đầu tiên thuộc về chu trình có thể được tìm thấy bằng cách di chuyển con trỏ a đến nút x và tiến các con trỏ từng bước một cho đến khi chúng gặp lại nhau.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a; // nút đầu tiên
```

²Ý tưởng của thuật toán được đề cập trong [46] và được cho là của R. W. Floyd; tuy nhiên, không rõ liệu Floyd có thực sự phát hiện ra thuật toán này hay không.

Sau đó, độ dài của chu trình có thể được tính như sau:

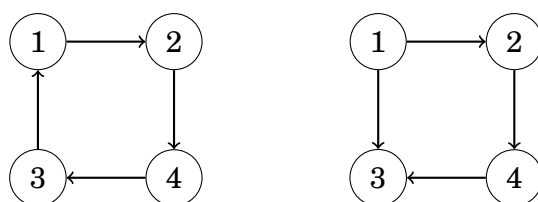
```
b = succ(a);  
length = 1; // do dai  
while (a != b) {  
    b = succ(b);  
    length++;  
}
```

Chương 17

Liên thông mạnh (Strong connectivity)

Trong một đồ thị có hướng, các cạnh chỉ có thể được duyệt theo một hướng, vì vậy ngay cả khi đồ thị là liên thông, điều này không đảm bảo rằng sẽ có một đường đi từ một nút đến một nút khác. Vì lý do này, việc định nghĩa một khái niệm mới đòi hỏi nhiều hơn tính liên thông là có ý nghĩa.

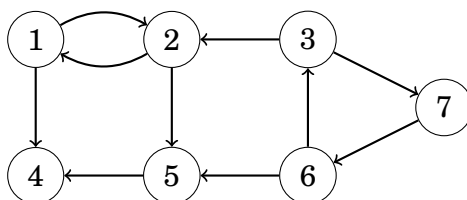
Một đồ thị được gọi là **liên thông mạnh (strongly connected)** nếu có một đường đi từ bất kỳ nút nào đến tất cả các nút khác trong đồ thị. Ví dụ, trong hình sau, đồ thị bên trái là liên thông mạnh trong khi đồ thị bên phải thì không.



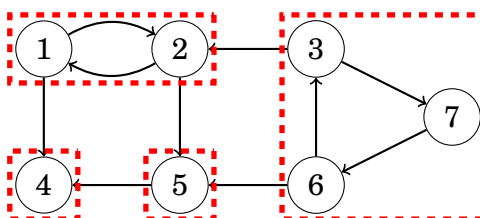
Đồ thị bên phải không liên thông mạnh bởi vì, ví dụ, không có đường đi từ nút 2 đến nút 1.

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 (component graph)** không có chu trình, đại diện cho cấu trúc sâu của đồ thị ban đầu.

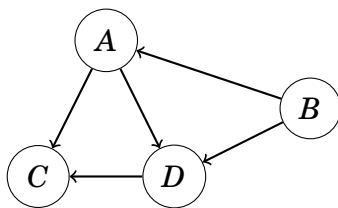
Ví dụ, đối với đồ thị



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 $D = \{5\}$.

Một 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, chúng ta luôn có thể xây dựng một sắp xếp tô pô và sử dụng các kỹ thuật quy hoạch động như những kỹ thuật đã được trình bày trong Chương 16.

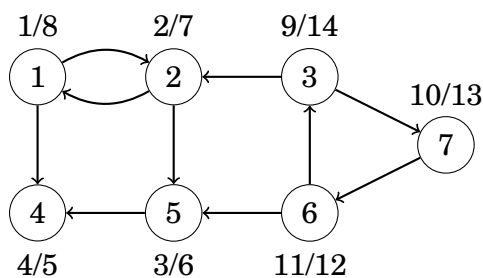
17.1 Thuật toán Kosaraju

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 nút theo cấu trúc của đồ thị, và lần tìm kiếm thứ hai hình thành các thành phần liên thông mạnh.

Tìm kiếm 1

Giai đoạn đầu tiên của thuật toán Kosaraju xây dựng một danh sách các nút theo thứ tự mà một tìm kiếm theo chiều sâu xử lý xong chúng. Thuật toán duyệt qua các nút, và bắt đầu một tìm kiếm theo chiều sâu tại mỗi nút chưa được xử lý. Mỗi nút sẽ được thêm vào danh sách sau khi nó đã được xử lý xong.

Trong đồ thị ví dụ, các nút được xử lý theo thứ tự sau:



Ký hiệu x/y có nghĩa là việc xử lý nút bắt đầu tại thời điểm x và kết thúc tại thời điểm y . Do đó, danh sách tương ứng như sau:

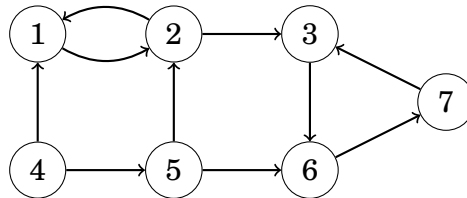
nút	thời gian xử lý xong
4	5
5	6
2	7
1	8
6	12
7	13
3	14

¹Theo [1], S. R. Kosaraju đã phát minh ra thuật toán này vào năm 1978 nhưng không công bố nó. Năm 1981, thuật toán tương tự đã được tái khám phá và công bố bởi M. Sharir [57].

Tìm kiếm 2

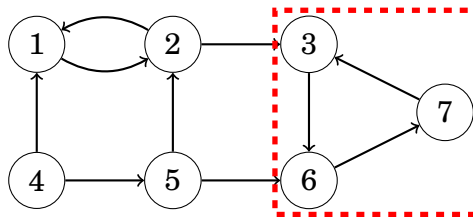
Giai đoạn thứ hai của thuật toán hình thành các thành phần liên thông mạnh của đồ thị. Đầu tiên, thuật toán đảo ngược mọi cạnh trong đồ thị. Điều này đảm bảo rằng trong lần tìm kiếm thứ hai, chúng ta sẽ luôn tìm thấy các thành phần liên thông mạnh mà không có các nút thừa.

Sau khi đảo ngược các cạnh, đồ thị ví dụ như sau:



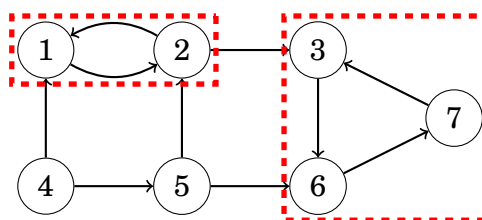
Sau đó, thuật toán duyệt qua danh sách các nút được tạo bởi lần tìm kiếm đầu tiên, theo thứ tự *ngược lại*. Nếu một nút không thuộc về một thành phần nào, thuật toán tạo một thành phần mới và bắt đầu một tìm kiếm theo chiều sâu thêm tất cả các nút mới tìm thấy trong quá trình tìm kiếm vào thành phần mới.

Trong đồ thị ví dụ, thành phần đầu tiên bắt đầu tại nút 3:

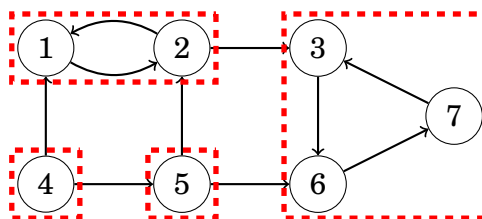


Lưu ý rằng vì tất cả các cạnh đều bị đảo ngược, thành phần không bị "tràn" sang các phần khác trong đồ thị.

Các nút tiếp theo trong danh sách là nút 7 và 6, nhưng chúng đã thuộc về một thành phần, vì vậy thành phần mới tiếp theo bắt đầu tại nút 1:



Cuối cùng, thuật toán xử lý các nút 5 và 4 tạo ra các thành phần liên thông mạnh còn lại:



Độ phức tạp thời gian của thuật toán là $O(n + m)$, bởi vì thuật toán thực hiện hai lần tìm kiếm theo chiều sâu.

17.2 Bài toán 2SAT

Liên thông mạnh cũng liên quan đến **bài toán 2SAT (2SAT problem)**². Trong bài toán này, chúng 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 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 thị các toán tử logic "và" và "hoặc". Nhiệm vụ của chúng ta là gán cho mỗi biến một giá trị sao cho công thức là đúng, hoặc phát biểu rằng điều này là không thể.

Ví dụ, công thức

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

là đúng khi các biến được gán như sau:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

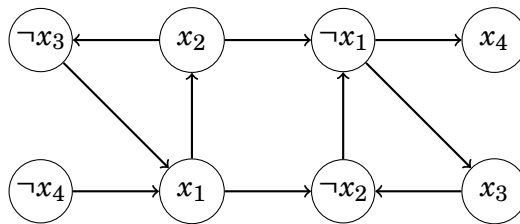
Tuy nhiên, công thức

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

luôn sai, bất kể chúng ta gán giá trị như thế nào. Lý do cho điều này là chúng ta không thể chọn một giá trị cho x_1 mà không tạo ra mâu thuẫn. Nếu x_1 là sai, cả x_2 và $\neg x_2$ phải là đúng, điều này là không thể, và nếu x_1 là đúng, cả x_3 và $\neg x_3$ phải là đúng, điều này cũng không thể.

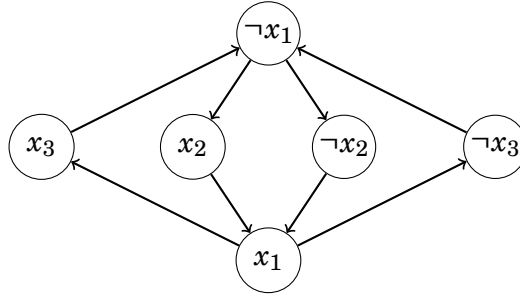
Bài toán 2SAT có thể được biểu diễn dưới dạng một đồ thị có các nút tương ứng với các biến x_i và các phủ định $\neg x_i$, và các cạnh xác định các mối liên kết giữa các biến. Mỗi cặp $(a_i \vee b_i)$ tạo ra hai cạnh: $\neg a_i \rightarrow b_i$ và $\neg b_i \rightarrow a_i$. Điều này có nghĩa là nếu a_i không đúng, thì b_i phải đúng, và ngược lại.

Đồ thị cho công thức L_1 là:



Và đồ thị cho công thức L_2 là:

²Thuật toán được trình bày ở đây đã được giới thiệu trong [4]. Cũng có một thuật toán thời gian tuyến tính nổi tiếng khác [19] dựa trên quay lui (backtracking).

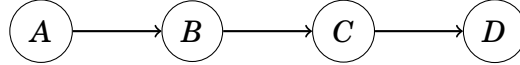


Cấu trúc của đồ thị cho chúng ta biết liệu có thể gán các giá trị của các biến sao cho công thức là đúng hay không. Hóa ra điều này có thể được thực hiện chính xác khi không có các nút x_i và $\neg x_i$ nào mà cả hai nút đều thuộc về cùng một thành phần liên thông mạnh. Nếu có các nút như vậy, đồ thị 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 là đúng điều này là không thể.

Trong đồ thị của công thức L_1 không có các nút x_i và $\neg x_i$ nào mà cả hai nút đều thuộc cùng một thành phần liên thông mạnh, vì vậy một lời giải tồn tại. Trong đồ thị của công thức L_2 tất cả các nút đều thuộc cùng một thành phần liên thông mạnh, vì vậy một lời giải không tồn tại.

Nếu một lời giải tồn tại, các giá trị cho các biến có thể được tìm thấy bằng cách duyệt qua các nút của đồ thị thành phần theo thứ tự sắp xếp tô pô ngược. Tại mỗi bước, chúng ta xử lý một thành phần không chứa các cạnh dẫn đến một thành phần chưa được xử lý. Nếu các biến trong thành phần chưa được gán giá trị, giá trị của chúng sẽ được xác định theo các giá trị trong thành phần, và nếu chúng đã có giá trị, chúng vẫn không thay đổi. 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, chúng ta trước tiên xử lý thành phần D nơi x_4 trở thành đúng. Sau đó, chúng ta xử lý thành phần C nơi x_1 và x_2 trở thành sai và x_3 trở thành đúng. Tất cả các biến đã được gán giá trị, vì vậy các thành phần còn lại A và B không thay đổi các biến.

Lưu ý rằng phương pháp này hoạt động, bởi vì đồ thị có một cấu trúc đặc biệt: nếu có các đường đi từ nút x_i đến nút x_j và từ nút x_j đến nút $\neg x_j$, thì nút x_i không bao giờ trở thành đúng. Lý do cho điều này là cũng có một đường đi từ nút $\neg x_j$ đến nút $\neg x_i$, và cả x_i và x_j đều trở thành sai.

Một bài toán khó hơn là **bài toán 3SAT (3SAT problem)**, trong đó mỗi phần của công thức có dạng $(a_i \vee b_i \vee c_i)$. Bài toán này là NP-khó (NP-hard), vì vậy không có thuật toán hiệu quả nào được biết đến để giải quyết bài toán này.

Chương 18

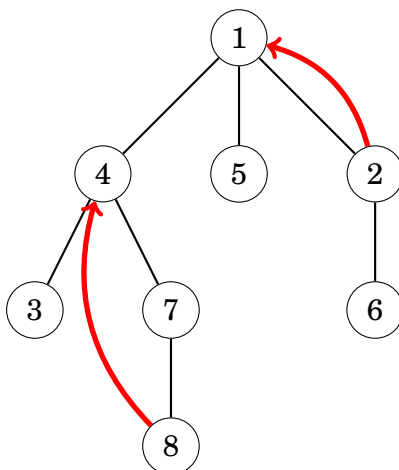
Truy vấn trên cây

Chương này thảo luận về các kỹ thuật xử lý các truy vấn trên các cây con và đường đi của một cây có gốc. Ví dụ, các truy vấn đó 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 một 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 một cây có gốc là đỉnh mà chúng ta sẽ đến nếu chúng ta di chuyển lên k mức từ x . Gọi $\text{ancestor}(x, k)$ là tổ tiên thứ k của một đỉnh x (hoặc 0 nếu không có tổ tiên như vậy). Ví dụ, trong cây sau, $\text{ancestor}(2, 1) = 1$ và $\text{ancestor}(8, 2) = 4$.



Một cách dễ dàng để tính bất kỳ giá trị nào của $\text{ancestor}(x, k)$ là thực hiện một chuỗi k lần di chuyển 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 có một chuỗi n đỉnh.

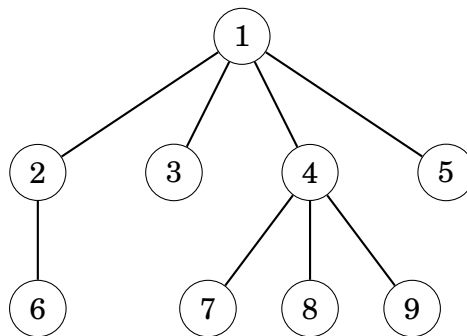
May mắn thay, sử dụng một kỹ thuật tương tự như kỹ thuật được sử dụng trong Chương 16.3, bất kỳ giá trị nào của $\text{ancestor}(x, k)$ cũng có thể được tính toán hiệu quả trong thời gian $O(\log k)$ sau khi tiền xử lý. Ý tưởng là tiền tính toán tất cả các giá trị $\text{ancestor}(x, k)$ trong đó $k \leq n$ là một 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
...								

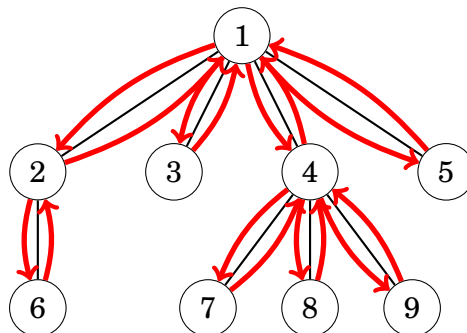
Việc tiền xử lý mất thời gian $O(n \log n)$, vì $O(\log n)$ giá trị được tính cho mỗi đỉnh. Sau đó, bất kỳ giá trị nào của $\text{ancestor}(x, k)$ cũng có thể được tính trong thời gian $O(\log k)$ bằng cách biểu diễn k dưới dạng tổng các lũy thừa của hai.

18.2 Cây con và đường đi

Một **mảng duyệt cây** chứa các đỉnh của một cây có gốc theo thứ tự mà một tìm kiếm theo chiều sâu từ đỉnh gốc duyệt qua 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

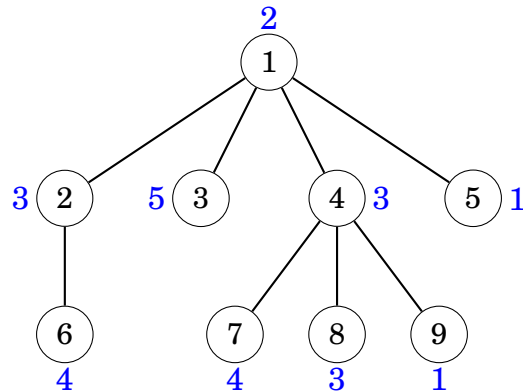
Mỗi cây con của một cây tương ứng với một mảng con của mảng duyệt cây sao cho phần tử đầu tiên của mảng con là đỉnh gốc. Ví dụ, mảng con sau đây chứa các đỉnh của cây con của đỉnh 4:

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

Sử dụng thực tế này, chúng ta có thể xử lý hiệu quả các truy vấn liên quan đến các cây con của một cây. Ví dụ, hãy xem xét một bài toán trong đó mỗi đỉnh được gán một giá trị, và nhiệm vụ của chúng 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

Hãy xem xét cây sau đây trong đó các số màu xanh là giá trị của các đỉnh. Ví dụ, tổng của cây con của đỉnh 4 là $3 + 4 + 3 + 1 = 11$.



Ý tưởng là xây dựng một mảng duyệt cây chứa ba giá trị cho mỗi đỉnh: định danh của đỉnh, kích thước của cây con, và giá trị của đỉnh. Ví dụ, mảng cho cây trên như sau:

định danh đỉnh	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
giá trị đỉnh	2	3	4	5	3	4	3	1	1

Sử dụng mảng này, chúng ta có thể tính tổng các giá trị trong bất kỳ cây con nào bằng cách trước tiên tìm ra kích thước của cây con và sau đó là giá trị của các đỉnh tương ứng. Ví dụ, các giá trị trong cây con của đỉnh 4 có thể được tìm thấy như sau:

định danh đỉnh	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
giá trị đỉnh	2	3	4	5	3	4	3	1	1

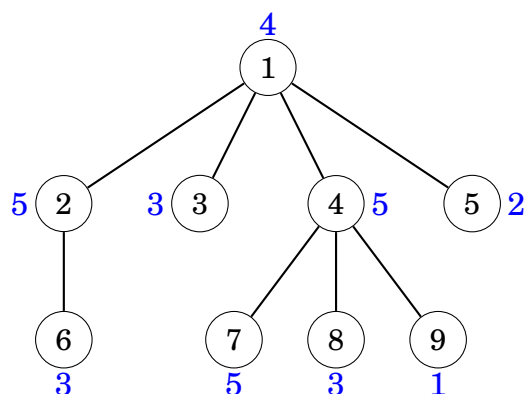
Để trả lời các truy vấn một cách hiệu quả, chỉ cần lưu trữ các giá trị của các đỉnh trong một cây chỉ số nhị phân hoặc cây phân đoạn. Sau đó, chúng ta có thể vừa cập nhật một giá trị vừa tính tổng các giá trị trong thời gian $O(\log n)$.

Truy vấn đường đi

Sử dụng một mảng duyệt cây, chúng ta cũng có thể tính toán hiệu quả tổng các giá trị trên các đường đi từ đỉnh gốc đến bất kỳ đỉnh nào của cây. Hãy xem xét một bài toán trong đó nhiệm vụ của chúng ta là hỗ trợ các truy vấn sau:

- thay đổi giá trị của một đỉnh
- tính tổng các giá trị trên một đường đi từ gốc đến một đỉnh

Ví dụ, trong cây sau, tổng các giá trị từ đỉnh gốc đến đỉnh 7 là $4 + 5 + 5 = 14$:



Chúng ta có thể giải quyết bài toán này như trước, nhưng bây giờ mỗi giá trị trong hàng cuối cùng của mảng là tổng các giá trị trên một đường đi từ gốc đến đỉnh. Ví dụ, mảng sau đây tương ứng với cây trên:

đỉnh danh đỉnh	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
tổng đường đi	2	3	4	5	3	4	3	1	1

Khi giá trị của một đỉnh tăng thêm x , tổng của tất cả các 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:

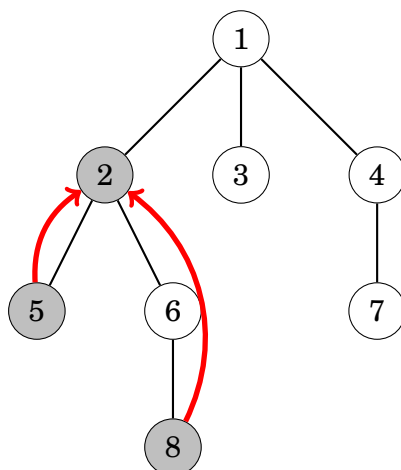
đỉnh danh đỉnh	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
tổng đường đi	2	3	4	5	3	4	3	1	1

Một lần nữa, chúng ta có thể sử dụng một cây chỉ số nhị phân hoặc cây phân đoạn để thực hiện các thao tác này một cách hiệu quả. Một truy vấn tổng đường đi có thể được trả lời bằng một truy vấn điểm, và một cập nhật giá trị tương ứng với một truy vấn phạm vi cộng một giá trị vào một phạm vi. Cả hai thao tác đều có thể được thực hiện trong thời gian $O(\log n)$.

18.3 Tổ tiên chung thấp nhất

Tổ tiên chung thấp nhất của hai đỉnh của một cây có gốc là đỉnh thấp nhất mà cây con của nó chứa cả hai đỉnh. 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:



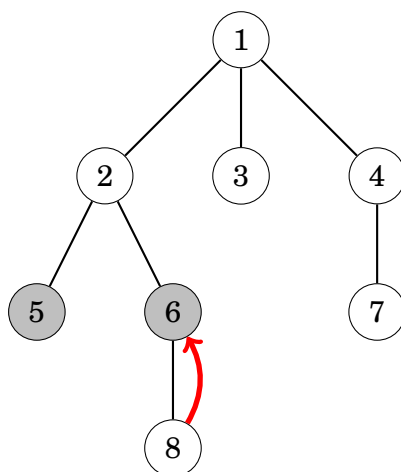
Tiếp theo, chúng ta sẽ thảo luận về hai kỹ thuật hiệu quả để tìm tổ tiên chung thấp nhất của hai đỉnh.

Phương pháp 1

Một cách để giải quyết vấn đề là sử dụng thực tế rằng chúng ta có thể tìm thấy tổ tiên thứ k của bất kỳ đỉnh nào trong cây một cách hiệu quả. Sử dụng điều này, chúng ta có thể chia vấn đề tìm tổ tiên chung thấp nhất thành hai phần.

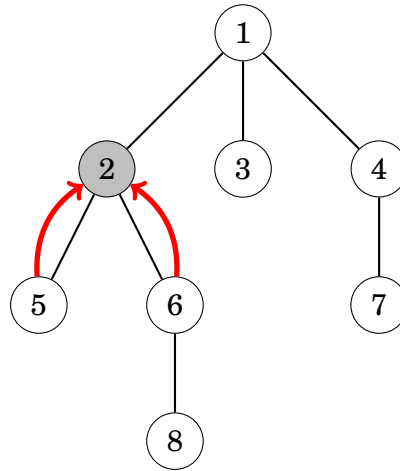
Chúng ta sử dụng hai con trỏ ban đầu trỏ đến hai đỉnh mà tổ tiên chung thấp nhất của chúng ta nên tìm. Đầu tiên, chúng ta di chuyển một trong các con trỏ lên trên để cả hai con trỏ đều trỏ đến các đỉnh ở cùng một cấp độ.

Trong kịch bản ví dụ, chúng ta di chuyển con trỏ thứ hai lên một cấp độ để nó trỏ đến đỉnh 6 nằm ở cùng cấp độ với đỉnh 5:



Sau đó, chúng ta xác định số bước tối thiểu cần thiết để di chuyển cả hai con trỏ lên trên để chúng sẽ trỏ đến cùng một đỉnh. Đỉnh mà các con trỏ trỏ đến sau đó chính là tổ tiên chung thấp nhất.

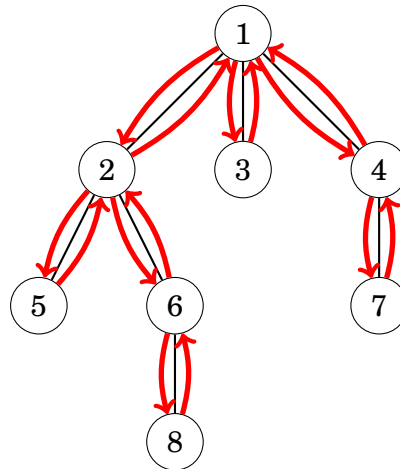
Trong kịch bản ví dụ, chỉ cần di chuyển cả hai con trỏ một bước lên trên đến đỉnh 2, đó là tổ tiên chung thấp nhất:



Vì cả hai phần của thuật toán có thể được thực hiện trong thời gian $O(\log n)$ bằng cách sử dụng thông tin đã được tính trước, chúng 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 quyết vấn đề dựa trên một mảng duyệt cây¹. Một lần nữa, ý tưởng là duyệt qua các đỉnh bằng cách sử dụng một tìm kiếm theo chiều sâu:



Tuy nhiên, chúng ta sử dụng một mảng duyệt cây khác với trước: chúng ta thêm mỗi đỉnh vào mảng *luôn luôn* khi tìm kiếm theo chiều sâu đi qua đỉnh đó, và không chỉ trong lần truy cập 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.

Chúng ta lưu trữ hai giá trị trong mảng: định danh của đỉnh và độ sâu của đỉnh trong cây. Mảng sau đây tương ứng với cây trên:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
định danh 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

¹Thuật toán tổ tiên chung thấp nhất này đã được trình bày trong [7]. Kỹ thuật này đôi khi được gọi là **kỹ thuật tour Euler** [66].

Bây giờ chúng 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 đầu tiên của a và b trong mảng. Ví dụ, tổ tiên chung thấp nhất của các đỉnh 5 và 8 có thể được tìm thấy như sau:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
định danh 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 nằm ở vị trí 2, đỉnh 8 nằm ở vị trí 5, và đỉnh có độ sâu tối thiểu giữa các vị trí 2...5 là đỉnh 2 ở vị trí 3 có độ sâu là 2. Do đó, tổ tiên chung thấp nhất của các đỉnh 5 và 8 là đỉnh 2.

Do đó, để tìm tổ tiên chung thấp nhất của hai đỉnh, chỉ cần thực hiện một truy vấn tìm kiếm tối thiểu trên khoảng. Vì mảng là tĩnh, chúng ta có thể xử lý các truy vấn như vậy trong thời gian $O(1)$ sau khi tiền xử lý trong thời gian $O(n \log n)$.

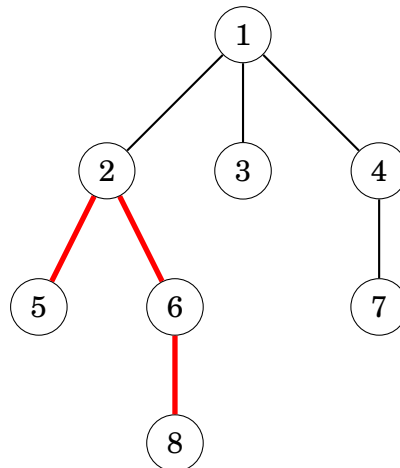
Khoảng cách giữa các đỉnh

Khoảng cách giữa các đỉnh a và b bằng với độ dài của đường đi từ a đến b . Hóa ra rằng vấn đề tính toán khoảng cách giữa các đỉnh giảm thành tìm tổ tiên chung thấp nhất của chúng.

Đầu tiên, chúng ta gán một 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 $\text{depth}(s)$ biểu thị độ sâu của đỉnh s . Ví dụ, hãy xem xét khoảng cách giữa các đỉnh 5 và 8:



Tổ tiên chung thấp nhất của các đỉnh 5 và 8 là đỉnh 2. Các 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 ngoại tuyến

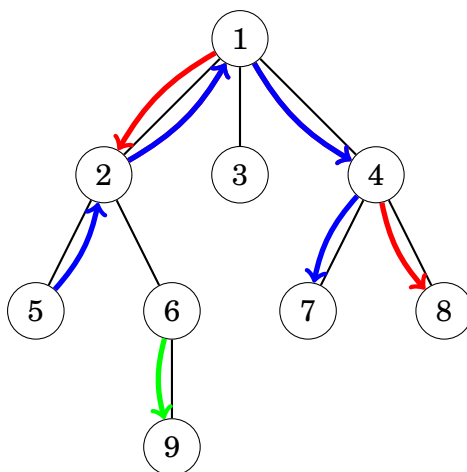
Cho đến nay, chúng ta đã thảo luận về các thuật toán *trực tuyến* cho các truy vấn trên cây. Các thuật toán đó có khả năng xử lý các truy vấn lần lượt để mỗi truy vấn được trả lời trước khi nhận truy vấn tiếp theo.

Tuy nhiên, trong nhiều bài toán, tính chất trực tuyến là không cần thiết. Trong phần này, chúng ta tập trung vào các thuật toán *ngoại tuyến*. Các thuật toán đó được cung cấp một tập hợp các truy vấn mà có thể được trả lời theo bất kỳ thứ tự nào. Thường thì dễ dàng hơn để thiết kế một thuật toán ngoại tuyến so với một thuật toán trực tuyến.

Gộp cấu trúc dữ liệu

Một phương pháp để xây dựng một thuật toán ngoại tuyến là thực hiện một duyệt cây theo chiều sâu và duy trì các cấu trúc dữ liệu trong các đỉnh. Tại mỗi đỉnh s , chúng ta tạo ra một cấu trúc dữ liệu $d[s]$ dựa trên các cấu trúc dữ liệu của các con của s . Sau đó, bằng cách sử dụng cấu trúc dữ liệu này, tất cả các truy vấn liên quan đến s được xử lý.

Như một ví dụ, hãy xem xét bài toán sau: cho một cây và một tập hợp các đường đi, nhiệm vụ của chúng ta là tìm số lượng đường đi tối đa giao nhau tại một đỉnh. Ví dụ, hãy xem xét cây sau và các đường đi $1 \rightarrow 8$, $5 \rightarrow 7$ và $6 \rightarrow 9$:



Trong trường hợp này, câu trả lời là 2, vì các đường đi $1 \rightarrow 8$ và $5 \rightarrow 7$ giao nhau tại các đỉnh 1, 2 và 4.

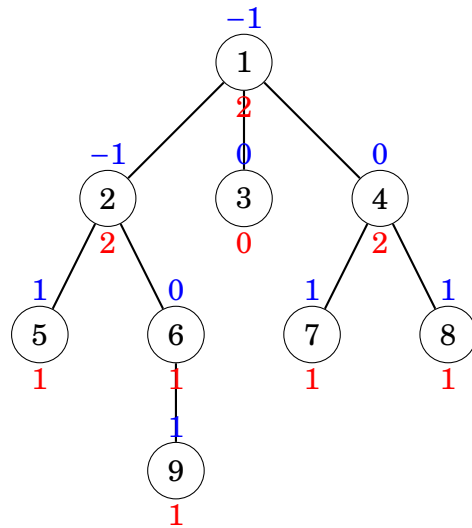
Chúng ta có thể giải quyết bài toán này bằng cách sử dụng một thuật toán ngoại tuyến. Ý tưởng là biểu diễn mỗi đường đi từ a đến b bằng cách thêm $+1$ tại các đỉnh a và b và -1 tại tổ tiên chung thấp nhất của a và b và (nếu có) cha của nó. Sau đó, đối với mỗi đỉnh s , tổng các giá trị trong cây con của nó cho biết có bao nhiêu đường đi đi qua s . Câu trả lời cho bài toán là tổng tối đa.

Ví dụ, trong cây trên, tổ tiên chung thấp nhất của 1 và 8 là 1, tổ tiên chung thấp nhất của 5 và 7 là 1, và tổ tiên chung thấp nhất của 6 và 9 là 6. Do đó, các giá trị sau được thêm vào cây:

- đường đi $1 \rightarrow 8$: $+1$ tại 1 và 8, -1 tại 1
- đường đi $5 \rightarrow 7$: $+1$ tại 5 và 7, -1 tại 1
- đường đi $6 \rightarrow 9$: $+1$ tại 6 và 9, -1 tại 6 và 2

Lưu ý rằng vì đỉnh 1 là gốc, nó không có cha. Ngoài ra, chúng ta thêm hai lần -1 cho đường đi $6 \rightarrow 9$ vì tổ tiên chung thấp nhất (6) không phải là điểm cuối.

Các giá trị cuối cùng trong cây và các tổng cây con là như sau:



Các tổng cây con tối đa là 2, đó là câu trả lời.

Chương 19

Đường đi và chu trình

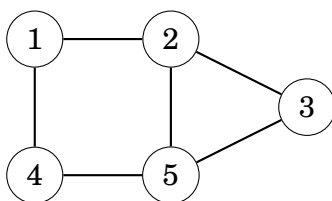
Chương này tập trung vào hai loại đường đi trong đồ thị:

- Một **đường đi Euler** là một đường đi đi qua mỗi cạnh đúng một lần.
- Một **đường đi Hamilton** là một đường đi thăm mỗi đỉnh đúng một lần.

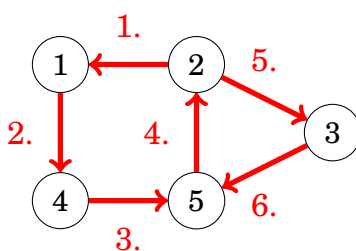
Mặc dù đường đi Euler và Hamilton trông giống như các khái niệm tương tự thoát nhìn, các bài toán tính toán liên quan đến chúng rất khác nhau. Hóa ra có một quy tắc đơn giản xác định xem một đồ thị có chứa đường đi Euler hay không, và cũng có một thuật toán hiệu quả để tìm một đường đi như vậy nếu nó tồn tại. Ngược lại, việc kiểm tra sự tồn tại của một đường đi Hamilton là một bài toán NP-khó, và không có thuật toán hiệu quả nào được biết để giải quyết bài toán này.

19.1 Đường đi Euler

Một **đường đi Euler**¹ là một đường đi đi qua đúng một lần qua mỗi cạnh của đồ thị. Ví dụ, đồ thị

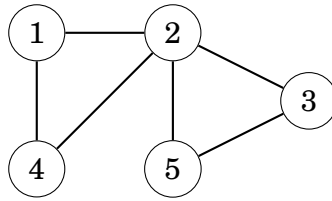


có một đường đi Euler từ đỉnh 2 đến đỉnh 5:

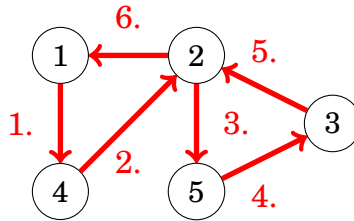


Một **chu trình Euler** là một đường đi Euler bắt đầu và kết thúc tại cùng một đỉnh. Ví dụ, đồ thị

¹L. Euler đã nghiên cứu những đường đi như vậy vào năm 1736 khi ông giải quyết bài toán bảy cây cầu nổi tiếng ở Königsberg. Đây là sự ra đời của lý thuyết đồ thị.



có một chu trình Euler bắt đầu và kết thúc tại đỉnh 1:



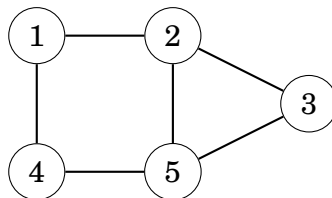
Sự tồn tại

Sự tồn tại của đường đi và chu trình Euler phụ thuộc vào bậc của các đỉnh. Đầu tiên, một đồ thị vô hướng có một đường đi Euler chính xác khi tất cả các cạnh thuộc cùng một thành phần liên thông và

- bậc của mỗi đỉnh là chẵn *hoặc*
- bậc của đúng hai đỉnh là lẻ, và bậc của tất cả các đỉnh khác là chẵn.

Trong trường hợp đầu tiên, mỗi đường đi Euler cũng là một chu trình Euler. Trong trường hợp thứ hai, các đỉnh bậc lẻ là các đỉnh bắt đầu và kết thúc của một đường đi Euler không phải là một chu trình Euler.

Ví dụ, trong đồ thị



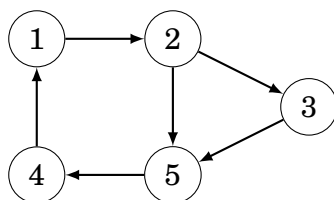
các đỉnh 1, 3 và 4 có bậc là 2, và các đỉnh 2 và 5 có bậc là 3. Chính xác hai đỉnh có bậc lẻ, vì vậy có một đường đi Euler giữa các đỉnh 2 và 5, nhưng đồ thị không chứa một chu trình Euler.

Trong một đồ thị có hướng, chúng ta tập trung vào bậc vào và bậc ra của các đỉnh. Một đồ thị có hướng chứa một đường đi Euler chính xác khi tất cả các cạnh thuộc cùng một thành phần liên thông và

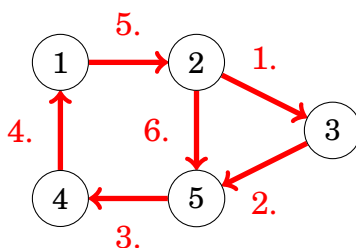
- ở mỗi đỉnh, bậc vào bằng bậc ra, *hoặc*
- ở một đỉnh, bậc vào lớn hơn bậc ra một, ở một đỉnh khác, bậc ra lớn hơn bậc vào một, và ở tất cả các đỉnh khác, bậc vào bằng bậc ra.

Trong trường hợp đầu tiên, mỗi đường đi Euler cũng là một chu trình Euler, và 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, và 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 của Hierholzer

Thuật toán của Hierholzer² là một phương pháp hiệu quả để xây dựng một chu trình Euler. Thuật toán bao gồm nhiều vòng, mỗi vòng thêm các cạnh mới vào chu trình. Tất nhiên, chúng ta giả định rằng đồ thị chứa một chu trình Euler; nếu không thì thuật toán của Hierholzer không thể tìm thấy nó.

Đầu tiên, thuật toán xây dựng một chu trình chứa một số (không nhất thiết là tất cả) các cạnh 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 tất cả các cạnh đã được thêm vào chu trình.

Thuật toán mở rộng chu trình bằng cách luôn tìm một đỉnh x thuộc chu trình nhưng có một cạnh đi ra không được bao gồm trong chu trình. Thuật toán xây dựng một đường đi mới từ đỉnh x chỉ chứa các cạnh chưa có trong chu trình. Sớm hay muộn, đường đi sẽ quay trở lại đỉnh x , tạo ra một chu trình con.

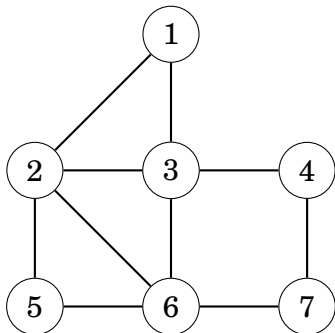
Nếu đồ thị chỉ chứa một đường đi Euler, chúng ta vẫn có thể sử dụng thuật toán của Hierholzer để tìm nó bằng cách thêm một cạnh phụ vào đồ thị và loại bỏ cạnh đó sau khi chu trình đã được xây dựng. Ví dụ, trong một đồ thị vô hướng, chúng ta thêm cạnh phụ giữa hai đỉnh bậc lẻ.

Tiếp theo chúng ta sẽ xem thuật toán của Hierholzer xây dựng một chu trình Euler cho một đồ thị vô hướng như thế nào.

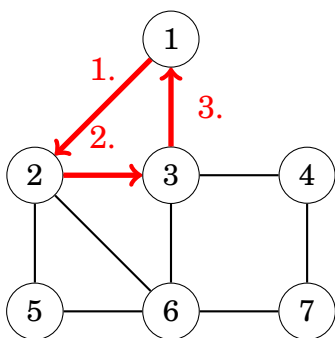
²Thuật toán được công bố vào năm 1873 sau khi Hierholzer qua đời [35].

Ví dụ

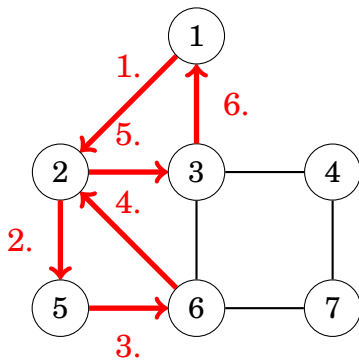
Hãy xem xét đồ thị sau:



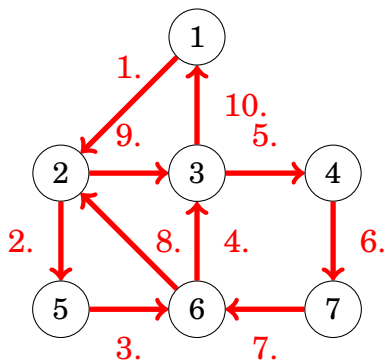
Giả sử rằng thuật toán đầu tiên tạo ra một chu trình bắt đầu tại đỉnh 1. Một chu trình có thể là $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



Sau đó, thuật toán thêm chu trình con $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ vào chu trình:



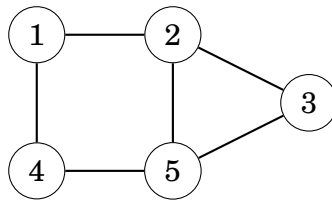
Cuối cùng, thuật toán thêm chu trình con $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ vào chu trình:



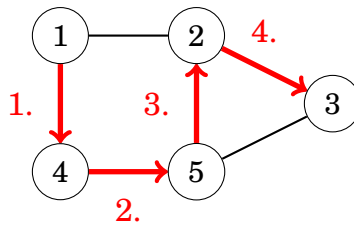
Bây giờ tất cả các cạnh đã được bao gồm trong chu trình, vì vậy chúng ta đã xây dựng thành công một chu trình Euler.

19.2 Đường đi Hamilton

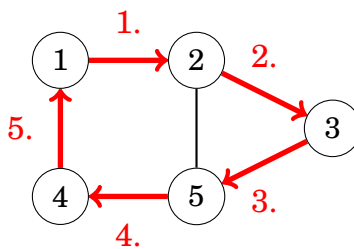
Một **đường đi Hamilton** là một đường đi thăm mỗi đỉnh của đồ thị đúng một lần. Ví dụ, đồ thị



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 một **chu trình Hamilton**. Đồ thị trên cũng có một chu trình Hamilton bắt đầu và kết thúc tại đỉnh 1:



Sự tồn tại

Không có phương pháp hiệu quả nào được biết để kiểm tra xem một đồ thị có chứa một đường đi Hamilton hay không, và bài toán này là NP-khó. Tuy nhiên, trong một số trường hợp đặc biệt, chúng ta có thể chắc chắn rằng một đồ thị chứa một đường đi Hamilton.

Một quan sát đơn giản là nếu đồ thị là đầy đủ, tức là, có một cạnh giữa tất cả các cặp đỉnh, nó cũng chứa một đường đi Hamilton. Các kết quả mạnh hơn cũng đã được đạt được:

- **Định lý Dirac:** Nếu bậc của mỗi đỉnh ít nhất là $n/2$, đồ thị chứa một đường đi Hamilton.
- **Định lý Ore:** Nếu tổng bậc của mỗi cặp đỉnh không kề nhau ít nhất là n , đồ thị chứa một đường đi Hamilton.

Một thuộc tính chung trong 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 *một số lượng lớn* các cạnh. Điều này có lý, bởi vì càng có nhiều cạnh trong đồ thị, càng có nhiều khả năng để xây dựng một đường đi Hamilton.

Xây dựng

Vì không có cách hiệu quả để kiểm tra xem một đường đi Hamilton có tồn tại hay không, rõ ràng là cũng không có phương pháp nào để xây dựng đường đi một cách hiệu quả, bởi vì nếu không chúng ta có thể chỉ cần thử xây dựng đường đi và xem liệu nó có tồn tại hay không.

Một cách đơn giản để tìm kiếm một đường đi Hamilton là sử dụng một thuật toán quay lui đi qua tất cả các cách có thể để xây dựng đường đi. Độ phức tạp thời gian của một thuật toán như vậy ít nhất là $O(n!)$, bởi vì có $n!$ cách khác nhau để chọn thứ tự của n đỉnh.

Một giải pháp hiệu quả hơn dựa trên quy hoạch động (xem Chương 10.5). Ý tưởng là tính toán các giá trị của một hàm $\text{possible}(S, x)$, trong đó S là một tập hợp con các đỉnh và x là một trong các đỉnh. Hàm này cho biết liệu có một đường đi Hamilton thăm các đỉnh của S và kết thúc tại đỉnh x hay không. Có thể triển khai giải pháp này trong thời gian $O(2^n n^2)$.

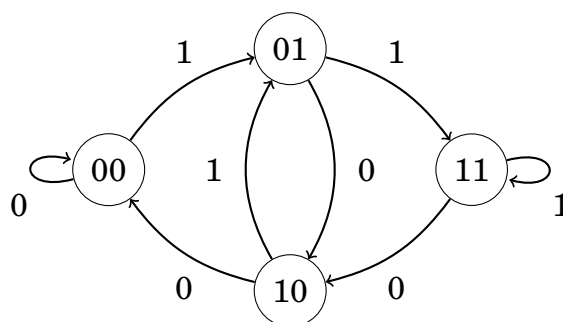
19.3 Chuỗi De Bruijn

Một **chuỗi De Bruijn** là một chuỗi chứa mọi chuỗi có độ dài n đúng một lần dưới dạng chuỗi con, cho một bảng chữ cái cố định gồm k ký tự. Độ dài của một chuỗi như vậy là $k^n + n - 1$ ký tự. Ví dụ, khi $n = 3$ và $k = 2$, một ví dụ về chuỗi De Bruijn là

0001011100.

Các chuỗi con của chuỗi này là tất cả các kết hợp của ba bit: 000, 001, 010, 011, 100, 101, 110 và 111.

Hóa ra mỗi chuỗi 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 chuỗi gồm $n - 1$ ký tự và mỗi cạnh thêm một ký tự vào chuỗi. Đồ thị sau đây tương ứng với kịch bản trên:



Một đường đi Euler trong đồ thị này tương ứng với một chuỗi chứa tất cả các chuỗi có độ dài n . Chuỗi này chứa các ký tự của đỉnh bắt đầu và tất cả các ký tự của các cạnh. Đỉnh bắt đầu có $n - 1$ ký tự và có k^n ký tự trong các cạnh, vì vậy độ dài của chuỗi là $k^n + n - 1$.

19.4 Hành trình của quân mã

Một **hành trình của quân mã** là một chuỗi các nước đi của một quân mã trên một bàn cờ $n \times n$ theo các quy tắc của cờ vua sao cho quân mã thăm mỗi ô đúng một lần. Một hành trình của quân mã được gọi là một *hành trình đóng* nếu quân mã cuối cùng quay trở lại ô xuất phát và ngược lại nó được gọi là một *hành trình mở*.

Ví dụ, đây là một hành trình mở của quân mã trên bàn cờ 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 của quân mã tương ứng với một đường đi Hamilton trong một đồ thị có các đỉnh đại diện cho các ô của bàn cờ, và hai đỉnh được nối với nhau bằng một cạnh nếu một quân mã có thể di chuyển giữa các ô theo các quy tắc của cờ vua.

Một cách tự nhiên để xây dựng một hành trình của quân mã là sử dụng thuật toán quay lui. Việc tìm kiếm có thể được thực hiện hiệu quả hơn bằng cách sử dụng *heuristic* cố gắng hướng dẫn quân mã để một hành trình hoàn chỉnh sẽ được tìm thấy nhanh chóng.

Quy tắc của Warnsdorf

Quy tắc của Warnsdorf là một heuristic đơn giản và hiệu quả để tìm một hành trình của quân mã³. Sử dụng quy tắc này, có thể xây dựng một hành trình một cách hiệu quả ngay cả trên một bàn cờ lớn. Ý tưởng là luôn di chuyển quân mã sao cho nó kết thúc ở một ô mà số lượng nước đi có thể là *nhỏ nhất* có thể.

Ví dụ, trong tình huống sau, có năm ô có thể mà quân mã có thể di chuyển đến (các ô $a \dots e$):

1				a
		2		
b				e
	c		d	

Trong tình huống này, quy tắc của Warnsdorf di chuyển quân mã đến ô a , bởi vì sau lựa chọn này, chỉ có một nước đi duy nhất có thể. Các lựa chọn khác sẽ di chuyển quân mã đến các ô nơi sẽ có ba nước đi có sẵn.

³Heuristic này được đề xuất trong cuốn sách của Warnsdorf [69] vào năm 1823. Cũng có các thuật toán đa thức để tìm hành trình của quân mã [52], nhưng chúng phức tạp hơn.

Chương 20

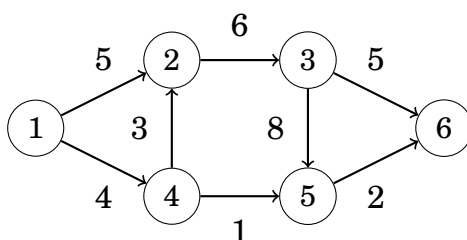
Luồng và lát cắt (Flows and cuts)

Trong chương này, chúng ta tập trung vào hai bài toán sau:

- **Finding a maximum flow:** Lượng luồng tối đa chúng ta có thể gửi từ một nút đến một nút khác là bao nhiêu?
- **Finding a minimum cut:** Tập hợp các cạnh có trọng số tối thiểu để tách hai nút của đồ thị là gì?

Đầu vào cho cả hai bài toán này là một đồ thị có hướng, có trọng số, chứa hai nút đặc biệt: *nguồn* (*source*) là một nút không có cạnh đi vào, và *đích* (*sink*) là một nút không có cạnh đi ra.

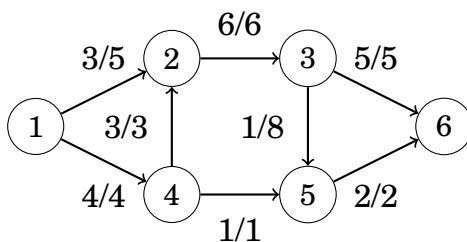
Ví dụ, chúng ta sẽ sử dụng đồ thị sau trong đó nút 1 là nguồn và nút 6 là đích:



Maximum flow

Trong bài toán **maximum flow**, nhiệm vụ của chúng ta là gửi càng nhiều luồng càng tốt từ nguồn đến đích. Trọng số của mỗi cạnh là một sức chứa (capacity) giới hạn luồng có thể đi qua cạnh đó. Ở mỗi nút trung gian, luồng đi vào và luồng đi ra phải bằng nhau.

Ví dụ, kích thước tối đa của một luồng trong đồ thị ví dụ là 7. Hình sau cho thấy cách chúng ta có thể định tuyến luồng:

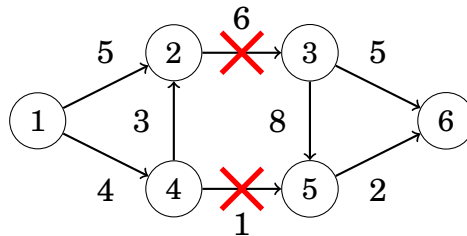


Ký hiệu v/k có nghĩa là một luồng v đơn vị được định tuyến qua một cạnh có sức chứa là k đơn vị. Kích thước của luồng là 7, bởi vì nguồn gửi đi $3 + 4$ đơn vị luồng và đích nhận $5 + 2$ đơn vị luồng. Dễ thấy luồng này là cực đại, bởi vì tổng sức chứa của các cạnh dẫn đến đích là 7.

Minimum cut

Trong bài toán **minimum cut**, nhiệm vụ của chúng ta là loại bỏ một tập hợp các cạnh khỏi đồ thị sao cho không còn đường đi từ nguồn đến đích sau khi loại bỏ và tổng trọng số của các cạnh bị loại bỏ là tối thiểu.

Kích thước tối thiểu của một lát cắt trong đồ thị ví dụ là 7. Chỉ cần loại bỏ các cạnh $2 \rightarrow 3$ và $4 \rightarrow 5$:



Sau khi loại bỏ các cạnh, sẽ không còn đường đi nào từ nguồn đến đích. Kích thước của lát cắt là 7, bởi vì trọng số của các cạnh bị loại bỏ là 6 và 1. Lát cắt này là tối thiểu, vì không có cách hợp lệ nào để loại bỏ các cạnh khỏi đồ thị sao cho tổng trọng số của chúng nhỏ hơn 7.

Không phải ngẫu nhiên mà kích thước tối đa của một luồng và kích thước tối thiểu của một lát cắt lại bằng nhau trong ví dụ trên. Hóa ra luồng cực đại và lát cắt cực tiểu *luôn luôn* có độ lớn bằng nhau, vì vậy các khái niệm này là hai mặt của cùng một đồng xu.

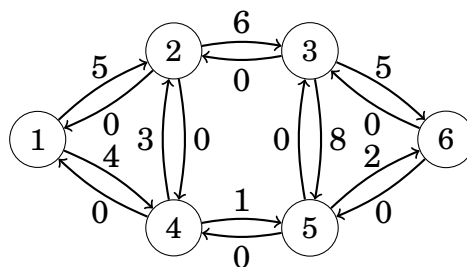
Tiếp theo chúng ta sẽ thảo luận về thuật toán Ford–Fulkerson có thể được sử dụng để tìm luồng cực đại và lát cắt cực tiểu của một đồ thị. Thuật toán này cũng giúp chúng ta hiểu *tại sao* chúng lại có độ lớn bằng nhau.

20.1 Thuật toán Ford–Fulkerson

Thuật toán Ford–Fulkerson (Ford–Fulkerson algorithm) [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 ra nhiều luồng hơn. Cuối cùng, khi thuật toán không thể tăng luồng được nữa, luồng cực đại đã được tìm thấy.

Thuật toán sử dụng một biểu diễn đặc biệt của đồ thị, trong đó mỗi cạnh ban đầu có một cạnh ngược theo hướng khác. Trọng số của mỗi cạnh cho biết chúng ta có thể định tuyến thêm bao nhiêu luồng qua nó. Khi bắt đầu thuật toán, trọng số của mỗi cạnh ban đầu bằng sức chứa của cạnh đó và trọng số của mỗi cạnh ngược là không.

Biểu diễn mới cho đồ thị ví dụ như sau:

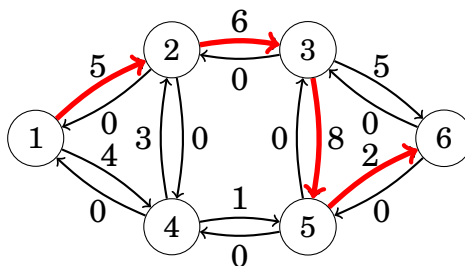


Mô tả thuật toán

Thuật toán Ford–Fulkerson bao 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 hơn

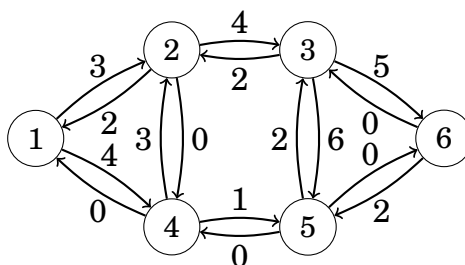
một đường đi khả dụng, chúng ta có thể chọn bất kỳ đường nào trong số đó.

Ví dụ, giả sử chúng ta chọn đường đi sau:



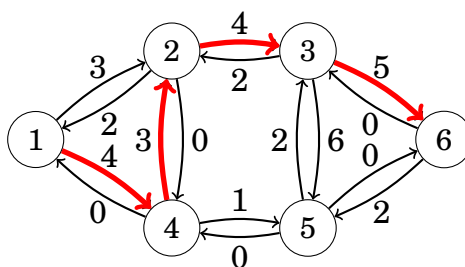
Sau khi chọn đường đi, luồng tăng thêm x đơn vị, trong đó x là trọng số cạnh nhỏ nhất trên đường đi. Ngoài ra, trọng số của mỗi cạnh trên đường đi giảm đi x và trọng số của mỗi cạnh ngược tăng thêm x .

Trong đường đi trên, trọng số của các cạnh là 5, 6, 8 và 2. Trọng số nhỏ nhất là 2, vì vậy luồng tăng thêm 2 và đồ thị mới như sau:



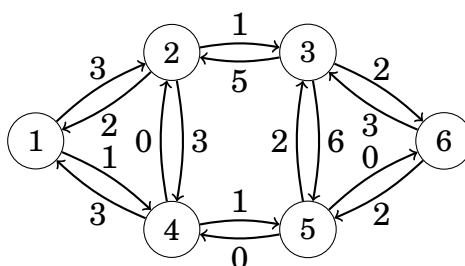
Ý tưởng là việc tăng luồng sẽ làm giảm lượng luồng có thể đi qua các cạnh trong tương lai. Mặt khác, có thể hủy luồng sau này bằng cách sử dụng các cạnh ngược của đồ thị nếu hóa ra việc định tuyến luồng theo một cách khác sẽ có lợi hơn.

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 của chúng ta có thể như sau:



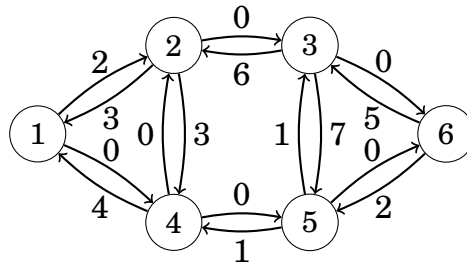
Trọng số cạnh nhỏ nhất trên đường đi này là 3, vì vậy đường đi này tăng luồng thêm 3, và tổng luồng sau khi xử lý đường đi là 5.

Đồ thị mới sẽ như sau:



Chúng ta vẫn cần thêm hai vòng nữa trước khi đạt được luồng cực đại. Ví dụ, chúng

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 tăng luồng thêm 1, và đồ thị cuối cùng như sau:



Không thể tăng luồng thêm nữa, vì không còn đường đi nào từ nguồn đến đích có trọng số cạnh dương. Do đó, thuật toán kết thúc và luồng cực đại là 7.

Tìm đường đi

Thuật toán Ford–Fulkerson không chỉ định cách chúng ta nên chọn các đường đi để tăng luồng. Trong mọi trường hợp, thuật toán sẽ kết thúc sớm hay muộn và tìm đúng luồng cực đại. Tuy nhiên, hiệu quả của thuật toán phụ thuộc vào cách các đường đi được chọn.

Một cách đơn giản để tìm đường đi là sử dụng tìm kiếm theo chiều sâu. Thông thường, cách này hoạt động tốt, nhưng trong trường hợp xấu nhất, mỗi đường đi chỉ tăng luồng thêm 1 và thuật toán sẽ chậm. May mắn thay, chúng ta có thể tránh tình huống này bằng cách sử dụng một trong các kỹ thuật sau:

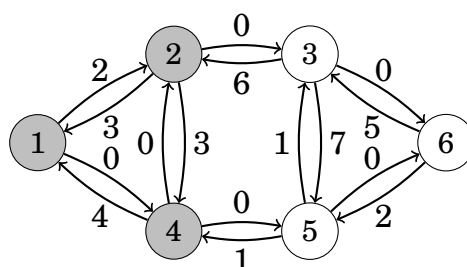
Thuật toán Edmonds–Karp (Edmonds–Karp algorithm) [18] chọn mỗi đường đi sao cho số lượng cạnh trên đường đi là nhỏ nhất có thể. Điều này có thể được thực hiện bằng cách sử dụng tìm kiếm theo chiều rộng thay vì tìm kiếm theo chiều sâu để tìm đường đi. Có thể chứng minh rằng điều này đảm bảo luồng tăng nhanh, và độ phức tạp thời gian của thuật toán là $O(m^2n)$.

Thuật toán co giãn (scaling algorithm) [2] sử dụng tìm kiếm theo chiều sâu để tìm các đường đi trong đó mỗi trọng số cạnh ít nhất bằng một giá trị ngưỡng. Ban đầu, giá trị ngưỡng là một số lớn nào đó, ví dụ như tổng tất cả trọng số cạnh của đồ thị. Mỗi khi không tìm thấy 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 co giãn dễ thực hiện hơn, vì có thể sử dụng 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 các cuộc thi lập trình.

Minimum cuts

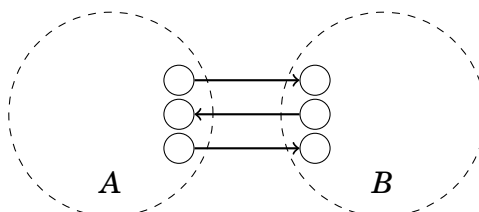
Hóa ra một khi thuật toán Ford–Fulkerson đã tìm thấy một luồng cực đại, nó cũng đã xác định được một lát cắt cực tiểu. Gọi A là tập hợp các nút có thể đến được từ nguồn bằng cách sử dụng các cạnh có trọng số dương. Trong đồ thị ví dụ, A chứa các nút 1, 2 và 4:



Bây giờ lát cắt cực tiểu bao gồm các cạnh của đồ thị ban đầu bắt đầu tại một nút nào đó trong A , kết thúc tại một nút nào đó bên ngoài A , và có sức chứa được sử dụng toàn bộ trong luồng cực đại. Trong đồ thị trên, các cạnh đó là $2 \rightarrow 3$ và $4 \rightarrow 5$, tương ứng với lát cắt cực tiểu $6 + 1 = 7$.

Tại sao luồng do thuật toán tạo ra là cực đại và tại sao lát cắt là cực tiểu? Lý do là một đồ thị không thể chứa một luồng có kích thước lớn hơn trọng số của bất kỳ lát cắt nào của đồ thị. Do đó, bất cứ khi nào một luồng và một lát cắt có độ lớn bằng nhau, chúng là một luồng cực đại và một lát cắt cực tiểu.

Hãy xem xét bất kỳ lát cắt nào của đồ thị sao cho nguồn thuộc A , đích thuộc B và có một số cạnh giữa các tập hợp:



Kích thước của lát cắt là tổng của các cạnh đi từ A đến B . Đây là một giới hạn trên cho luồng trong đồ thị, bởi vì luồng phải đi từ A đến B . Do đó, kích thước của một luồng cực đại nhỏ hơn hoặc bằng kích thước của bất kỳ lát cắt nào trong đồ thị.

Mặt khác, thuật toán Ford–Fulkerson tạo ra một luồng có kích thước *chính xác* bằng kích thước của một lát cắt trong đồ thị. Do đó, luồng phải là luồng cực đại và lát cắt phải là lát cắt cực tiểu.

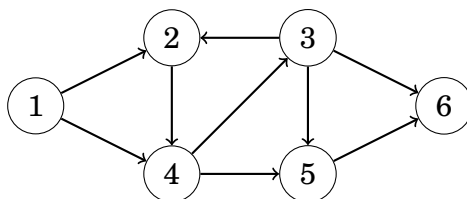
20.2 Đường đi không giao nhau (Disjoint paths)

Nhiều bài toán đồ thị có thể được giải quyết bằng cách quy chúng về bài toán luồng cực đại. Ví dụ đầu tiên của chúng ta về một bài toán như vậy là như sau: chúng ta được cho một đồ thị có hướng với một nguồn và một đích, và nhiệm vụ của chúng ta là tìm số lượng tối đa các đường đi không giao nhau từ nguồn đến đích.

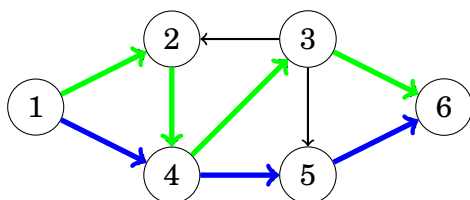
Edge-disjoint paths

Đầu tiên chúng ta sẽ tập trung vào bài toán tìm số lượng tối đa các **đường đi không giao nhau trên cạnh (edge-disjoint paths)** từ nguồn đến đích. Điều này có nghĩa là chúng ta nên xây dựng một tập hợp các đường đi sao cho mỗi cạnh xuất hiện trong tối đa một đường đi.

Ví dụ, hãy xem xét đồ thị sau:



Trong đồ thị này, số lượng tối đa các đường đi không giao nhau trên cạnh là 2. Chúng 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:

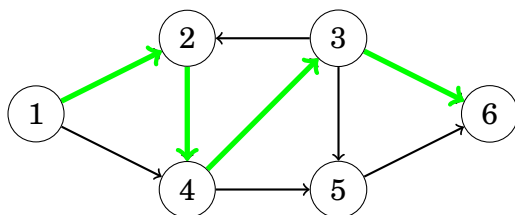


Hóa ra số lượng tối đa các đường đi không giao nhau trên cạnh bằng luồng cực đại của đồ thị, giả sử sức chứa của mỗi cạnh là một. Sau khi luồng cực đại đã được xây dựng, các đường đi không giao nhau trên cạnh có thể được tìm thấy một cách tham lam bằng cách đi theo các đường đi từ nguồn đến đích.

Node-disjoint paths

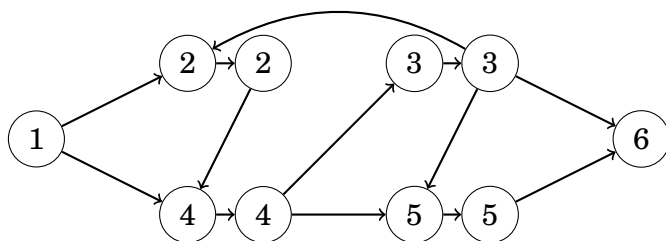
Bây giờ chúng ta hãy xem xét một bài toán khác: tìm số lượng tối đa các **đường đi không giao nhau trên đỉnh (node-disjoint paths)** từ nguồn đến đích. Trong bài toán này, mỗi nút, ngoại trừ nguồn và đích, có thể xuất hiện trong tối đa một đường đi. Số lượng đường đi không giao nhau trên đỉnh có thể nhỏ hơn số lượng đường đi không giao nhau trên cạnh.

Ví dụ, trong đồ thị trước, số lượng tối đa các đường đi không giao nhau trên đỉnh là 1:

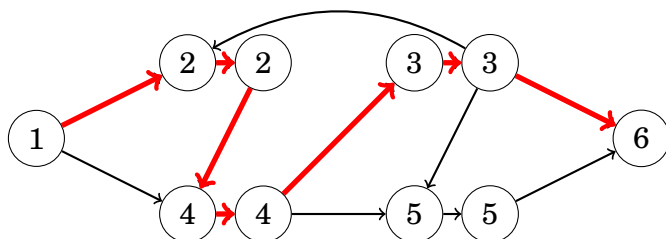


Chúng ta cũng có thể quy bài toán này về bài toán luồng cực đại. Vì mỗi nút có thể xuất hiện trong tối đa một đường đi, chúng ta phải giới hạn luồng đi qua các nút. Một phương pháp tiêu chuẩn cho việc này là chia mỗi nút thành hai nút sao cho nút thứ nhất có các cạnh đi vào của nút ban đầu, nút thứ hai có các cạnh đi ra của nút ban đầu, và có một cạnh mới từ nút thứ nhất đến nút thứ hai.

Trong ví dụ của chúng ta, đồ thị trở thành như sau:



Luồng cực đại cho đồ thị là như sau:



Do đó, số lượng tối đa các đường đi không giao nhau trên đỉnh từ nguồn đến đích là 1.

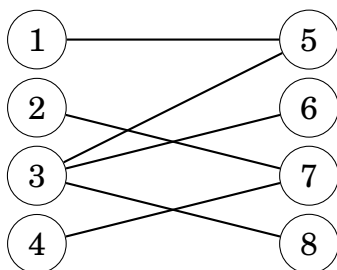
20.3 Maximum matchings

Bài toán **maximum matching** yêu cầu tìm một tập hợp các cặp nút có kích thước tối đa trong một đồ thị vô hướng sao cho mỗi cặp được nối với nhau bằng một cạnh và mỗi nút thuộc về tối đa một cặp.

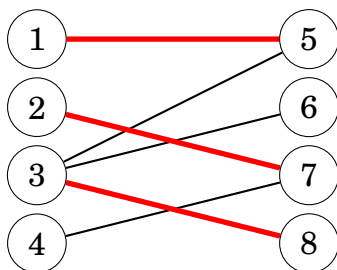
Có các thuật toán đa thức để tìm ghép cặp cực đại trong đồ thị tổng quát [17], nhưng các thuật toán như vậy rất phức tạp và hiếm khi thấy trong các cuộc thi lập trình. Tuy nhiên, trong các đồ thị hai phía, bài toán ghép cặp cực đại dễ giải quyết hơn nhiều, bởi vì chúng ta có thể quy nó về bài toán luồng cực đại.

Tìm ghép cặp cực đại

Các nút của một đồ thị hai phía luôn có thể được chia thành hai nhóm sao cho tất cả các cạnh của đồ thị đi từ nhóm bên trái sang nhóm bên phải. Ví dụ, trong đồ thị hai phía sau, các nhóm là $\{1, 2, 3, 4\}$ và $\{5, 6, 7, 8\}$.

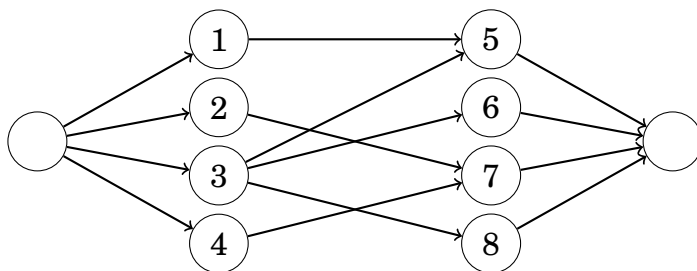


Kích thước của một ghép cặp cực đại của đồ thị này là 3:

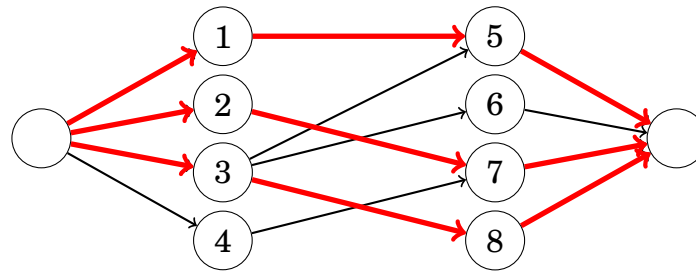


Chúng ta có thể quy bài toán ghép cặp cực đại trên đồ thị hai phía về bài toán luồng cực đại bằng cách thêm hai nút mới vào đồ thị: một nguồn và một đích. Chúng ta cũng thêm các cạnh từ nguồn đến mỗi nút bên trái và từ mỗi nút bên phải đến đích. Sau đó, kích thước của một luồng cực đại trong đồ thị bằng kích thước của một ghép cặp cực đại trong đồ thị ban đầu.

Ví dụ, việc quy đổi cho đồ thị trên như sau:



Luồng cực đại của đồ thị này như sau:

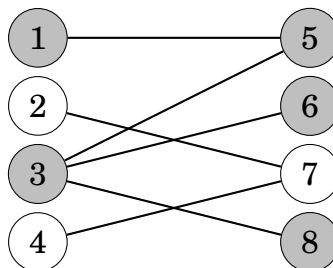


Hall's theorem

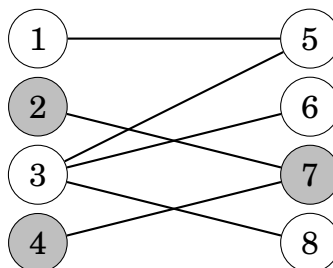
Định lý Hall (Hall's theorem) có thể được sử dụng để tìm ra liệu một đồ thị hai phía có một ghép cặp chứa tất cả các nút bên trái hoặc bên phải hay không. Nếu số lượng nút bên trái và bên phải bằng nhau, định lý Hall cho chúng ta biết liệu có thể xây dựng một **ghép cặp hoàn hảo (perfect matching)** chứa tất cả các nút của đồ thị hay không.

Giả sử chúng ta muốn tìm một ghép cặp chứa tất cả các nút bên trái. Gọi X là một tập hợp bất kỳ các nút bên trái và gọi $f(X)$ là tập hợp các hàng xóm của chúng. Theo định lý Hall, một ghép cặp chứa tất cả các nút bên trái tồn tại chính xác khi với mỗi X , điều kiện $|X| \leq |f(X)|$ thỏa mãn.

Hãy nghiên cứu định lý Hall trong đồ thị ví dụ. Đầu tiên, đặt $X = \{1, 3\}$, ta có $f(X) = \{5, 6, 8\}$:



Điều kiện của định lý Hall thỏa mãn, bởi vì $|X| = 2$ và $|f(X)| = 3$. Tiếp theo, đặt $X = \{2, 4\}$, ta có $f(X) = \{7\}$:



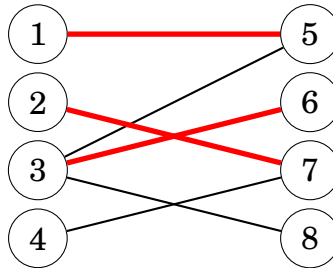
Trong trường hợp này, $|X| = 2$ và $|f(X)| = 1$, vì vậy điều kiện của định lý Hall không thỏa mãn. Điều này có nghĩa là không thể tạo thành một ghép cặp hoàn hảo cho đồ thị. Kết quả này không đáng ngạc nhiên, bởi vì chúng ta đã biết rằng ghép cặp cực đại của đồ thị là 3 chứ không phải 4.

Nếu điều kiện của định lý Hall không thỏa mãn, tập hợp X cung cấp một lời giải thích tại sao chúng ta không thể tạo thành một ghép cặp như vậy. Vì X chứa nhiều nút hơn $f(X)$, không có đủ cặp cho tất cả các nút trong X . Ví dụ, trong đồ thị trên, cả hai nút 2 và 4 đều phải được nối với nút 7, điều này là không thể.

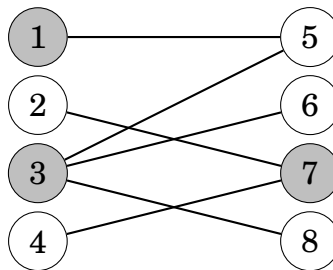
König's theorem

Một **phủ đỉnh tối thiểu (minimum node cover)** của một đồ thị là một tập hợp các nút có kích thước tối thiểu sao cho mỗi cạnh của đồ thị có ít nhất một điểm cuối trong tập hợp. Trong một đồ thị tổng quát, việc tìm một phủ đỉnh tối thiểu là một bài toán NP-khó. Tuy nhiên, nếu đồ thị là hai phía, **Định lý König (König's theorem)** cho chúng ta biết rằng kích thước của một phủ đỉnh tối thiểu và kích thước của một ghép cặp cực đại luôn bằng nhau. Do đó, chúng ta có thể tính kích thước của một phủ đỉnh tối thiểu bằng cách sử dụng một thuật toán luồng cực đại.

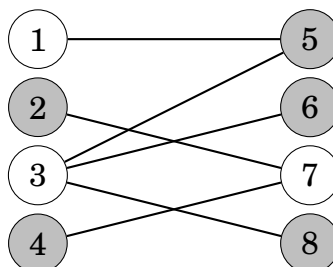
Hãy xem xét đồ thị sau với một ghép cặp cực đại có kích thước 3:



Bây giờ định lý König cho chúng ta biết rằng kích thước của một phủ đỉnh tối thiểu cũng là 3. Một phủ như vậy có thể được xây dựng như sau:



Các nút *không* thuộc về một phủ đỉnh tối thiểu tạo thành một **tập độc lập cực đại (maximum independent set)**. Đây là tập hợp các nút lớn nhất có thể sao cho không có hai nút nào trong tập hợp được nối với nhau bằng một cạnh. Một lần nữa, việc tìm một tập độc lập cực đại trong một đồ thị tổng quát là một bài toán NP-khó, nhưng trong một đồ thị hai phía, chúng ta có thể sử dụng định lý König để giải quyết bài toán một cách hiệu quả. Trong đồ thị ví dụ, tập độc lập cực đại như sau:

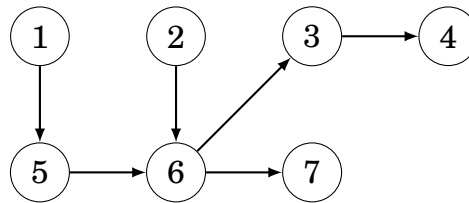


20.4 Path covers

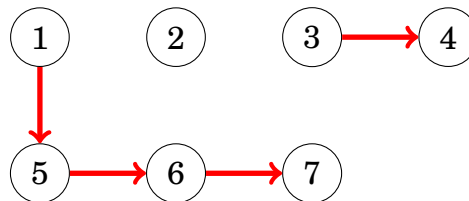
Một **phủ đường đi (path cover)** là một tập hợp các đường đi trong một đồ thị sao cho mỗi nút của đồ thị thuộc về ít nhất một đường đi. Hóa ra trong các đồ thị có hướng, không chu trình, chúng ta có thể quy bài toán tìm một phủ đường đi tối thiểu về bài toán tìm một luồng cực đại trong một đồ thị khác.

Node-disjoint path cover

Trong một **phủ đường đi không giao nhau trên đỉnh (node-disjoint path cover)**, mỗi nút thuộc về chính xác một đường đi. Ví dụ, hãy xem xét đồ thị sau:



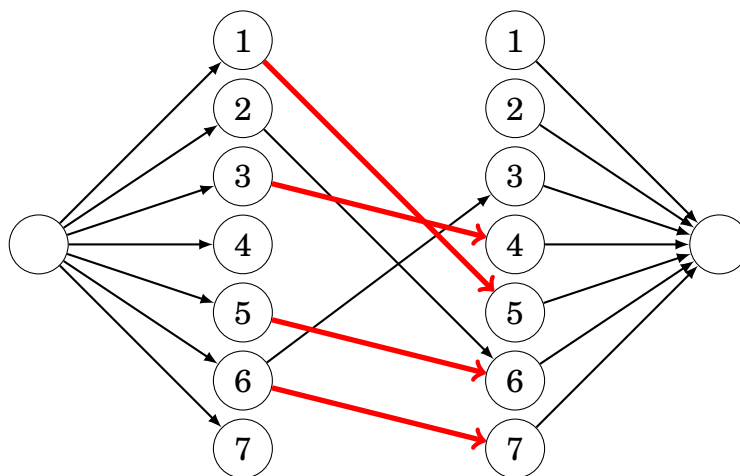
Một phủ đường đi không giao nhau trên đỉnh tối thiểu của đồ thị này bao gồm ba đường đi. Ví dụ, chúng ta có thể chọn các đường đi sau:



Lưu ý rằng một trong các đường đi chỉ chứa nút 2, vì vậy có thể một đường đi không chứa bất kỳ cạnh nào.

Chúng ta có thể tìm một phủ đường đi không giao nhau trên đỉnh tối thiểu bằng cách xây dựng một *đồ thị ghép cặp* trong đó mỗi nút của đồ thị ban đầu được biểu diễn bởi hai nút: một nút bên trái và một nút bên phải. Có một cạnh từ một nút bên trái đến một nút bên phải nếu có một cạnh như vậy trong đồ thị ban đầu. Ngoài ra, đồ thị ghép cặp chứa một nguồn và một đích, và có các cạnh từ nguồn đến tất cả các nút bên trái và từ tất cả các nút bên phải đến đích.

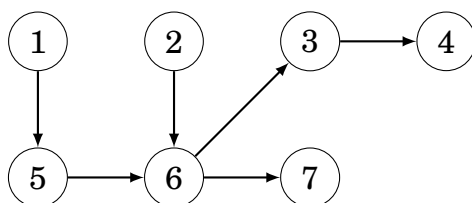
Một ghép cặp cực đại trong đồ thị kết quả tương ứng với một phủ đường đi không giao nhau trên đỉnh tối thiểu trong đồ thị ban đầu. Ví dụ, đồ thị ghép cặp sau cho đồ thị trên chứa một ghép cặp cực đại có kích thước 4:



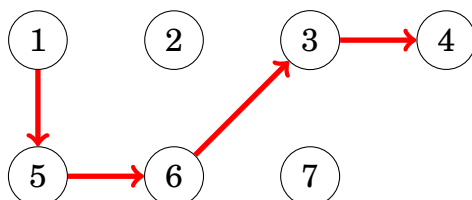
Mỗi cạnh trong ghép cặp cực đại của đồ thị ghép cặp tương ứng với một cạnh trong phủ đường đi không giao nhau trên đỉnh tối thiểu của đồ thị ban đầu. Do đó, kích thước của phủ đường đi không giao nhau trên đỉnh tối thiểu là $n - c$, trong đó n là số nút trong đồ thị ban đầu và c là kích thước của ghép cặp cực đại.

General path cover

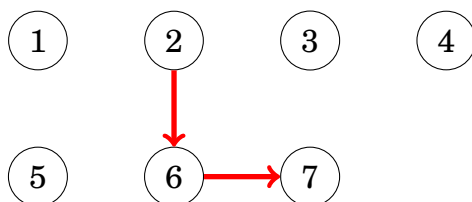
Một **phủ đường đi tổng quát (general path cover)** là một phủ đường đi trong đó một nút có thể thuộc về nhiều hơn một đường đi. Một phủ đường đi tổng quát tối thiểu có thể nhỏ hơn một phủ đường đi không giao nhau trên đỉnh tối thiểu, bởi vì một nút có thể được sử dụng nhiều lần trong các đường đi. Hãy xem xét lại đồ thị sau:



Phủ đường đi tổng quát tối thiểu của đồ thị này bao gồm hai đường đi. Ví dụ, đường đi thứ nhất có thể như sau:

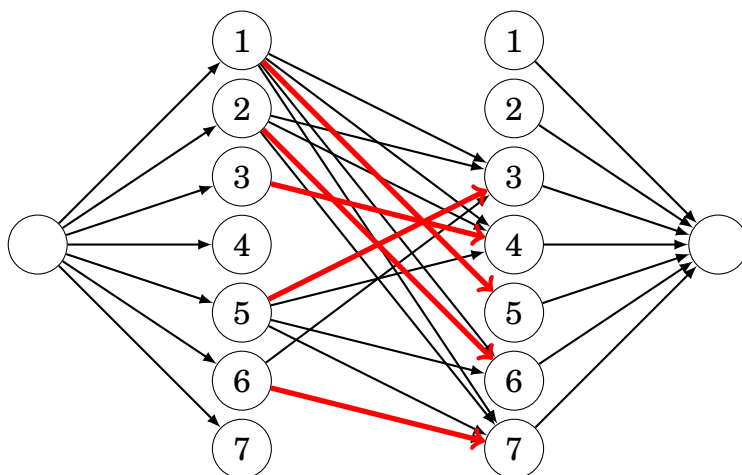


Và đường đi thứ hai có thể như sau:



Một phủ đường đi tổng quát tối thiểu có thể được tìm thấy gần giống như một phủ đường đi không giao nhau trên đỉnh tối thiểu. Chỉ cần thêm một số cạnh mới vào đồ thị ghép cặp sao cho luôn có một cạnh $a \rightarrow b$ bất cứ khi nào có một đường đi từ a đến b trong đồ thị ban đầu (có thể qua nhiều cạnh).

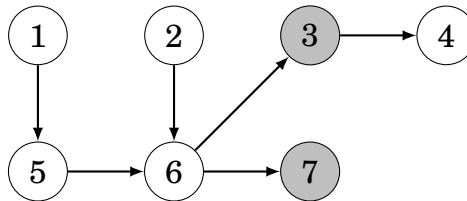
Đồ thị ghép cặp cho đồ thị trên như sau:



Dilworth's theorem

Một **đối chuỗi (antichain)** là một tập hợp các nút của một đồ thị sao cho không có đường đi từ bất kỳ nút nào đến một nút khác sử dụng các cạnh của đồ thị. **Định lý Dilworth (Dilworth's theorem)** phát biểu rằng trong một đồ thị có hướng không chu trình, kích thước của một phủ đường đi tổng quát tối thiểu bằng kích thước của một đối chuỗi cực đại.

Ví dụ, các nút 3 và 7 tạo thành một đối chuỗi trong đồ thị sau:



Đây là một đối chuỗi cực đại, vì không thể xây dựng bất kỳ đối chuỗi nào chứa ba nút. Chúng ta đã thấy trước đó rằng kích thước của một phủ đường đi tổng quát tối thiểu của đồ thị này bao gồm hai đường đi.

Phần III

Advanced topics

Chương 21

Lý thuyết số

Lý thuyết số (Number theory) là một nhánh của toán học nghiên cứu về các số nguyên. Lý thuyết số là một lĩnh vực hấp dẫn, bởi vì nhiều câu hỏi liên quan đến số nguyên rất khó giải quyết mặc dù chúng trông có vẻ đơn giản thoạt nhìn.

Ví dụ, hãy xem xét phương trình sau:

$$x^3 + y^3 + z^3 = 33$$

Dễ dàng tìm thấy ba số thực x, y và z thỏa mãn phương trình. Ví dụ, chúng ta có thể chọn

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

Tuy nhiên, đó là một bài toán mở trong lý thuyết số liệu có ba số *nguyên* x, y và z nào thỏa mãn phương trình hay không [6].

Trong chương này, chúng ta sẽ tập trung vào các khái niệm cơ bản và các thuật toán trong lý thuyết số. Trong suốt chương, chúng ta sẽ giả định rằng tất cả các số đều là số nguyên, trừ khi có quy định khác.

21.1 Số nguyên tố và ước số

Một số a được gọi là một **thừa số (factor)** hoặc một **ước số (divisor)** của một số b nếu a chia hết cho b . Nếu a là một thừa số của b , chúng ta viết $a \mid b$, và ngược lại chúng ta viết $a \nmid b$. Ví dụ, các ước số của 24 là 1, 2, 3, 4, 6, 8, 12 và 24.

Một số $n > 1$ là một **số nguyên tố (prime)** nếu các ước số dương duy nhất của nó là 1 và n . Ví dụ, 7, 19 và 41 là các số nguyên tố, nhưng 35 không phải là một số nguyên tố, vì $5 \cdot 7 = 35$. Với mọi số $n > 1$, có một **phân tích ra thừa số nguyên tố (prime factorization)** duy nhất

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

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 ra thừa số nguyên tố của 84 là

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Số lượng các ước số của một số n là

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

bởi vì với mỗi số nguyên tố p_i , có $\alpha_i + 1$ cách để chọn số lần nó xuất hiện trong ước số. Ví dụ, số lượng các ước số của 84 là $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. Các ước số là 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 và 84.

Tổng các ước số 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 số 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 số của n là

$$\mu(n) = n^{\tau(n)/2},$$

bởi vì chúng ta có thể tạo thành $\tau(n)/2$ cặp từ các ước số, mỗi cặp có tích là n . Ví dụ, các ước số của 84 tạo ra các cặp $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$, v.v., và tích của các ước số là $\mu(84) = 84^6 = 351298031616$.

Một số n được gọi là một **số hoàn hảo (perfect number)** nếu $n = \sigma(n) - n$, tức là, n bằng tổng các ước số của nó trong khoảng từ 1 đến $n - 1$. Ví dụ, 28 là một số hoàn hảo, bởi vì $28 = 1 + 2 + 4 + 7 + 14$.

Số lượng các số nguyên tố

Dễ dàng chứng minh rằng có vô số số nguyên tố. Nếu số lượng các số nguyên tố là hữu hạn, chúng ta có thể xây dựng một tập hợp $P = \{p_1, p_2, \dots, p_n\}$ chứa tất cả các số nguyên tố. Ví dụ, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, và cứ thế. Tuy nhiên, sử dụng P , chúng ta có thể tạo ra một số nguyên tố mới

$$p_1 p_2 \cdots p_n + 1$$

lớn hơn tất cả các phần tử trong P . Đây là một mâu thuẫn, và số lượng các số nguyên tố phải là vô hạn.

Mật độ của các số nguyên tố

Mật độ của các số nguyên tố có nghĩa là các số nguyên tố xuất hiện thường xuyên như thế nào trong các số. Gọi $\pi(n)$ là số lượng các số nguyên tố trong khoảng từ 1 đến n . Ví dụ, $\pi(10) = 4$, bởi vì có 4 số nguyên tố trong khoảng từ 1 đến 10: 2, 3, 5 và 7.

Có thể chứng minh rằng

$$\pi(n) \approx \frac{n}{\ln n},$$

điều này có nghĩa là các số nguyên tố khá thường xuyên. Ví dụ, số lượng các số nguyên tố trong khoảng từ 1 đến 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 các số nguyên tố. Hầu hết mọi người nghĩ rằng các giả thuyết là đúng, nhưng không ai có thể chứng minh chúng. Ví dụ, các giả thuyết sau đây là nổi tiếng:

- **Giả thuyết Goldbach (Goldbach's conjecture):** 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 các số nguyên tố.
- **Giả thuyết số nguyên tố sinh đôi (Twin prime conjecture):** Có vô số cặp có dạng $\{p, p + 2\}$, trong đó cả p và $p + 2$ đều là các số nguyên tố.
- **Giả thuyết Legendre (Legendre's conjecture):** 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 là số nguyên tố, nó có thể được biểu diễn dưới dạng một tích $a \cdot b$, trong đó $a \leq \sqrt{n}$ hoặc $b \leq \sqrt{n}$, vì vậy nó chắc chắn có một ước số trong khoảng từ 2 đến $\lfloor \sqrt{n} \rfloor$. Sử dụng quan sát này, chúng ta có thể vừa kiểm tra xem một số có phải là số nguyên tố hay không vừa tìm phân tích ra thừa số nguyên tố của một số trong thời gian $O(\sqrt{n})$.

Hàm prime sau đây kiểm tra xem số đã cho n có phải là số nguyên tố hay không. Hàm cố gắng chia n cho tất cả các số trong khoảng từ 2 đến $\lfloor \sqrt{n} \rfloor$, và nếu không có số nào trong số đó chia hết cho 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 đây xây dựng một vector chứa phân tích ra 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 số nào trong khoảng từ 2 đến $\lfloor \sqrt{n} \rfloor$. 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 nhiều lần bằng số lần nó chia hết cho số đó. Ví dụ, $24 = 2^3 \cdot 3$, vì vậy kết quả của hàm là $[2, 2, 2, 3]$.

Sàng Eratosthenes

Sàng Eratosthenes (sieve of Eratosthenes) là một thuật toán tiền xử lý xây dựng một mảng mà chúng ta có thể sử dụng để kiểm tra hiệu quả xem một số đã cho trong

khoảng từ $2 \dots n$ có phải là số nguyên tố hay không và, nếu không, tìm một thừa số nguyên tố của số đó.

Thuật toán xây dựng một mảng sieve mà các vị trí $2, 3, \dots, n$ được sử dụng. Giá trị $\text{sieve}[k] = 0$ có nghĩa là k là số nguyên tố, và giá trị $\text{sieve}[k] \neq 0$ có nghĩa là k không phải là số nguyên tố và một trong các thừa số nguyên tố của nó là $\text{sieve}[k]$.

Thuật toán lặp qua các số $2 \dots n$ một cách lần lượt. Mỗi khi một số nguyên tố mới x được tìm thấy, thuật toán ghi lại rằng các bội số của x ($2x, 3x, 4x, \dots$) không phải là số nguyên tố, bởi vì số x chia hết cho 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 đây triển khai sàng Eratosthenes. Đoạn mã giả định 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 bên trong của thuật toán được thực hiện n/x lần cho mỗi giá trị của x . Do đó, một giới hạn 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 tế, thuật toán hiệu quả hơn, bởi vì vòng lặp bên trong sẽ 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 (greatest common divisor) của các số a và b , $\text{gcd}(a, b)$, là số lớn nhất chia hết cho cả a và b , và **bội chung nhỏ nhất (least common multiple)** 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 mối liên hệ như sau:

$$\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

Thuật toán Euclid (Euclid's algorithm)¹ cung cấp một cách hiệu quả để tìm ước chung lớn nhất của hai số. Thuật toán dựa trên công thức sau:

$$\text{gcd}(a, b) = \begin{cases} a & b = 0 \\ \text{gcd}(b, a \bmod b) & b \neq 0 \end{cases}$$

¹Euclid là một nhà toán học người Hy Lạp sống vào khoảng năm 300 TCN. Đây có lẽ là thuật toán đầu tiên được biết đến trong lịch sử.

Ví dụ,

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Thuật toán có thể được triển khai 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 hoạt động trong thời gian $O(\log n)$, trong đó $n = \max(a, b)$. Trường hợp xấu nhất cho thuật toán là trường hợp khi a và b là các số Fibonacci liên tiếp. Ví dụ,

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

Hàm phi Euler

Các số a và b là **nguyên tố cùng nhau (coprime)** nếu $\gcd(a, b) = 1$. **Hàm phi Euler (Euler's totient function)** $\varphi(n)$ cho ra số lượng các số nguyên tố cùng nhau với n trong khoảng từ 1 đến n . Ví dụ, $\varphi(12) = 4$, bởi vì 1, 5, 7 và 11 là nguyên tố cùng nhau với 12.

Giá trị của $\varphi(n)$ có thể được tính từ phân tích ra 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 mô-đun

Trong **số học mô-đun (modular arithmetic)**, tập hợp các số bị giới hạn sao cho chỉ có các số $0, 1, 2, \dots, m-1$ được sử dụng, trong đó m là một hằng số. Mỗi số x được biểu diễn bằng số $x \bmod m$: phần dư sau khi chia x cho m . Ví dụ, nếu $m = 17$, thì 75 được biểu diễn bằng $75 \bmod 17 = 7$.

Thường thì chúng ta có thể lấy phần dư trước khi thực hiện các phép tính. Cụ thể, các công thức sau đây 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 mô-đun

Thường có nhu cầu tính toán hiệu quả giá trị của $x^n \bmod m$. Điều này có thể được thực hiện trong thời gian $O(\log n)$ sử dụng công thức đệ quy sau:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ là số chẵn} \\ x^{n-1} \cdot x & n \text{ là số lẻ} \end{cases}$$

Điều quan trọng là trong trường hợp n chẵn, giá trị của $x^{n/2}$ chỉ được tính một lần. Điều này đảm bảo rằng độ phức tạp thời gian của thuật toán là $O(\log n)$, bởi vì n luôn được chia đôi khi nó chẵn.

Hàm sau tính giá trị của $x^n \bmod m$:

```
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 (Fermat's theorem) phát biểu rằng

$$x^{m-1} \bmod m = 1$$

khi m là số nguyên tố và x và m là nguyên tố cùng nhau. Điều này cũng cho ra

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

Tổng quát hơn, **Định lý Euler (Euler's theorem)** phát biểu rằng

$$x^{\varphi(m)} \bmod m = 1$$

khi x và m là nguyên tố cùng nhau. Định lý Fermat là một hệ quả của định lý Euler, bởi vì nếu m là một số nguyên tố, thì $\varphi(m) = m - 1$.

Nghịch đảo mô-đun

Nghịch đảo của x theo mô-đun m là một số x^{-1} sao cho

$$xx^{-1} \bmod m = 1.$$

Ví dụ, nếu $x = 6$ và $m = 17$, thì $x^{-1} = 3$, bởi vì $6 \cdot 3 \bmod 17 = 1$.

Sử dụng nghịch đảo mô-đun, chúng ta có thể chia các số theo mô-đun m , bởi vì phép chia cho x tương ứng với phép nhân với x^{-1} . Ví dụ, để đánh giá giá trị của $36/6 \bmod 17$, chúng ta có thể sử dụng công thức $2 \cdot 3 \bmod 17$, bởi vì $36 \bmod 17 = 2$ và $6^{-1} \bmod 17 = 3$.

Tuy nhiên, một nghịch đảo mô-đun 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 được, bởi vì tất cả các bội số của 2 đều chẵn và phần dư không bao giờ có thể là 1 khi $m = 4$. Hóa ra giá trị của $x^{-1} \bmod m$ có thể được tính chính xác khi x và m là nguyên tố cùng nhau.

Nếu một nghịch đảo mô-đun 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 chúng ta tính toán hiệu quả các nghịch đảo mô-đun bằng thuật toán lũy thừa mô-đun. Công thức có thể được suy ra bằng cách sử dụng định lý Euler. Đầu tiên, nghịch đảo mô-đun phải thỏa mãn phương trình sau:

$$xx^{-1} \bmod m = 1.$$

Mặt khác, theo định lý Euler,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

vì vậy các số x^{-1} và $x^{\varphi(m)-1}$ bằng nhau.

Số học máy tính

Trong lập trình, các số nguyên không dấu được biểu diễn theo mô-đun 2^k , trong đó k là số bit của kiểu dữ liệu. Một hệ quả thông thường của điều này là một số sẽ quay vòng nếu nó trở nên quá lớn.

Ví dụ, trong C++, các số thuộc kiểu unsigned int được biểu diễn theo mô-đun 2^{32} . Đoạn mã sau khai báo một biến unsigned int có giá trị là 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 (Diophantine equation)** là một phương trình có dạng

$$ax + by = c,$$

trong đó a , b và c là các hằng số và các giá trị của x và y cần được tìm. Mỗi số trong phương trình phải là một số nguyên. Ví dụ, một nghiệm của phương trình $5x + 2y = 11$ là $x = 3$ và $y = -2$.

Chúng ta có thể giải quyết hiệu quả một phương trình Diophantine bằng cách sử dụng thuật toán Euclid. Hóa ra chúng ta có thể mở rộng thuật toán Euclid để nó sẽ tìm ra các số x và y thỏa mãn phương trình sau:

$$ax + by = \gcd(a, b)$$

Một phương trình Diophantine có thể được giải nếu c chia hết cho $\gcd(a, b)$, và ngược lại nó không thể được giải.

Ví dụ, chúng ta hãy tìm các số x và y thỏa mãn phương trình sau:

$$39x + 15y = 12$$

Phương trình có thể được giải, bởi 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 các lần 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:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Sử dụng các phương trình này, chúng ta có thể suy ra

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

và bằng cách nhân điều này với 4, kết quả là

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

vì vậy một nghiệm của phương trình là $x = 8$ và $y = -20$.

Một nghiệm của một phương trình Diophantine không phải là duy nhất, bởi vì chúng ta có thể tạo ra vô số nghiệm nếu chúng ta biết một nghiệm. Nếu một cặp (x, y) là một nghiệm, thì tất cả các cặp

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

cũng là nghiệm, trong đó k là một số nguyên bất kỳ.

Định lý số dư Trung Hoa

Định lý số dư Trung Hoa (Chinese remainder theorem) giải quyết một nhóm các phương trình có 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 tất cả các cặp m_1, m_2, \dots, m_n là nguyên tố cùng nhau.

Gọi x_m^{-1} là nghịch đảo của x theo mô-đun m , và

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Sử dụng 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 $k = 1, 2, \dots, n$,

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

bởi vì

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Vì tất cả các số hạng 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 cho

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

là

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Một khi chúng ta đã tìm thấy một nghiệm x , chúng ta có thể tạo ra vô số nghiệm khác, bởi vì tất cả các 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 (Lagrange's theorem) phát biểu rằng mọi số nguyên dương có thể được biểu diễn dưới dạng tổng của bốn số chính 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 (Zeckendorf's theorem) phát biểu rằng mọi số nguyên dương có một biểu diễn duy nhất dưới dạng một tổng các số Fibonacci sao cho không có hai số nào bằng nhau hoặc là các 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 (Pythagorean triple)** là một bộ ba (a, b, c) thỏa mãn định lý Pythagoras $a^2 + b^2 = c^2$, có nghĩa là có một tam giác vuông với các cạnh có độ dài a , b và c . Ví dụ, $(3, 4, 5)$ là một bộ ba Pythagoras.

Nếu (a, b, c) là một bộ ba Pythagoras, tất cả các bộ ba có dạng (ka, kb, kc) cũng là các bộ ba Pythagoras trong đó $k > 1$. Một bộ ba Pythagoras là *nguyên thủy* nếu a , b và c là nguyên tố cùng nhau, và tất cả các bộ ba Pythagoras có thể được xây dựng từ các bộ ba nguyên thủy bằng cách sử dụng một hệ số nhân k .

Công thức Euclid (Euclid's formula) có thể được sử dụng để tạo ra 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 là 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 tạo 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 (Wilson's theorem) phát biểu rằng một số n là số nguyên tố chính xác khi

$$(n - 1)! \bmod n = n - 1.$$

Ví dụ, số 11 là số nguyên tố, bởi vì

$$10! \bmod 11 = 10,$$

và số 12 không phải là số nguyên tố, bởi vì

$$11! \bmod 12 = 0 \neq 11.$$

Do đó, định lý Wilson có thể được sử dụng để tìm ra xem một số có phải là số nguyên tố hay không. Tuy nhiên, trong thực tế, định lý không thể được áp dụng cho các giá trị lớn của n , bởi vì rất khó để tính các giá trị của $(n-1)!$ khi n lớn.

Chương 22

Tổ hợp

Tổ hợp (Combinatorics) nghiên cứu các phương pháp đếm các kết hợp của các đối tượng. Thông thường, mục tiêu là tìm ra một cách để đếm các kết hợp một cách hiệu quả mà không cần tạo ra từng kết hợp riêng lẻ.

Ví dụ, hãy xem xét bài toán đếm số cách để biểu diễn một số nguyên n dưới dạng tổng của các số nguyên dương. Ví dụ, có 8 cách biểu diễn cho số 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 quyết bằng một hàm đệ quy. Trong bài toán này, chúng ta có thể định nghĩa một hàm $f(n)$ cho ra số cách biểu diễn của n . Ví dụ, $f(4) = 8$ theo ví dụ trên. Các giá trị của hàm có thể được tính đệ quy như sau:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \dots + f(n-1) & n > 0 \end{cases}$$

Trường hợp cơ sở là $f(0) = 1$, bởi vì tổng rỗng biểu diễn số 0. Sau đó, nếu $n > 0$, chúng ta xem xét tất cả các cách để chọn số đầu tiên của tổng. Nếu số đầu tiên là k , có $f(n-k)$ cách biểu diễn cho phần còn lại của tổng. Do đó, chúng ta tính tổng của tất cả các giá trị có dạng $f(n-k)$ trong đó $k < n$.

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 dạng đóng. Trong bài toán này,

$$f(n) = 2^{n-1},$$

dựa trên thực tế là có $n-1$ vị trí có thể cho các dấu $+$ trong tổng và chúng ta có thể chọn bất kỳ tập hợp con nào trong số chúng.

22.1 Tổ hợp chập (Binomial coefficients)

Tổ hợp chập (Binomial coefficient) $\binom{n}{k}$ bằng số cách chúng ta có thể chọn một tập hợp con gồm k phần tử từ một tập hợp gồm n phần tử. Ví dụ, $\binom{5}{3} = 10$, bởi vì tập hợp $\{1, 2, 3, 4, 5\}$ có 10 tập hợp 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

Tổ hợp chập 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 hợp. Nếu x được bao gồm trong tập hợp con, chúng ta phải chọn $k-1$ phần tử từ $n-1$ phần tử, và nếu x không được bao gồm trong tập hợp con, chúng ta phải chọn k phần tử từ $n-1$ phần tử.

Các trường hợp cơ sở cho đệ quy là

$$\binom{n}{0} = \binom{n}{n} = 1,$$

bởi vì luôn có chính xác một cách để xây dựng một tập hợp con rỗng và một tập hợp con chứa tất cả các phần tử.

Công thức 2

Một cách khác để tính tổ hợp chập như sau:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Có $n!$ hoán vị của n phần tử. Chúng ta duyệt qua tất cả các hoán vị và luôn bao gồm k phần tử đầu tiên của hoán vị trong tập hợp con. Vì thứ tự của các phần tử trong tập hợp con và bên ngoài tập hợp con không quan trọng, kết quả được chia cho $k!$ và $(n-k)!$

Các thuộc tính

Đối với tổ hợp chập,

$$\binom{n}{k} = \binom{n}{n-k},$$

bởi vì thực tế chúng ta chia một tập hợp gồm n phần tử thành hai tập hợp con: tập hợp thứ nhất chứa k phần tử và tập hợp thứ hai chứa $n-k$ phần tử.

Tổng của các tổ hợp chập là

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Lý do cho tên "tổ hợp chập" có thể được nhìn thấy khi nhị thức $(a + b)$ được nâng lên lũy thừa thứ n :

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

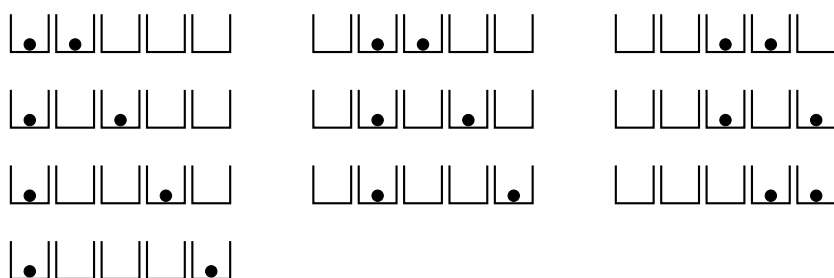
Tổ hợp chập cũng xuất hiện trong **tam giác Pascal (Pascal's triangle)** trong đó mỗi giá trị bằng tổng của hai giá trị phía trên:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & & \dots & & \dots & & \dots & & \dots & \end{array}$$

Hộp và bóng

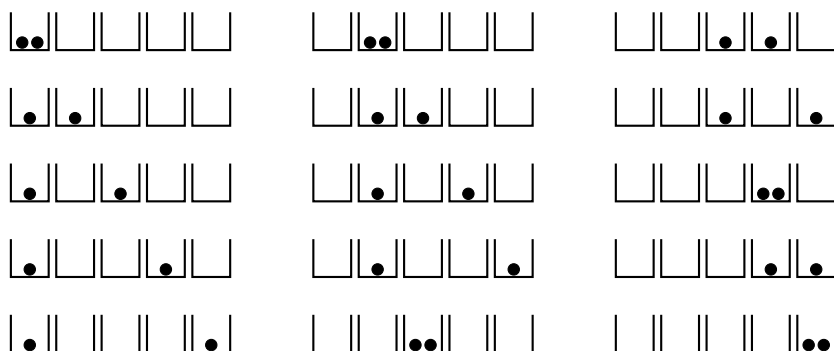
"Hộp và bóng" là một mô hình hữu ích, trong đó chúng ta đếm số cách để đặt k quả bóng vào n hộp. Hãy xem xét ba kịch bản:

Kịch bản 1: Mỗi hộp có thể chứa nhiều nhất một quả bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 10 giải pháp:



Trong kịch bản này, câu trả lời trực tiếp là tổ hợp chập $\binom{n}{k}$.

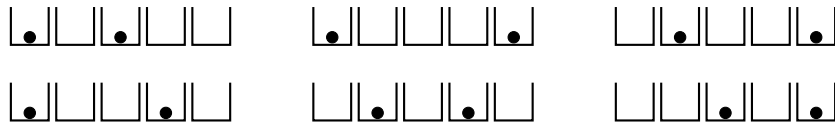
Kịch bản 2: Một hộp có thể chứa nhiều quả bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 15 giải pháp:



Quá trình đặt các quả bóng vào các hộp có thể được biểu diễn dưới dạng một chuỗi bao gồm các ký hiệu "o" và "→". Ban đầu, giả sử rằng chúng ta đang đứng ở hộp ngoài cùng bên trái. Ký hiệu "o" có nghĩa là chúng ta đặt một quả bóng vào hộp hiện tại, và ký hiệu "→" có nghĩa là chúng ta di chuyển đến hộp tiếp theo bên phải.

Sử dụng ký hiệu này, mỗi giải pháp là một chuỗi chứa k lần ký hiệu "o" và $n - 1$ lần ký hiệu "→". Ví dụ, giải pháp trên cùng bên phải trong hình trên tương ứng với chuỗi "→ → o → o →". Do đó, số lượng các giải pháp là $\binom{k+n-1}{k}$.

Kịch bản 3: Mỗi hộp có thể chứa nhiều nhất một quả bóng, và ngoài ra, không có hai hộp liên tiếp nào có thể cùng chứa một quả bóng. Ví dụ, khi $n = 5$ và $k = 2$, có 6 giải pháp:



Trong kịch bản này, chúng ta có thể giả sử rằng k quả bóng ban đầu được đặt trong các hộp và có một hộp trống giữa mỗi hai hộp liên tiếp. Nhiệm vụ còn lại là chọn các vị trí cho các hộp trống còn lại. Có $n - 2k + 1$ hộp như vậy và $k + 1$ vị trí cho chúng. Do đó, sử dụng công thức của kịch bản 2, số lượng các giải pháp là $\binom{n-k+1}{n-2k+1}$.

Hệ số đa thức

Hệ số đa thức (multinomial coefficient)

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

bằng số cách chúng ta có thể chia n phần tử thành các tập hợp con có kích thước k_1, k_2, \dots, k_m , trong đó $k_1 + k_2 + \dots + k_m = n$. Hệ số đa thức có thể được coi là một sự tổng quát hóa của tổ hợp chập; nếu $m = 2$, công thức trên tương ứng với công thức tổ hợp chập.

22.2 Số Catalan

Số Catalan (Catalan number) C_n bằng số lượng các biểu thức ngoặc hợp lệ bao gồm n dấu ngoặc trái và n dấu ngoặc phải.

Ví dụ, $C_3 = 5$, bởi vì chúng ta có thể xây dựng các biểu thức ngoặc sau đây bằng cách sử dụng ba dấu ngoặc trái và phải:

- $()()$
- $((()))$
- $()(())$
- $((()))$
- $((()))$

Biểu thức ngoặc

Chính xác thì *biểu thức ngoặc hợp lệ* là gì? Các quy tắc sau đây định nghĩa chính xác tất cả các biểu thức ngoặc hợp lệ:

- Một biểu thức ngoặc rỗng là hợp lệ.
- Nếu một biểu thức A là hợp lệ, thì biểu thức (A) cũng là hợp lệ.
- Nếu các biểu thức A và B là hợp lệ, thì biểu thức AB cũng là hợp lệ.

Một cách khác để mô tả các biểu thức ngoặc hợp lệ là nếu chúng ta chọn bất kỳ tiền tố nào của một biểu thức như vậy, nó phải chứa ít nhất số dấu ngoặc trái bằng số dấu ngoặc phải. Ngoài ra, biểu thức hoàn chỉnh phải chứa một số lượng bằng nhau các dấu ngoặc trái và phải.

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 duyệt qua các cách để chia biểu thức thành hai phần sao cho cả hai phần đều là các biểu thức hợp lệ và phần đầu tiên càng ngắn càng tốt nhưng không rỗng. Với bất kỳ i nào, phần đầu tiên chứa $i + 1$ cặp dấu ngoặc và số lượng các biểu thức là tích của các giá trị sau:

- C_i : số cách để xây dựng một biểu thức sử dụng các dấu ngoặc của phần đầu tiên, không tính các dấu ngoặc ngoài cùng
- C_{n-i-1} : số cách để xây dựng một biểu thức sử dụng các dấu ngoặc của phần thứ hai

Trường hợp cơ sở là $C_0 = 1$, bởi vì chúng ta có thể xây dựng một biểu thức ngoặc rỗng bằng cách sử dụng không cặp dấu ngoặc.

Công thức 2

Các số Catalan cũng có thể được tính bằng tổ hợp chập:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Công thức có thể được giải thích như sau:

Có tổng cộng $\binom{2n}{n}$ cách để xây dựng một biểu thức ngoặc (không nhất thiết phải hợp lệ) chứa n dấu ngoặc trái và n dấu ngoặc phải. Chúng ta 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 trong đó số lượng dấu ngoặc phải vượt quá số lượng dấu ngoặc trái. Ý tưởng là đảo ngược mỗi dấu ngoặc thuộc về một tiền tố như vậy. Ví dụ, biểu thức $((()())$ chứa một tiền tố $((()$, và sau khi đảo ngược tiền tố, biểu thức trở thành $)(((())$.

Biểu thức kết quả bao 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 các biểu thức ngoặc không hợp lệ. Do đó, số lượng các biểu thức ngoặc hợp lệ có thể được tính bằng công thức

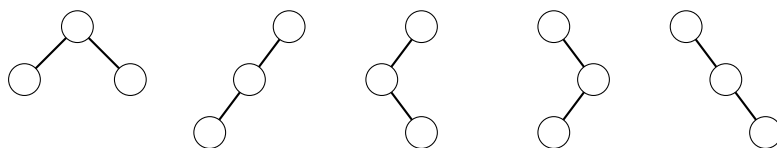
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Đếm cây

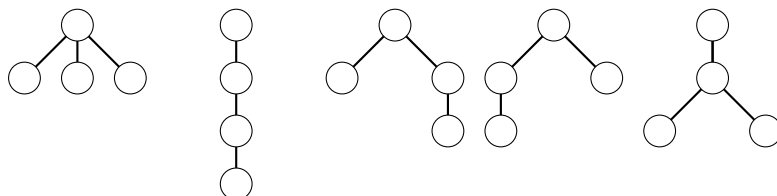
Các số Catalan cũng liên quan đến cây:

- có C_n cây nhị phân gồm n nút
- có C_{n-1} cây có gốc gồm n nút

Ví dụ, với $C_3 = 5$, các cây nhị phân là



và các cây có gốc là

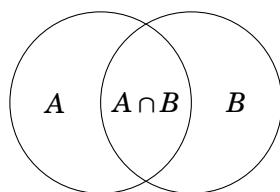


22.3 Bao hàm - loại trừ

Bao hàm - loại trừ (Inclusion-exclusion) là một kỹ thuật có thể được sử dụng để đếm kích thước của một hợp các tập hợp khi kích thước của các giao được biết, 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 hợp và $|X|$ biểu thị kích thước của X . Công thức có thể được minh họa như sau:

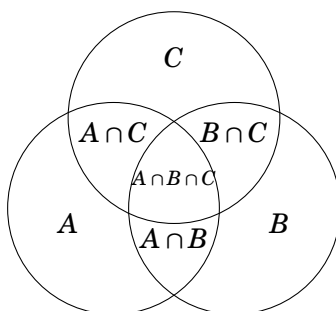


Mục tiêu của chúng ta là tính kích thước của hợp $A \cup B$ tương ứng với diện tích của vùng thuộc về ít nhất một vòng tròn. Hình ảnh cho thấy chúng ta có thể tính diện tích của $A \cup B$ bằng cách đầu tiên cộng diện tích của A và B và sau đó trừ đi diện tích của $A \cap B$.

Ý tưởng tương tự có thể được áp dụng khi số lượng các tập hợp lớn hơn. Khi có ba tập hợp, công thức bao hàm-loại 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 ảnh 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 duyệt qua tất cả các giao có thể có chứa một số các tập hợp X_1, X_2, \dots, X_n . Nếu giao chứa một số lẻ các tập hợp, kích thước của nó được cộng vào câu trả lời, và ngược lại, kích thước của nó được trừ khỏi câu trả lời.

Lưu ý rằng 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 chỗ

Ví dụ, chúng ta hãy đếm số lượng các **hoán vị mất chỗ (derangements)** của các phần tử $\{1, 2, \dots, n\}$, tức là, các hoán vị trong đó không có phần tử nào ở vị trí ban đầu của nó. Ví dụ, khi $n = 3$, có hai hoán vị mất chỗ: $(2, 3, 1)$ và $(3, 1, 2)$.

Một cách tiếp cận để giải quyết bài toán là sử dụng bao hàm-loại trừ. Gọi X_k là tập hợp các hoán vị chứa phần tử k ở vị trí k . Ví dụ, khi $n = 3$, các tập hợp như sau:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Sử dụng các tập hợp này, số lượng các hoán vị mất chỗ bằng

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

vì vậy chỉ cần tính kích thước của hợp. Sử dụng bao hàm-loại trừ, điều này quy về việc tính kích thước của các giao có thể được thực hiện một cách hiệu quả. Ví dụ, khi $n = 3$, kích thước của $|X_1 \cup X_2 \cup X_3|$ là

$$\begin{aligned} &|X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

vì vậy số lượng các giải pháp là $3! - 4 = 2$.

Hóa ra bài toán cũng có thể được giải quyết mà không cần sử dụng bao hàm-loại trừ. Gọi $f(n)$ là số lượng các hoán vị mất chỗ cho $\{1, 2, \dots, n\}$. Chúng ta có thể sử dụng 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 xem xét các khả năng cách phần tử 1 thay đổi trong hoán vị mất chỗ. Có $n-1$ cách để chọn một phần tử x thay thế phần tử 1. Trong mỗi lựa chọn như vậy, có hai tùy chọn:

Tùy chọn 1: Chúng ta cũng thay thế phần tử x bằng phần tử 1. Sau đó, nhiệm vụ còn lại là xây dựng một hoán vị mất chỗ của $n-2$ phần tử.

Tùy chọn 2: Chúng ta thay thế phần tử x bằng một phần tử khác 1. Bây giờ chúng ta phải xây dựng một hoán vị mất chỗ của $n-1$ phần tử, bởi vì chúng ta không thể thay thế phần tử x bằng phần tử 1, và tất cả các phần tử khác phải được thay đổi.

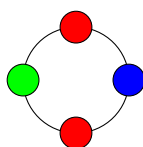
22.4 Bổ đề Burnside

Bổ đề Burnside (Burnside's lemma) có thể được sử dụng để đếm số lượng các kết hợp sao cho chỉ có một đại diện được đếm cho mỗi nhóm các kết hợp đối xứng. Bổ đề Burnside phát biểu rằng số lượng các kết hợp là

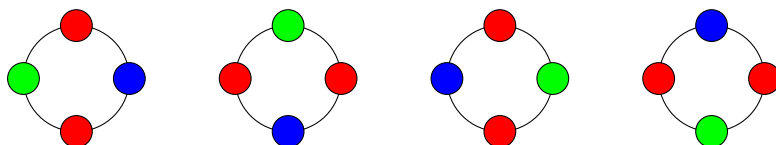
$$\sum_{k=1}^n \frac{c(k)}{n},$$

trong đó có n cách để thay đổi vị trí của một kết hợp, và có $c(k)$ kết hợp vẫn không thay đổi khi cách thứ k được áp dụng.

Ví dụ, chúng ta hãy tính số lượng các 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 chúng. Ví dụ, vòng cổ



có các vòng cổ đối xứng sau:



Có n cách để thay đổi vị trí của một vòng cổ, bởi vì chúng 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 vẫn giữ nguyên, và nếu số bước là 1, chỉ có m vòng cổ trong đó mỗi hạt có cùng một màu vẫn giữ nguyên.

Tổng quát hơn, khi số bước là k , tổng cộng

$$m^{\gcd(k,n)}$$

vòng cổ vẫn giữ nguyên, trong đó $\gcd(k,n)$ là ước chung lớn nhất của k và n . Lý do cho điều này là các khối hạt có kích thước $\gcd(k,n)$ sẽ thay thế cho nhau. Do đó, theo bổ đề Burnside, số lượng các vòng cổ là

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

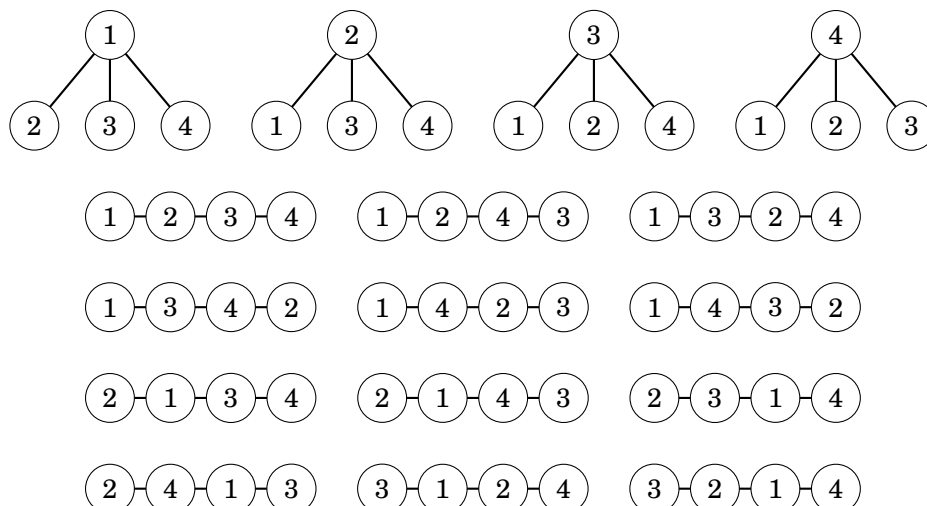
Ví dụ, số lượng các vòng cổ có độ dài 4 với 3 màu là

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Công thức Cayley

Công thức Cayley (Cayley's formula) phát biểu rằng có n^{n-2} cây có nhãn chứa n nút. Các nút được dán nhãn $1, 2, \dots, n$, và hai cây là khác nhau nếu cấu trúc của chúng hoặc nhãn của chúng khác nhau.

Ví dụ, khi $n = 4$, số lượng các cây có nhãn là $4^{4-2} = 16$:

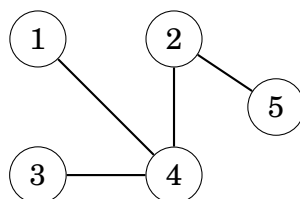


Tiếp theo chúng ta sẽ xem làm thế nào công thức Cayley có thể được suy ra bằng cách sử dụng mã Prüfer.

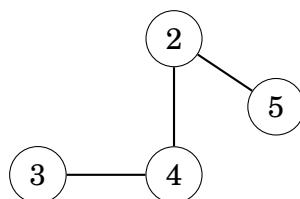
Mã Prüfer

Một **mã Prüfer (Prüfer code)** là một dãy $n - 2$ số mô tả một cây có nhãn. Mã được xây dựng bằng cách tuân theo một quy trình loại bỏ $n - 2$ lá khỏi cây. Ở mỗi bước, lá có nhãn nhỏ nhất bị loại bỏ, và nhãn của hàng xóm duy nhất của nó được thêm vào mã.

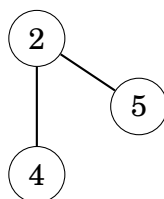
Ví dụ, chúng ta hãy tính mã Prüfer của đồ thị sau:



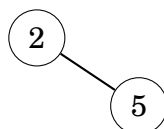
Đầu tiên chúng ta loại bỏ nút 1 và thêm nút 4 vào mã:



Sau đó chúng ta loại bỏ nút 3 và thêm nút 4 vào mã:



Cuối cùng chúng ta loại bỏ nút 4 và thêm nút 2 vào mã:



Do đó, mã Prüfer của đồ thị là $[4, 4, 2]$.

Chúng ta có thể xây dựng một mã Prüfer cho bất kỳ cây nào, và quan trọng hơn, cây ban đầu có thể được tái tạo từ một mã Prüfer. Do đó, số lượng các cây có nhãn gồm n nút bằng n^{n-2} , số lượng các mã Prüfer có kích thước n .

Chương 23

Ma trận

Ma trận (matrix) là một khái niệm toán học tương ứng với 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 đến phần tử ở hàng i và cột j trong 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 **vectơ** (vector) là ma trận một chiều kích thước $n \times 1$. Ví dụ,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

là một vectơ chứa ba phần tử.

Chuyển vị (transpose) A^T của một ma trận A được tạo ra khi các hàng và cột của A được hoán đổi, 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** (square matrix) 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 Các phép toán

Tổng $A + B$ của các ma trận A và B được định nghĩa nếu các 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 có nghĩa là mỗi phần tử của A được nhân với x . Ví dụ,

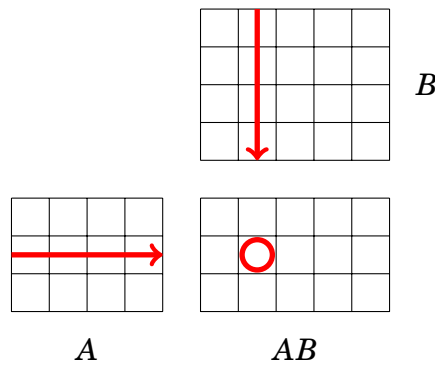
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Phép 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 $a \times b$ với 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 một tổng các tích của các phần tử của A và B theo hình minh họa 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, tức là $A(BC) = (AB)C$ thỏa mãn, nhưng không có tính giao hoán, tức là thường thì $AB \neq BA$.

Ma trận đơn vị (identity matrix) là ma trận vuông mà mỗi phần tử trên đường chéo là 1 và tất cả các phần tử khác là 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 một ma trận đơn vị không làm thay đổi nó. Ví dụ,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{và} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Sử dụng một thuật toán đơn giản, ta có thể tính tích của hai ma trận $n \times n$ trong thời gian $O(n^3)$. Cũng có những thuật toán hiệu quả hơn cho phép nhân ma trận¹, nhưng chúng chủ yếu mang tính lý thuyết và những thuật toán như vậy không cần thiết trong lập trình thi đấu.

¹Thuật toán đầu tiên như vậy là thuật toán Strassen, được công bố năm 1969 [63], có độ phức tạp thời gian là $O(n^{2.80735})$; thuật toán tốt nhất hiện tại [27] chạy trong thời gian $O(n^{2.37286})$.

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 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 $O(n^3 \log k)$ sử dụng thuật toán trong 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 (determinant) $\det(A)$ 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 theo công thức

$$\det(A) = \sum_{j=1}^n A[1, j]C[1, j],$$

trong đó $C[i, j]$ là **phần bù đại số** (cofactor) của A tại $[i, j]$. Phần bù đại số được tính theo công thức

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

trong đó $M[i, j]$ được tạo ra bằng cách loại bỏ hàng i và cột j khỏi A . Do hệ số $(-1)^{i+j}$ trong phần bù đại số, nên các định thức sẽ lần lượt dương và âm. Ví dụ,

$$\det\left(\begin{bmatrix} 3 & 4 \\ 1 & 6 \end{bmatrix}\right) = 3 \cdot 6 - 4 \cdot 1 = 14$$

và

$$\det\left(\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}\right) = 2 \cdot \det\left(\begin{bmatrix} 1 & 6 \\ 2 & 4 \end{bmatrix}\right) - 4 \cdot \det\left(\begin{bmatrix} 5 & 6 \\ 7 & 4 \end{bmatrix}\right) + 3 \cdot \det\left(\begin{bmatrix} 5 & 1 \\ 7 & 2 \end{bmatrix}\right) = 81.$$

Định thức của A cho ta biết liệu có tồn tại **ma trận nghịch đảo** (inverse matrix) 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 theo công thức

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Ví dụ,

$$\underbrace{\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{bmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_I.$$

23.2 Hệ thức truy hồi tuyến tính

Hệ thức truy hồi tuyến tính (linear recurrence) là một hàm $f(n)$ mà các giá trị ban đầu là $f(0), f(1), \dots, f(k-1)$ và các giá trị lớn hơn được tính đệ quy theo 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 số.

Quy hoạch động có thể được sử dụng để tính bất kỳ giá trị nào của $f(n)$ trong thời gian $O(kn)$ bằng cách tính tất cả các giá trị $f(0), f(1), \dots, f(n)$ lần lượt. Tuy nhiên, nếu k nhỏ, ta có thể tính $f(n)$ hiệu quả hơn nhiều trong thời gian $O(k^3 \log n)$ bằng cách sử dụng các phép toán ma trận.

Dãy Fibonacci

Một ví dụ đơn giản về hệ thức truy hồi tuyến tính là hàm sau đây định nghĩa dãy 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 hiệu quả các số Fibonacci, ta biểu diễn công thức Fibonacci bằng một ma trận vuông X kích thước 2×2 , thỏa mãn điều kiện sau:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Do đó, các giá trị $f(i)$ và $f(i+1)$ được cho là "đầu vào" cho X , và X tính các giá trị $f(i+1)$ và $f(i+2)$ từ chúng. Hóa ra ma trận như vậy là

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Do đó, ta có thể tính $f(n)$ sử dụng 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 $O(\log n)$, do đó giá trị của $f(n)$ cũng có thể được tính trong thời gian $O(\log n)$.

Trường hợp tổng quát

Bây giờ ta xét trường hợp tổng quát khi $f(n)$ là một hệ thức truy hồi tuyến tính bất kỳ. Một lần nữa, mục tiêu của ta là xây dựng một ma trận X mà thỏa mãn

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

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, mỗi phần tử là 0 ngoại trừ một phần tử là 1. Các hàng này thay thế $f(i)$ bằng $f(i+1)$, $f(i+1)$ bằng $f(i+2)$, và tiếp tục như vậy. Hàng cuối cùng chứa các hệ số của hệ thức truy hồi để tính giá trị mới $f(i+k)$.

Bây giờ, $f(n)$ có thể được tính trong thời gian $O(k^3 \log n)$ sử dụng công thức

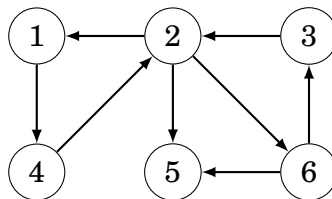
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

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 có n cạnh giữa các nút 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

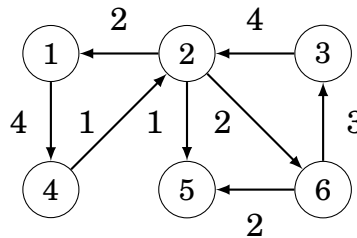
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

chứa số lượng đường đi có 4 cạnh giữa các nút. Ví dụ, $V^4[2,5] = 2$, bởi vì có hai đường đi 4 cạnh từ nút 2 đến nút 5: $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ và $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Đường đi ngắn nhất

Sử dụng một ý tưởng tương tự trong đồ thị có trọng số, ta có thể tính cho mỗi cặp nút độ dài nhỏ nhất của đường đi giữa chúng có chính xác n cạnh. Để tính được 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 của đường đi.

As an example, consider the following graph:



Ta hãy xây dựng một ma trận kề trong đó ∞ nghĩa là không tồn tại cạnh, 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

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j]$$

ta bây giờ sử dụng công thức

$$AB[i,j] = \min_{k=1}^n A[i,k] + B[k,j]$$

cho phép nhân ma trận, vì vậy ta tính giá trị nhỏ nhất thay vì tổng, và tổng các phần tử thay vì tích. Sau sự thay đổi này, lũy thừa ma trận tương ứng với đường đi ngắn nhất trong đồ thị.

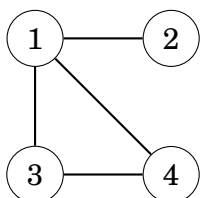
Ví dụ, khi

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

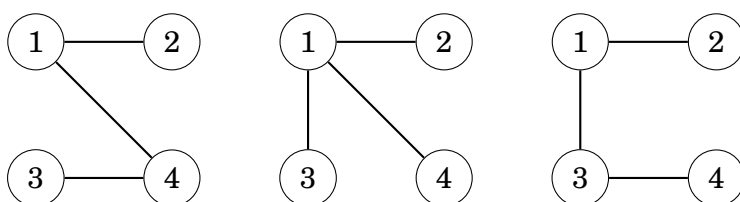
ta có thể kết luận rằng độ dài nhỏ nhất của đường đi có 4 cạnh từ nút 2 đến nút 5 là 8. Một đường đi như vậy là $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

Định lý Kirchhoff

Định lý Kirchhoff (Kirchhoff's theorem) cung cấp một cách để tính số lượng cây khung của một đồ thị dưới dạng định thức của một ma trận đặc biệt. Ví dụ, đồ thị



has three spanning trees:



Để tính số lượng cây khung, ta xây dựng một **ma trận Laplace** (Laplacian matrix) L , trong đó $L[i, i]$ là bậc của nút i và $L[i, j] = -1$ nếu có cạnh nối giữa các nút i và j , và ngược lại $L[i, j] = 0$. Ma trận Laplace cho 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 ma trận thu được khi ta loại bỏ bất kỳ hàng và cột nào từ L . Ví dụ, nếu ta loại bỏ hàng và cột đầu tiên, kết quả là

$$\det \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} = 3.$$

Định thức luôn giống nhau, bất kể ta loại bỏ hàng và cột nào từ L .

Chú ý rằng công thức Cayley trong Chương 22.5 là một trường hợp đặc biệt của định lý Kirchhoff, bởi vì trong đồ thị đầy đủ có n nút

$$\det \begin{bmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{bmatrix} = n^{n-2}.$$

Chương 24

Xác suất (Probability)

Một **xác suất (probability)** là một số thực trong khoảng từ 0 đến 1 chỉ ra khả năng xảy ra của một biến cố. Nếu một biến cố chắc chắn sẽ xảy ra, xác suất của nó là 1, và nếu một biến cố không thể xảy ra, xác suất của nó là 0. Xác suất của một biến cố được ký hiệu là $P(\dots)$ trong đó ba dấu chấm mô tả biến cố đó.

Ví dụ, khi gieo một con súc sắc, kết quả là một số nguyên từ 1 đến 6, và xác suất của mỗi kết quả là $1/6$. Ví dụ, chúng 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 biến cố, chúng ta có thể sử dụng tổ hợp hoặc mô phỏng quá trình tạo ra biến cố đó. Ví dụ, chúng ta hãy tính xác suất rút được ba lá bài có cùng giá trị từ một bộ bài đã được xáo trộn (ví dụ, $\spadesuit 8$, $\clubsuit 8$ và $\diamondsuit 8$).

Phương pháp 1

Chúng 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 trường hợp mà giá trị của mỗi lá bài là như nhau. Có $13 \binom{4}{3}$ kết quả như vậy, bởi vì có 13 khả năng cho giá trị của các lá bài và $\binom{4}{3}$ cách để chọn 3 chất từ 4 chất có thể.

Tổng cộng có $\binom{52}{3}$ kết quả, bởi vì chúng ta chọn 3 lá bài từ 52 lá. Do đó, xác suất của biến cố là

$$\frac{13 \binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Phương pháp 2

Một cách khác để tính xác suất là mô phỏng quá trình tạo ra biến cố. Trong ví dụ này, chúng ta rút ba lá bài, vì vậy quá trình bao gồm ba bước. Chúng ta yêu cầu mỗi bước của

quá trình đều phải thành công.

Việc rút lá bài đầu tiên chắc chắn thành công, vì không có ràng buộc nào. Bước thứ hai thành công với xác suất $3/51$, vì còn lại 51 lá bài và 3 trong số đó có cùng giá trị với lá bài 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 Biến cố (Events)

Một biến cố trong lý thuyết xác suất có thể được biểu diễn dưới dạng một tập hợp

$$A \subset X,$$

trong đó X chứa tất cả các kết quả có thể xảy ra và A là một tập hợp con các kết quả. Ví dụ, khi gieo một con súc sắc, các kết quả là

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Bây giờ, ví dụ, biến cố "kết quả là số chẵn" tương ứng với tập hợp

$$A = \{2, 4, 6\}.$$

Mỗi kết quả x được gán một xác suất $p(x)$. Khi đó, xác suất $P(A)$ của một biến cố A có thể được tính bằng tổng các xác suất của các kết quả bằng công thức

$$P(A) = \sum_{x \in A} p(x).$$

Ví dụ, khi gieo một con súc sắc, $p(x) = 1/6$ cho mỗi kết quả x , vì vậy xác suất của biến cố "kết quả là số chẵn" là

$$p(2) + p(4) + p(6) = 1/2.$$

Tổng xác suất của các kết quả trong X phải là 1, tức là, $P(X) = 1$.

Vì các biến cố trong lý thuyết xác suất là các tập hợp, chúng ta có thể thao tác với chúng bằng các phép toán tập hợp tiêu chuẩn:

- **Phần bù (complement)** \bar{A} có nghĩa là "A không xảy ra". Ví dụ, khi gieo một con súc sắc, phần bù của $A = \{2, 4, 6\}$ là $\bar{A} = \{1, 3, 5\}$.
- **Hợp (union)** $A \cup B$ có 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 (intersection)** $A \cap B$ có nghĩa là "A và B 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, chúng ta có thể giải quyết một bài toán dễ dàng bằng cách sử dụng phần bù bằng cách giải bài toán ngược lại. Ví dụ, xác suất nhận được ít nhất một mặt sáu khi gieo súc sắc mười lần là

$$1 - (5/6)^{10}.$$

Ở đây $5/6$ là xác suất mà kết quả của một lần gieo không phải là sáu, và $(5/6)^{10}$ là xác suất mà không có lần nào trong mười lần gieo là sáu. Phần bù của điều này là câu trả lời cho 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 súc sắc, hợp của các biến cố

$$A = \text{"kết quả là số chẵn"}$$

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 biến cố A và B là **rời nhau (disjoint)**, tức là, $A \cap B$ là tập rỗng, xác suất của biến cố $A \cup B$ đơn giản là

$$P(A \cup B) = P(A) + P(B).$$

Conditional probability

Xác suất có điều kiện (conditional probability)

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

là xác suất của A giả sử rằng B xảy ra. Do đó, khi tính xác suất của A , chúng ta chỉ xem xét các kết quả cũng thuộc về B .

Sử dụng các tập hợp trước đó,

$$P(A|B) = 1/3,$$

bởi vì các kết quả của B là $\{1, 2, 3\}$, và một trong số đó là chẵn. Đây là xác suất của một kết quả chẵn nếu chúng ta biết rằng kết quả nằm trong khoảng $1 \dots 3$.

Giao

Sử dụng 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 biến cố A và B là **độc lập (independent)** nếu

$$P(A|B) = P(A) \quad \text{và} \quad P(B|A) = P(B),$$

điều này có nghĩa là việc B xảy ra không 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 biến cố

$$A = \text{"chất là chuồn"}$$

và

$$B = \text{"giá trị là bốn"}$$

là độc lập. Do đó biến cố

$$A \cap B = \text{"lá bài là bốn chuồn"}$$

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 (Random variables)

Một **biến ngẫu nhiên (random variable)** là một giá trị được tạo ra bởi một quá trình ngẫu nhiên. Ví dụ, khi gieo hai con súc sắc, một biến ngẫu nhiên có thể là

$$X = \text{"tổng của các kết quả"}.$$

Ví dụ, nếu các kết quả là $[4, 6]$ (nghĩa là chúng ta gieo được mặt bốn trước rồi đến mặt sáu), thì giá trị của X là 10.

Chúng ta ký hiệu $P(X = x)$ là xác suất mà giá trị của một biến ngẫu nhiên X là x . Ví dụ, khi gieo hai con súc sắc, $P(X = 10) = 3/36$, bởi vì tổng số kết quả là 36 và có ba cách có thể để có được tổng 10: $[4, 6]$, $[5, 5]$ và $[6, 4]$.

Giá trị kỳ vọng (Expected value)

Giá trị kỳ vọng (expected value) $E[X]$ chỉ ra 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 duyệt qua tất cả các giá trị có thể của X .

Ví dụ, khi gieo một con súc sắc, 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 thuộc tính hữu ích của giá trị kỳ vọng là **tính tuyến tính (linearity)**. Nó có 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 đúng ngay cả khi các biến ngẫu nhiên phụ thuộc vào nhau.

Ví dụ, khi gieo hai con súc sắc, tổng kỳ vọng là

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Bây giờ chúng ta hãy xem xét một bài toán trong đó n quả bóng được đặt ngẫu nhiên vào n hộp, và nhiệm vụ của chúng ta là tính số hộp rỗng kỳ vọng. Mỗi quả bóng có xác suất bằng 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 bị rỗng là

$$\left(\frac{n-1}{n}\right)^n,$$

bởi vì không có quả bóng nào được đặt vào đó. Do đó, sử dụng tính tuyến tính, số hộp rỗng kỳ vọng là

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Phân phối (Distributions)

Phân phối (distribution) của một biến ngẫu nhiên X cho thấy xác suất của mỗi giá trị mà X có thể có. Phân phối bao gồm các giá trị $P(X = x)$. Ví dụ, khi gieo hai con súc sắc, phân phối cho tổng của chúng 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 (uniform distribution)**, 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 súc sắc, $a = 1$, $b = 6$ và $P(X = x) = 1/6$ cho 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 (binomial distribution)**, 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 chúng ta có thể chọn thứ tự của các lần thử.

Ví dụ, khi gieo một con súc sắc 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 (geometric distribution)**, xác suất để một lần thử thành công là p , và chúng ta tiếp tục cho đến khi lần 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 chúng ta gieo một con súc sắc cho đến khi gieo được 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

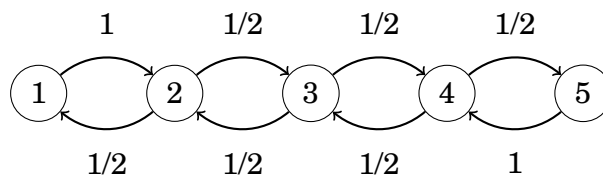
$$E[X] = \frac{1}{p}.$$

24.4 Chuỗi Markov (Markov chains)

Một **chuỗi Markov (Markov chain)** là một quá trình ngẫu nhiên bao gồm các trạng thái và các chuyển tiếp giữa chúng. Đối với mỗi trạng thái, chúng ta biết xác suất để di chuyển đến các trạng thái khác. Một chuỗi Markov có thể được biểu diễn dưới dạng một đồ thị có các nút là trạng thái và các cạnh là các chuyển tiếp.

Ví dụ, hãy xem xét một bài toán trong đó chúng ta đang ở tầng 1 của một tòa nhà n tầng. Ở mỗi bước, chúng ta ngẫu nhiên đi lên một tầng hoặc đi xuống một tầng, ngoại trừ việc chúng ta luôn đi lên một tầng từ tầng 1 và đi xuống một tầng từ tầng n . Xác suất 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 một chuỗi Markov. Ví dụ, nếu $n = 5$, đồ thị như sau:



Phân phối xác suất của một chuỗi Markov là một vector $[p_1, p_2, \dots, p_n]$, trong đó p_k là xác suất mà trạng thái hiện tại là k . Công thức $p_1 + p_2 + \dots + p_n = 1$ luôn đúng.

Trong kịch bản trên, phân phối ban đầu là $[1, 0, 0, 0, 0]$, vì chúng ta luôn bắt đầu ở tầng 1. Phân phối tiếp theo là $[0, 1, 0, 0, 0]$, vì chúng ta chỉ có thể di chuyển từ tầng 1 đến tầng 2. Sau đó, chúng ta có thể đi lên một tầng hoặc đi xuống một tầng, vì vậy phân phối tiếp theo là $[1/2, 0, 1/2, 0, 0]$, và cứ thế.

Một cách hiệu quả để mô phỏng việc di chuyển trong một chuỗi Markov là sử dụng quy hoạch động. Ý tưởng là duy trì phân phối xác suất, và ở mỗi bước duyệt qua tất cả các khả năng chúng ta có thể di chuyển. Sử dụng phương pháp này, chúng ta có thể mô phỏng một cuộc đi bộ m bước trong thời gian $O(n^2m)$.

Các chuyển tiếp của một chuỗi Markov cũng có thể được biểu diễn dưới dạng một ma trận cập nhật phân phối xác suất. Trong kịch bản 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 chúng ta nhân một phân phối xác suất với ma trận này, chúng ta sẽ nhận được phân phối mới sau khi di chuyển một bước. Ví dụ, chúng ta có thể di chuyển từ phân phối $[1, 0, 0, 0, 0]$ đến 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 một cách hiệu quả, chúng ta có thể tính phân phối sau m bước trong thời gian $O(n^3 \log m)$.

24.5 Thuật toán ngẫu nhiên (Randomized algorithms)

Đôi khi chúng ta có thể sử dụng tính ngẫu nhiên để giải quyết một bài toán, ngay cả khi bài toán đó không liên quan đến xác suất. Một **thuật toán ngẫu nhiên (randomized algorithm)** là một thuật toán dựa trên sự ngẫu nhiên.

Một **thuật toán Monte Carlo (Monte Carlo algorithm)** là một thuật toán ngẫu nhiên đôi khi có thể cho ra câu trả lời sai. Để một thuật toán như vậy hữu ích, xác suất của một câu trả lời sai phải nhỏ.

Một **thuật toán Las Vegas (Las Vegas algorithm)** là một thuật toán ngẫu nhiên luôn cho ra câu trả lời đúng, nhưng thời gian chạy của nó thay đổi một cách ngẫu nhiên. Mục tiêu là thiết kế một thuật toán hiệu quả với xác suất cao.

Tiếp theo chúng ta sẽ xem xét ba bài toán ví dụ có thể được giải quyết bằng cách sử dụng tính ngẫu nhiên.

Thông kê thứ tự (Order statistics)

Thông 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. Dễ dàng tính toán bất kỳ thông kê thứ tự nào trong thời gian $O(n \log n)$ bằng cách sắp xếp mảng trước, nhưng liệu có thực sự cần thiết phải sắp xếp toàn bộ mảng chỉ để tìm một phần tử?

Hóa ra chúng ta có thể tìm thông kê thứ tự sử dụng một thuật toán ngẫu nhiên mà không cần sắp xếp mảng. Thuật toán, được 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, và di chuyển các phần tử nhỏ hơn x sang phần bên trái của mảng, và tất cả các phần tử khác sang phần bên phải của mảng. Việc này mất thời gian $O(n)$ khi có n phần tử. Giả sử phần bên trái chứa a phần tử

¹Năm 1961, C. A. R. Hoare đã công bố hai thuật toán hiệu quả trên trung bình: **quicksort** [36] để sắp xếp mảng và **quickselect** [37] để tìm thông kê thứ tự.

và phần bên phải chứa b phần tử. Nếu $a = k$, phần tử x là thống kê thứ tự thứ k . Ngược lại, nếu $a > k$, chúng ta đệ quy tìm thống kê thứ tự thứ k cho phần bên trái, và nếu $a < k$, chúng ta đệ quy tìm thống kê thứ tự thứ r cho phần bên phải trong đó $r = k - a$. Việc tìm kiếm tiếp tục một cách tương tự, cho đến khi phần tử được tìm thấy.

Khi mỗi phần tử x được chọn ngẫu nhiên, kích thước của mảng giảm đi khoảng một nửa ở mỗi bước, vì vậy độ phức tạp thời gian để tìm thống kê thứ tự thứ k là khoảng

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

Trường hợp xấu nhất của thuật toán vẫn yêu cầu thời gian $O(n^2)$, bởi vì có thể x luôn được chọn theo cách mà nó là một trong những phần tử nhỏ nhất hoặc lớn nhất trong mảng và cần $O(n)$ bước. Tuy nhiên, xác suất cho điều này nhỏ đến mức nó không bao giờ xảy ra trong thực tế.

Kiểm tra phép nhân ma trận (Verifying matrix multiplication)

Bài toán tiếp theo của chúng ta là *kiểm tra* liệu $AB = C$ có đúng không khi A , B và C là các ma trận kích thước $n \times n$. Tất nhiên, chúng ta có thể giải quyết bài toán bằng cách tính lại tích AB (trong thời gian $O(n^3)$ sử dụng thuật toán cơ bản), nhưng người ta có thể hy vọng rằng việc kiểm tra câu trả lời sẽ dễ dàng hơn là tính toán nó từ đầu.

Hóa ra chúng ta có thể giải quyết bài toán sử dụng một thuật toán Monte Carlo² mà độ phức tạp thời gian chỉ là $O(n^2)$. Ý tưởng rất đơn giản: chúng ta chọn một vector ngẫu nhiên X gồm n phần tử, và tính các ma trận ABX và CX . Nếu $ABX = CX$, chúng ta báo cáo rằng $AB = C$, và ngược lại chúng ta báo cáo rằng $AB \neq C$.

Độ phức tạp thời gian của thuật toán là $O(n^2)$, bởi vì chúng ta có thể tính các ma trận ABX và CX trong thời gian $O(n^2)$. Chúng ta có thể tính ma trận ABX một cách hiệu quả bằng cách sử dụng biểu diễn $A(BX)$, vì vậy 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 rằng thuật toán mắc lỗi khi nó báo cáo rằng $AB = C$. Ví dụ,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

nhưng

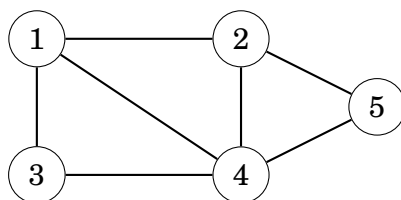
$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

Tuy nhiên, trong thực tế, xác suất mà thuật toán mắc lỗi là nhỏ, và chúng ta có thể giảm xác suất này bằng cách kiểm tra kết quả bằng nhiều vector ngẫu nhiên X trước khi báo cáo rằng $AB = C$.

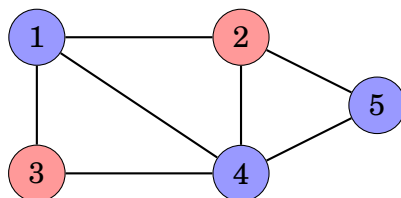
Tô màu đồ thị (Graph coloring)

Cho một đồ thị chứa n nút và m cạnh, nhiệm vụ của chúng ta là tìm một cách tô màu các nút của đồ thị bằng hai màu sao cho với ít nhất $m/2$ cạnh, các điểm cuối có màu khác nhau. Ví dụ, trong đồ thị

²R. M. Freivalds đã công bố thuật toán này vào năm 1977 [26], và nó đôi khi được gọi là **thuật toán Freivalds (Freivalds' algorithm)**.



một cách tô màu hợp lệ như sau:



Đồ thị trên chứa 7 cạnh, và với 5 trong số chúng, các điểm cuối có màu khác nhau, vì vậy việc tô màu là hợp lệ.

Bài toán có thể được giải quyết bằng một thuật toán Las Vegas tạo ra các cách tô màu ngẫu nhiên cho đến khi một cách tô màu hợp lệ được tìm thấy. Trong một cách tô màu ngẫu nhiên, màu của mỗi nút được chọn độc lập sao cho xác suất của cả hai màu là $1/2$.

Trong một cách tô màu ngẫu nhiên, xác suất để các điểm cuối của một cạnh đơn lẻ có màu khác nhau là $1/2$. Do đó, số cạnh kỳ vọng có các điểm cuối có màu khác nhau là $m/2$. Vì kỳ vọng rằng một cách tô màu ngẫu nhiên là hợp lệ, chúng ta sẽ nhanh chóng tìm thấy một cách tô màu hợp lệ trong thực tế.

Chương 25

Lý thuyết trò chơi (Game theory)

Trong chương này, chúng ta sẽ tập trung vào các trò chơi hai người chơi không chứa các yếu tố ngẫu nhiên. Mục tiêu của chúng ta là tìm ra một chiến lược mà chúng ta có thể tuân 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 chung cho các trò chơi như vậy, và chúng ta có thể phân tích các trò chơi bằng cách sử dụng **lý thuyết Nim (nim theory)**. Đầu tiên, chúng ta sẽ phân tích các trò chơi đơn giản trong đó người chơi lấy que khỏi các đồng, và sau đó, chúng ta sẽ tổng quát hóa chiến lược được sử dụng trong các trò chơi đó cho các trò chơi khác.

25.1 Trạng thái trò chơi (Game states)

Hãy xem xét một trò chơi ban đầu có một đồng gồm n que. Người chơi A và B di chuyển luân phiên, và người chơi A bắt đầu. Trong mỗi lượt đi, 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 lại 8 que).
- Người chơi B lấy 3 que (còn lại 5 que).
- Người chơi A lấy 1 que (còn lại 4 que).
- Người chơi B lấy 2 que (còn lại 2 que).
- Người chơi A lấy 2 que và thắng.

Trò chơi này bao 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 (winning state)** là một trạng thái mà người chơi sẽ thắng trò chơi nếu họ chơi một cách tối ưu, và một **trạng thái thua (losing state)** là một trạng thái mà người chơi sẽ thua trò chơi nếu đối thủ chơi một cách tối ưu. Hóa ra chúng ta có thể phân loại tất cả các trạng thái của một trò chơi sao cho mỗi trạng thái hoặc là một trạng thái thắng hoặc là một trạng thái thua.

Trong trò chơi trên, trạng thái 0 rõ ràng là một trạng thái thua, vì người chơi không thể thực hiện bất kỳ nước đi nào. Các trạng thái 1, 2 và 3 là các trạng thái thắng, bởi vì chúng ta có thể lấy 1, 2 hoặc 3 que và thắng trò chơi. Trạng thái 4, ngược lại, là một trạng thái thua, bởi vì bất kỳ nước đi nào cũng dẫn đến một trạng thái là 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 một trạng thái thắng, và ngược lại, trạng thái hiện tại là một

trạng thái thua. Sử dụng quan sát này, chúng ta có thể phân loại tất cả các trạng thái của một trò chơi bắt đầu từ các trạng thái thua nơi không có nước đi nào khả dĩ.

Các trạng thái $0 \dots 15$ của trò chơi trên có thể được phân loại như sau (W biểu thị một trạng thái thắng và L biểu thị một 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

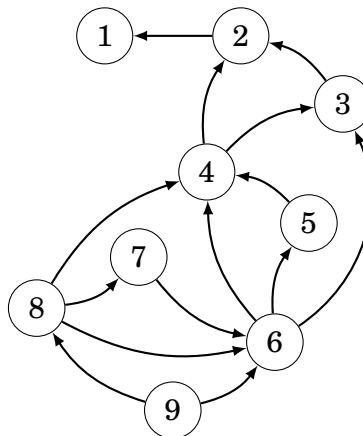
Dễ dàng phân tích trò chơi này: một trạng thái k là một trạng thái thua nếu k chia hết cho 4, và ngược lại nó là một trạng thái thắng. Một cách chơi tối ưu là luôn chọn một nước đi sao cho sau đó số que trong đồng chia hết cho 4. Cuối cùng, sẽ không còn que nào và đối thủ đã thua.

Tất nhiên, chiến lược này yêu cầu số que *không* chia hết cho 4 khi đến lượt chúng ta đi. Nếu nó chia hết, chúng ta không thể làm gì, và đối thủ sẽ thắng trò chơi nếu họ chơi một cách tối ưu.

Đồ thị trạng thái (State graph)

Bây giờ chúng ta hãy xem xét một trò chơi que khác, trong đó ở mỗi trạng thái k , được phép lấy đi một số lượng que x bất kỳ sao cho x nhỏ hơn k và là ước của k . Ví dụ, ở trạng thái 8 chúng 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 cho thấy các trạng thái $1 \dots 9$ của trò chơi dưới dạng một **đồ thị trạng thái (state graph)**, có các nút là các trạng thái và các cạnh là các nước đi giữa chúng:



Trạng thái cuối cùng trong trò chơi này luôn là trạng thái 1, là một trạng thái thua, vì không có nước đi hợp lệ nào. 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

Thật đáng ngạc nhiên, trong trò chơi này, tất cả các trạng thái có số chẵn đều là trạng thái thắng, và tất cả các trạng thái có số lẻ đều là trạng thái thua.

25.2 Trò chơi Nim (Nim game)

Trò chơi Nim (nim game) là một trò chơi đơn giản có vai trò quan trọng trong lý thuyết trò chơi, bởi vì nhiều trò chơi khác có thể được chơi bằng cùng một chiến lược. Đầu tiên, chúng ta tập trung vào nim, và sau đó chúng ta tổng quát hóa chiến lược cho các trò chơi khác.

Có n đồng trong nim, và mỗi đồng chứa một số lượng que nhất định. Người chơi đi luân phiên, và trong mỗi lượt, người chơi chọn một đồng vẫn còn que và lấy đi một số lượng que bất kỳ từ 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 với 10, 12 và 5 que. Trạng thái $[0, 0, \dots, 0]$ là một trạng thái thua, bởi vì không thể lấy que nào, và đây luôn là trạng thái cuối cùng.

Phân tích

Hóa ra chúng 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 toán xor¹. Các trạng thái có tổng Nim bằng 0 là các trạng thái thua, và tất cả các trạng thái khác là các trạng thái thắng. Ví dụ, tổng Nim của $[10, 12, 5]$ là $10 \oplus 12 \oplus 5 = 3$, vì vậy trạng thái này là một trạng thái thắng.

Nhưng tổng Nim liên quan đến trò chơi nim như thế nào? Chúng ta có thể giải thích điều này bằng cách xem xét cách tổng Nim thay đổi khi trạng thái nim thay đổi.

Trạng thái thua: Trạng thái cuối cùng $[0, 0, \dots, 0]$ là một trạng thái thua, và tổng Nim của nó là 0, như mong đợi. Trong các trạng thái thua khác, bất kỳ nước đi nào cũng dẫn đến một trạng thái thắng, bởi vì khi một giá trị x_k duy nhất thay đổi, tổng Nim cũng thay đổi, vì vậy tổng Nim sẽ khác 0 sau nước đi.

Trạng thái thắng: Chúng ta có thể di chuyển đến một trạng thái thua nếu có bất kỳ đồng k nào mà $x_k \oplus s < x_k$. Trong trường hợp này, chúng ta có thể lấy que khỏi đồng k sao cho nó sẽ chứa $x_k \oplus s$ que, điều này sẽ dẫn đến một trạng thái thua. Luôn có một đồng như vậy, trong đó x_k có một bit một ở vị trí của bit một trái nhất của s .

Ví dụ, hãy xem xét trạng thái $[10, 12, 5]$. Trạng thái này là một trạng thái thắng, bởi vì tổng Nim của nó là 3. Do đó, phải có một nước đi dẫn đến một trạng thái thua. Tiếp theo chúng ta sẽ tìm ra một nước đi như vậy.

Tổng Nim của trạng thái như sau:

10	1010
12	1100
5	0101
3	0011

Trong trường hợp này, đồng có 10 que là đồng duy nhất có một bit một ở vị trí của bit một trái nhất 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 $10 \oplus 3 = 9$, vì vậy chúng ta sẽ chỉ lấy đi một que. Sau đó, trạng thái sẽ là $[9, 12, 5]$, là một trạng thái thua:

¹Chiến lược tối ưu cho nim được công bố vào năm 1901 bởi C. L. Bouton [10].

9	1001
12	1100
5	0101
0	0000

Trò chơi Misère

Trong một **trò chơi misère (misère game)**, mục tiêu của trò chơi là ngược lại, vì vậy người chơi lấy que cuối cùng sẽ thua trò chơi. Hóa ra trò chơi nim misère có thể được chơi một cách tối ưu gần giống như trò chơi nim tiêu chuẩn.

Ý tưởng là đầu tiên chơi trò chơi misère giống như trò chơi tiêu chuẩn, nhưng thay đổi chiến lược vào cuối trò chơi. Chiến lược mới sẽ được áp dụng trong tình huống mà mỗi đồng sẽ chứa tối đa một que sau nước đi tiếp theo.

Trong trò chơi tiêu chuẩn, chúng ta nên chọn một nước đi sau đó có một số chẵn các đồng có một que. Tuy nhiên, trong trò chơi misère, chúng ta chọn một nước đi sao cho có một số lẻ các đồng có một que.

Chiến lược này hoạt động vì một trạng thái mà chiến lược thay đổi luôn xuất hiện trong trò chơi, và trạng thái này là một trạng thái thắng, bởi vì nó chứa chính xác một đồng có nhiều hơn một que nên tổng Nim không phải là 0.

25.3 Định lý Sprague–Grundy (Sprague–Grundy theorem)

Định lý Sprague–Grundy (Sprague–Grundy theorem)² tổng quát hóa chiến lược được sử dụng trong nim cho tất cả các trò chơi thỏa mãn các yêu cầu sau:

- Có hai người chơi đi luân phiên.
- Trò chơi bao gồm các trạng thái, và các nước đi có thể trong một trạng thái không phụ thuộc vào lượt của ai.
- Trò chơi kết thúc khi một người chơi không thể thực hiện nước đi.
- Trò chơi chắc chắn sẽ kết thúc sớm hay muộn.
- Người chơi có thông tin đầy đủ về các trạng thái và các nước đi được phép, và không có sự ngẫu nhiên trong trò chơi.

Ý tưởng là tính cho mỗi trạng thái trò chơi một số Grundy tương ứng với số que trong một đồng nim. Khi chúng ta biết các số Grundy của tất cả các trạng thái, chúng ta có thể chơi trò chơi như trò chơi nim.

Số Grundy (Grundy numbers)

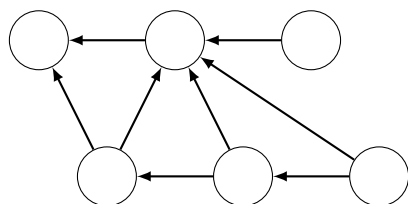
Số Grundy (Grundy number) của một trạng thái trò chơi là

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

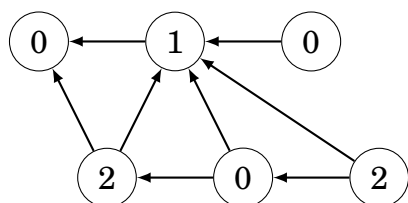
trong đó g_1, g_2, \dots, g_n là các số Grundy của các trạng thái mà chúng ta có thể di chuyển đến, và hàm mex cho ra số không âm nhỏ nhất không có trong tập hợp. Ví dụ, $\text{mex}(\{0, 1, 3\}) = 2$. Nếu không có nước đi nào khả dĩ trong một trạng thái, số Grundy của nó là 0, vì $\text{mex}(\emptyset) = 0$.

Ví dụ, trong đồ thị trạng thái

²Định lý được khám phá độc lập bởi R. Sprague [61] và P. M. Grundy [31].



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 là 0, chúng ta chỉ có thể di chuyển đến các trạng thái có số Grundy dương, và nếu số Grundy là $x > 0$, chúng ta có thể di chuyển đến các trạng thái có số Grundy bao gồm tất cả các số $0, 1, \dots, x - 1$.

Ví dụ, hãy xem xét một trò chơi trong đó người chơi di chuyển một quân cờ trong một mê cung. Mỗi ô trong mê cung là sàn hoặc tường. Trong mỗi lượt, người chơi phải di chuyển quân cờ một số bước sang trái hoặc lên trên. Người thắng trò chơi là người thực hiện nước đi cuối cùng.

Hình sau cho thấy một trạng thái ban đầu có thể có của trò chơi, trong đó @ biểu thị quân cờ và * biểu thị một ô mà nó có thể di chuyển đến.

				*
				*
*	*	*	*	@

Các trạng thái của trò chơi là tất cả các ô sàn của 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 cho ô dưới cùng bên phải là 2, vì vậy đó là một trạng thái thắng. Chúng ta có thể đến một trạng thái thua và thắng trò chơi bằng cách di chuyển hoặc bốn bước sang trái hoặc hai bước lên trên.

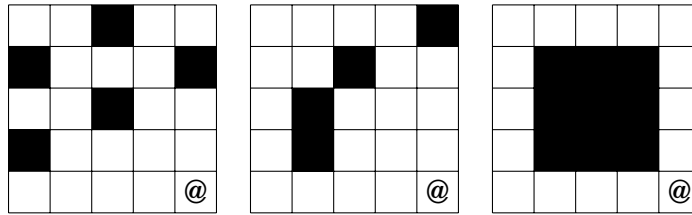
Lưu ý rằng không giống như trong trò chơi nim ban đầu, có thể di chuyển đến một trạng thái có số Grundy lớn hơn số Grundy của trạng thái hiện tại. Tuy nhiên, đối thủ luôn có thể chọn một nước đi hủy bỏ một nước đi như vậy, vì vậy không thể thoát khỏi một trạng thái thua.

Trò chơi con (Subgames)

Tiếp theo chúng ta sẽ giả sử rằng trò chơi của chúng ta bao gồm các trò chơi con, và trong mỗi lượt, người chơi đầu tiên chọn một trò chơi con và sau đó là một nước đi trong trò chơi con đó. Trò chơi kết thúc khi không thể thực hiện bất kỳ nước đi nào trong bất kỳ trò chơi con nào.

Trong trường hợp này, số Grundy của một 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 các số Grundy cho các trò chơi con và sau đó là tổng Nim của chúng.

Ví dụ, hãy xem xét một trò chơi bao gồm ba mê cung. Trong trò chơi này, trong mỗi lượt, người chơi chọn một trong các mê cung và sau đó di chuyển quân cờ trong mê cung đó. Giả sử trạng thái ban đầu của trò chơi như sau:



Các số Grundy cho 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

Ở trạng thái ban đầu, tổng Nim của các số Grundy là $2 \oplus 3 \oplus 3 = 2$, vì vậy người chơi đầu tiên có thể thắng trò chơi. Một nước đi tối ưu là di chuyển hai bước lên trên trong mê cung đầu tiên, tạo ra tổng Nim $0 \oplus 3 \oplus 3 = 0$.

Trò chơi của Grundy

Đôi khi một nước đi trong một 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 tạo ra các trò chơi con với các số Grundy $a_{k,1}, a_{k,2}, \dots, a_{k,m}$.

Một ví dụ về trò chơi như vậy là **trò chơi của Grundy (Grundy's game)**. Ban đầu, có một đồng duy nhất chứa n que. Trong mỗi lượt, người chơi chọn một đồng và chia nó thành hai đồng không rỗng sao cho các đồng có kích thước khác nhau. Người chơi thực hiện nước đi 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 qua tất cả các cách chia đồng thành hai đồng. Ví dụ, khi $n = 8$, các khả năng là $1 + 7$, $2 + 6$ và $3 + 5$, vì vậy

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

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 $f(1) = f(2) = 0$, bởi vì không thể chia các đồng có 1 và 2 que. Các số Grundy đầu tiên là:

$$f(1) = 0$$

$$f(2) = 0$$

$$f(3) = 1$$

$$f(4) = 0$$

$$f(5) = 2$$

$$f(6) = 1$$

$$f(7) = 0$$

$$f(8) = 2$$

Số Grundy cho $n = 8$ là 2, vì vậy có thể thắng trò chơi. Nước đi thắng là tạo ra các đồng 1 + 7, vì $f(1) \oplus f(7) = 0$.

Chương 26

Các thuật toán chuỗi

Chương này đề cập đến các thuật toán hiệu quả để xử lý chuỗi. Nhiều bài toán về chuỗi có thể được giải quyết dễ dàng trong thời gian $O(n^2)$, nhưng thách thức là tìm ra các thuật toán hoạt động 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 có độ dài n và một mẫu có độ dài m , nhiệm vụ của chúng ta là tìm các lần xuất hiện của mẫu trong chuỗi. Ví dụ, mẫu ABC xuất hiện hai lần trong chuỗi ABABCBABC.

Bài toán khớp mẫu có thể được giải quyết dễ dàng trong thời gian $O(nm)$ bằng một thuật toán duyệt toàn bộ (brute force) kiểm tra tất cả các vị trí mà mẫu có thể xuất hiện trong chuỗi. Tuy nhiên, trong chương này, chúng ta sẽ thấy rằng có các thuật toán hiệu quả hơn chỉ yêu cầu thời gian $O(n + m)$.

26.1 Thuật ngữ chuỗi

Trong suốt chương này, chúng ta giả định rằng việc đánh chỉ số bắt đầu từ 0 được sử dụng trong các chuỗi. Do đó, một chuỗi s có độ dài n bao gồm các ký tự $s[0], s[1], \dots, s[n-1]$. Tập hợp các ký tự có thể xuất hiện trong chuỗi được gọi là **bảng chữ cái (alphabet)**. Ví dụ, bảng chữ cái $\{A, B, \dots, Z\}$ bao gồm các chữ cái viết hoa của tiếng Anh.

Một **chuỗi con (substring)** là một dãy các ký tự liên tiếp trong một chuỗi. Chúng ta sử dụng 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 có độ dài n có $n(n+1)/2$ chuỗi con. Ví dụ, các chuỗi con của ABCD là A, B, C, D, AB, BC, CD, ABC, BCD và ABCD.

Một **dãy con (subsequence)** là một dãy các ký tự (không nhất thiết phải liên tiếp) trong một chuỗi theo thứ tự ban đầu của chúng. Một chuỗi có 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.

Một **tiền tố (prefix)** là một chuỗi con bắt đầu từ đầu của một chuỗi, và một **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.

Một **phép xoay vòng (rotation)** có thể được tạo ra bằng cách di chuyển các ký tự của một chuỗi từng ký tự một từ đầu đến cuối (hoặc ngược lại). Ví dụ, các phép xoay vòng của ABCD là ABCD, BCDA, CDAB và DABC.

Một **chu kỳ (period)** là một tiền tố của một chuỗi sao cho chuỗi đó có thể được xây dựng bằng cách lặp lại chu kỳ đó. Lặp cuối cùng có thể là một phần và chỉ chứa một tiền tố của chu kỳ. Ví dụ, chu kỳ ngắn nhất của ABCABCA là ABC.

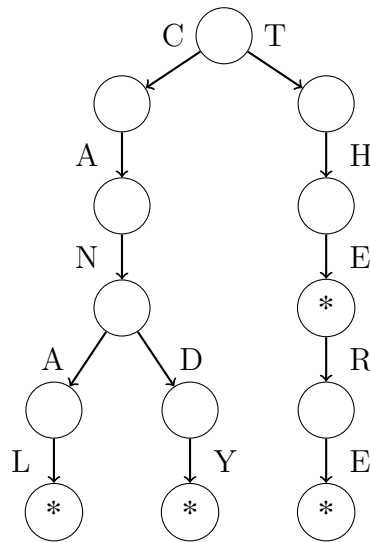
Một **biên (border)** là một chuỗi vừa là một tiền tố vừa là một hậu tố của một chuỗi. Ví dụ, các biên của ABACABA là A, ABA và ABACABA.

Các chuỗi được so sánh bằng cách sử dụng **thứ tự từ điển (lexicographical order)** (tương ứng với thứ tự chữ cái). Điều đó có nghĩa là $x < y$ nếu $x \neq y$ và x là tiền tố của y , hoặc có một vị trí k sao cho $x[i] = y[i]$ khi $i < k$ và $x[k] < y[k]$.

26.2 Cấu trúc Trie

Một **cây trie (trie)** là một cây có gốc duy trì một tập hợp các chuỗi. Mỗi chuỗi trong tập hợp được lưu trữ dưới dạng một chuỗi các ký tự bắt đầu từ gốc. Nếu hai chuỗi có một tiền tố chung, chúng cũng có một chuỗi chung trong cây.

Ví dụ, hãy xem xét cây trie sau:



Cây trie này tương ứng với tập hợp {CANAL, CANDY, THE, THERE}. Ký tự * trong một nút có nghĩa là một chuỗi trong tập hợp kết thúc tại nút đó. Một ký tự như vậy là cần thiết, bởi vì một chuỗi có thể là một tiền tố của một chuỗi khác. Ví dụ, trong cây trie trên, THE là một tiền tố của THERE.

Chúng ta có thể kiểm tra trong thời gian $O(n)$ xem một cây trie có chứa một chuỗi có độ dài n hay không, bởi vì chúng ta có thể đi theo chuỗi bắt đầu từ nút gốc. Chúng ta cũng có thể thêm một chuỗi có độ dài n vào cây trie trong thời gian $O(n)$ bằng cách đi theo chuỗi trước và sau đó thêm các nút mới vào cây nếu cần.

Sử dụng cây trie, chúng ta có thể tìm tiền tố dài nhất của một chuỗi cho trước sao cho tiền tố đó thuộc tập hợp. Hơn nữa, bằng cách lưu trữ thêm thông tin trong mỗi nút, chúng ta có thể tính số lượng chuỗi thuộc tập hợp và có một chuỗi cho trước làm tiền tố.

Một cây trie có thể được lưu trữ 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 hợp) và A là kích thước của bảng chữ cái. Các nút của một cây trie được đánh số $0, 1, 2, \dots$ sao cho số của gốc là 0 , và $\text{trie}[s][c]$ là nút tiếp theo trong chuỗi khi chúng ta di chuyển từ nút s bằng ký tự c .

26.3 Băm chuỗi (String hashing)

Băm chuỗi (String hashing) là một kỹ thuật cho phép chúng ta kiểm tra hiệu quả xem hai chuỗi có bằng nhau không¹. Ý tưởng trong băm chuỗi là so sánh các giá trị băm của các chuỗi thay vì các ký tự riêng lẻ của chúng.

Tính giá trị băm

Một **giá trị băm (hash value)** của một chuỗi là một số được tính từ các ký tự của chuỗi. Nếu hai chuỗi giống nhau, các giá trị băm của chúng cũng giống nhau, điều này cho phép so sánh các chuỗi dựa trên giá trị băm của chúng.

Một cách thông thường để thực hiện băm chuỗi là **băm đa thức (polynomial hashing)**, có nghĩa là giá trị băm của một chuỗi s có 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 của s , và A và B là các hằng số được chọn trước.

Ví dụ, mã của các ký tự của 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

Sử dụng băm đa thức, chúng ta có thể tính giá trị băm của bất kỳ chuỗi con nào của một chuỗi s trong thời gian $O(1)$ sau một lần tiền xử lý thời gian $O(n)$. 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 mảng có thể được tính đệ quy như sau:

$$\begin{aligned} h[0] &= s[0] \\ h[k] &= (h[k-1]A + s[k]) \bmod B \end{aligned}$$

Ngoài ra, chúng 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 nào $s[a \dots b]$ có thể được tính trong thời gian $O(1)$ bằng công thức

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

giả sử rằng $a > 0$. Nếu $a = 0$, giá trị băm đơn giản là $h[b]$.

¹Kỹ thuật này đã được phổ biến bởi thuật toán khớp mẫu Karp–Rabin [42].

Sử dụng giá trị băm

Chúng ta có thể so sánh các chuỗi một cách hiệu quả bằng cách sử dụng giá trị băm. Thay vì so sánh các ký tự riêng lẻ của các chuỗi, ý tưởng là so sánh các giá trị băm của chúng. Nếu các giá trị băm bằng nhau, các chuỗi *có thể* bằng nhau, và nếu các giá trị băm khác nhau, các chuỗi *chắc chắn* khác nhau.

Sử dụng băm, chúng ta thường có thể làm cho một thuật toán duyệt toàn bộ trở nên hiệu quả. Ví dụ, hãy xem 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 duyệt toàn bộ sẽ duyệt qua tất cả các vị trí mà p có thể xuất hiện và so sánh các chuỗi từng ký tự một. Độ phức tạp thời gian của một thuật toán như vậy là $O(n^2)$.

Chúng ta có thể làm cho thuật toán duyệt toàn bộ hiệu quả hơn bằng cách sử dụng băm, bởi vì thuật toán so sánh các chuỗi con của các chuỗi. Sử dụng băm, mỗi phép so sánh chỉ mất thời gian $O(1)$, bởi vì chỉ có giá trị băm của các chuỗi con được so sánh. Điều này dẫn đến một thuật toán có độ phức tạp thời gian là $O(n)$, là độ phức tạp thời gian tốt nhất có thể cho bài toán này.

Bằng cách kết hợp băm và *tìm kiếm nhị phân (binary search)*, cũng có thể tìm ra 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 các chuỗi bằng tìm kiếm nhị phân. Một khi chúng ta biết độ dài của tiền tố chung, chúng ta chỉ cần kiểm tra ký tự tiếp theo sau tiền tố, bởi vì điều này xác định thứ tự của các chuỗi.

Xung đột và tham số

Một rủi ro rõ ràng khi so sánh các giá trị băm là một **xung đột (collision)**, có nghĩa là hai chuỗi có nội dung khác nhau nhưng giá trị băm bằng nhau. Trong trường hợp này, một thuật toán dựa trên các giá trị băm kết luận rằng các chuỗi bằng nhau, nhưng thực tế chúng không phải vậy, và thuật toán có thể cho kết quả không chính xác.

Xung đột luôn có thể xảy ra, bởi vì số lượng các chuỗi khác nhau lớn hơn số lượng các giá trị băm khác nhau. Tuy nhiên, xác suất xảy ra xung đột là nhỏ nếu các hằng số A và B được chọn cẩn thận. Một cách thông thường là chọn các hằng số ngẫu nhiên gần 10^9 , ví dụ như sau:

$$\begin{aligned}A &= 911382323 \\ B &= 972663749\end{aligned}$$

Sử dụng các hằng số như vậy, kiểu long long có thể được sử dụng khi tính toán các giá trị băm, bởi vì các tích AB và BB sẽ vừa với long long. Nhưng liệu có đủ để có khoảng 10^9 giá trị băm khác nhau không?

Hãy xem xét ba kịch bản mà băm có thể được sử dụng:

Kịch bản 1: Chuỗi x và y được so sánh với nhau. Xác suất xảy ra xung đột là $1/B$ giả sử rằng tất cả các giá trị băm đều có khả năng xảy ra như nhau.

Kịch bản 2: Một chuỗi x được so sánh với các chuỗi y_1, y_2, \dots, y_n . Xác suất của một hoặc nhiều xung đột là

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

Kịch bản 3: Tất cả các cặp chuỗi x_1, x_2, \dots, x_n được so sánh với nhau. Xác suất của một hoặc nhiều xung đột là

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Bảng sau cho thấy xác suất xung đột khi $n = 10^6$ và giá trị của B thay đổi:

hàng số B	kịch bản 1	kịch bản 2	kịch bản 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 rằng trong kịch bản 1, xác suất xảy ra xung đột là không đáng kể khi $B \approx 10^9$. Trong kịch bản 2, một xung đột có thể xảy ra nhưng xác suất vẫn khá nhỏ. Tuy nhiên, trong kịch bản 3, tình hình rất khác: một xung đột gần như luôn luôn xảy ra khi $B \approx 10^9$.

Hiện tượng trong kịch bản 3 được biết đến với tên gọi **ngịch lý ngày sinh (birthday paradox)**: nếu có n người trong một căn phòng, xác suất để *một vài* hai người có cùng ngày sinh là lớn ngay cả khi n khá nhỏ. Tương ứng, trong băm, khi tất cả các giá trị băm được so sánh với nhau, xác suất để một vài hai giá trị băm bằng nhau là lớn.

Chúng ta có thể làm cho xác suất xảy ra xung đột nhỏ hơn bằng cách tính *nhiều* giá trị băm sử dụng các tham số khác nhau. Không có khả năng một xung đột sẽ xảy ra ở tất cả các giá trị băm cùng một lúc. Ví dụ, hai giá trị băm với tham số $B \approx 10^9$ tương ứng với một giá trị băm với tham số $B \approx 10^{18}$, điều này làm cho xác suất xảy ra xung đột rất nhỏ.

Một số người sử dụng các hằng số $B = 2^{32}$ và $B = 2^{64}$, điều này thuận tiện, bởi vì các phép toán với các số nguyên 32 và 64 bit được tính theo modulo 2^{32} và 2^{64} . Tuy nhiên, đây *không phải* là một lựa chọn tốt, bởi vì có thể xây dựng các đầu vào luôn tạo ra xung đột khi các hằng số có dạng 2^x được sử dụng [51].

26.4 Thuật toán Z (Z-algorithm)

Mảng Z (Z-array) z của một chuỗi s có độ dài n chứa cho mỗi $k = 0, 1, \dots, n-1$ độ dài của chuỗi con dài nhất của s bắt đầu tại vị trí k và là một tiền tố của s . Do đó, $z[k] = p$ cho chúng ta biết rằng $s[0 \dots p-1]$ bằng $s[k \dots k+p-1]$. Nhiều bài toán xử lý chuỗi có thể được giải quyết hiệu quả bằng cách sử dụng 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, ví dụ, $z[6] = 5$, bởi vì chuỗi con ACBAC có độ dài 5 là một tiền tố của s , nhưng chuỗi con ACBACB có độ dài 6 không phải là một tiền tố của s .

Mô tả thuật toán

Tiếp theo chúng ta mô tả một thuật toán, được gọi là **thuật toán Z (Z-algorithm)**², xây dựng mảng Z một cách hiệu quả trong thời gian $O(n)$. Thuật toán tính toán các giá trị mảng Z từ trái sang phải bằng cách vừa sử dụng thông tin đã được lưu trữ trong mảng Z vừa so sánh các chuỗi con từng ký tự một.

²Thuật toán Z được trình bày trong [32] là phương pháp đơn giản nhất được biết đến cho việc khớp mẫu trong thời gian tuyến tính, và ý tưởng ban đầu được cho là của [50].

Để tính toán hiệu quả các giá trị mảng Z , thuật toán duy trì một khoảng $[x, y]$ sao cho $s[x \dots y]$ là một tiền tố của s và y lớn nhất có thể. Vì chúng ta biết rằng $s[0 \dots y-x]$ và $s[x \dots y]$ bằng nhau, chúng ta có thể sử dụng thông tin này khi tính các giá trị Z cho các vị trí $x+1, x+2, \dots, y$.

Tại mỗi vị trí k , chúng ta trước tiên kiểm tra giá trị của $z[k-x]$. Nếu $k+z[k-x] < y$, chúng 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 $z[k]$ chúng ta cần so sánh các chuỗi con từng ký tự một. Tuy nhiên, thuật toán vẫn hoạt động trong thời gian $O(n)$, bởi vì chúng ta bắt đầu so sánh tại các vị trí $y-k+1$ và $y+1$.

Ví dụ, chúng ta 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 $z[6] = 5$, khoảng $[x, y]$ hiện tại là $[6, 10]$:

0	1	2	3	4	5	$x \quad y$					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	?	?	?	?	?	?	?	?	?

Bây giờ chúng ta có thể tính các giá trị mảng Z tiếp theo một cách hiệu quả, bởi vì chúng ta biết rằng $s[0 \dots 4]$ và $s[6 \dots 10]$ bằng nhau. Đầu tiên, vì $z[1] = z[2] = 0$, chúng ta ngay lập tức biết rằng cũng $z[7] = z[8] = 0$:

0	1	2	3	4	5	$x \quad y$					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	?	?	?	?	?	?	?



Sau đó, vì $z[3] = 2$, chúng ta biết rằng $z[9] \geq 2$:

0	1	2	3	4	5	$x \quad y$					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, chúng ta không có thông tin về chuỗi sau vị trí 10, vì vậy chúng ta 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 $z[9] = 7$, vì vậy khoảng $[x, y]$ 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, tất cả các giá trị mảng Z còn lại có thể được xác định bằng cách sử dụng thông tin đã được lưu trữ 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

Sử dụng mảng Z

Thường thì việc sử dụng băm chuỗi hay thuật toán Z là tùy thuộc vào sở thích. Không giống như băm, thuật toán Z luôn hoạt động và không có nguy cơ xảy ra xung đột. Mặt khác, thuật toán Z khó thực hiện hơn và một số bài toán chỉ có thể được giải quyết bằng cách sử dụng băm.

Ví dụ, hãy xem xét lại bài toán khớp mẫu, trong đó nhiệm vụ của chúng ta là tìm các lần xuất hiện của một mẫu p trong một chuỗi s . Chúng ta đã giải quyết bài toán này một cách hiệu quả bằng cách sử dụng băm chuỗi, nhưng thuật toán Z cung cấp một cách khác để giải quyết bài toán.

Một ý tưởng thông thường trong xử lý chuỗi là xây dựng một chuỗi bao gồm nhiều chuỗi được phân tách bằng các ký tự đặc biệt. Trong bài toán này, chúng ta có thể xây dựng một chuỗi $p\#s$, trong đó p và s được phân tách bằng một ký tự đặc biệt $\#$ không xuất hiện trong các chuỗi. Mảng Z của $p\#s$ cho chúng ta biết các vị trí mà p xuất hiện trong s , bởi 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, có nghĩa là mẫu ATT xuất hiện ở các vị trí tương ứng của HATTIVATTI .

Độ phức tạp thời gian của thuật toán kết quả là tuyến tính, bởi vì chỉ cần xây dựng mảng Z và duyệt qua các giá trị của nó.

Cài đặt

Đây là một cài đặt ngắn gọn 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 của nó. Một căn bậc hai có thể được coi là một "logarit của người nghèo": độ phức tạp $O(\sqrt{n})$ tốt hơn $O(n)$ nhưng tệ hơn $O(\log n)$. Trong mọi trường hợp, nhiều thuật toán căn bậc hai nhanh và có thể sử dụng được trong thực tế.

Ví dụ, hãy xem xét bài toán tạo một cấu trúc dữ liệu hỗ trợ hai thao tác trên một mảng: sửa đổi một phần tử tại một vị trí cho trước và tính tổng các phần tử trong một phạm vi cho trước. Trước đây chúng ta đã giải quyết bài toán này bằng cách sử dụng cây chỉ số nhị phân và cây phân đoạn, hỗ trợ cả hai thao tác trong thời gian $O(\log n)$. Tuy nhiên, bây giờ chúng ta sẽ giải quyết bài toán theo một cách khác bằng cách sử dụng cấu trúc căn bậc hai cho phép chúng ta sửa đổi các 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 chứa tổng các phần tử bên trong khối đó. Ví dụ, một mảng gồm 16 phần tử sẽ được chia thành các khối gồm 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 sửa đổi các phần tử mảng rất dễ dàng, bởi vì chỉ cần cập nhật tổng của một khối duy nhất sau mỗi lần sửa đổi, điều này có thể được thực hiện trong thời gian $O(1)$. Ví dụ, hình sau cho thấy cách giá trị của một phần tử và tổng của khối tương ứng thay đổi:

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 phạm vi, chúng ta chia phạm vi thành ba phần sao cho tổng bao gồm các giá trị của các phần tử đơn lẻ và tổng của các khối giữa chúng:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Vì số lượng phần tử đơn lẻ là $O(\sqrt{n})$ và số lượng khối cũng là $O(\sqrt{n})$, truy vấn tổng mất thời gian $O(\sqrt{n})$. Mục đích của kích thước khối \sqrt{n} là nó *cân bằng* hai thứ: mảng được chia thành \sqrt{n} khối, mỗi khối chứa \sqrt{n} phần tử.

Trong thực tế, không cần thiết phải sử dụng giá trị chính xác của \sqrt{n} làm tham số, và thay vào đó chúng ta có thể sử dụng các tham số k và n/k trong đó k khác \sqrt{n} . Tham số tối ưu phụ thuộc vào bài toán và đầu vào. Ví dụ, nếu một thuật toán thường xuyên duyệt qua các khối nhưng hiếm khi kiểm tra các phần tử đơn lẻ bên trong các khối, có thể là một ý tưởng tốt để chia mảng thành $k < \sqrt{n}$ khối, mỗi khối chứa $n/k > \sqrt{n}$ phần tử.

27.1 Kết hợp các thuật toán

Trong phần này, chúng ta thảo luận về hai thuật toán căn bậc hai dựa trên việc kết hợp hai thuật toán thành một thuật toán. Trong cả hai trường hợp, chúng ta có thể sử dụng một trong hai thuật toán mà không cần đến thuật toán còn lại và giải quyết bài toán trong thời gian $O(n^2)$. Tuy nhiên, bằng cách kết hợp các thuật toán, thời gian chạy chỉ là $O(n\sqrt{n})$.

Xử lý theo trường hợp

Giả sử chúng ta được cho một lưới hai chiều chứa n ô. Mỗi ô được gán một chữ cái, và nhiệm vụ của chúng ta là tìm hai ô có cùng chữ cái mà khoảng cách của chúng là nhỏ nhất, trong đó khoảng cách giữa các ô (x_1, y_1) và (x_2, y_2) là $|x_1 - x_2| + |y_1 - y_2|$. Ví dụ, hãy xem 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'.

Chúng ta có thể giải quyết bài toán bằng cách xem xét từng chữ cái riêng biệt. Sử dụng cách tiếp cận này, bài toán mới là tính khoảng cách nhỏ nhất giữa hai ô có một chữ cái *cố định* c . Chúng ta tập trung vào hai thuật toán cho việc này:

Thuật toán 1: Duyệt qua tất cả các cặp ô có chữ cái c , và tính khoảng cách nhỏ nhất giữa các ô đó. Việc này sẽ 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 đồng thời bắt đầu tại mỗi 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 để giải quyết bài toán là chọn một trong hai thuật toán và sử dụng nó cho tất cả các chữ cái. Nếu chúng ta sử dụng Thuật toán 1, thời gian chạy là $O(n^2)$, bởi vì tất cả các ô có thể chứa cùng một chữ cái, và trong trường hợp này $k = n$. Cũng như vậy, nếu chúng ta sử dụng Thuật toán 2, thời gian chạy là $O(n^2)$, bởi vì tất cả các ô có thể có các chữ cái khác nhau, và trong trường hợp này cần n lần tìm kiếm.

Tuy nhiên, chúng ta có thể *kết hợp* hai thuật toán và sử dụng các thuật toán khác nhau cho các chữ cái khác nhau tùy thuộc vào số lần mỗi chữ cái xuất hiện trong lưới. Giả sử một chữ cái c xuất hiện k lần. Nếu $k \leq \sqrt{n}$, chúng ta sử dụng Thuật toán 1, và nếu $k > \sqrt{n}$, chúng ta sử dụng Thuật toán 2. Hóa ra bằng cách làm như vậy, tổng thời gian chạy của thuật toán chỉ là $O(n\sqrt{n})$.

Đầu tiên, giả sử chúng ta sử dụng Thuật toán 1 cho một chữ cái c . Vì c xuất hiện nhiều nhất \sqrt{n} lần trong lưới, chúng ta so sánh mỗi ô có chữ cái c $O(\sqrt{n})$ lần với các ô khác. Do đó, thời gian được sử dụng để xử lý tất cả các ô đó là $O(n\sqrt{n})$. Sau đó, giả sử chúng ta sử dụng Thuật toán 2 cho một chữ cái c . Có nhiều nhất \sqrt{n} chữ cái như vậy, vì vậy việc xử lý các chữ cái đó cũng mất thời gian $O(n\sqrt{n})$.

Xử lý theo lô

Bài toán tiếp theo của chúng ta cũng liên quan đến một lưới hai chiều chứa n ô. Ban đầu, mỗi ô ngoại trừ một ô là màu trắng. Chúng ta thực hiện $n - 1$ thao tác, mỗi thao tác trước tiên tính khoảng cách nhỏ nhất từ một ô trắng cho trước đến một ô đen, và sau đó tô ô trắng đó thành màu đen.

Ví dụ, hãy xem xét thao tác sau:

■		*	
	■		■

Đầu tiên, chúng ta tính khoảng cách nhỏ nhất từ ô trắng được đánh dấu * đến một ô đen. Khoảng cách nhỏ nhất là 2, bởi vì chúng ta có thể di chuyển hai bước sang trái đến một ô đen. Sau đó, chúng ta tô ô trắng đó thành màu đen:

■		■	
	■		■

Hãy xem xét hai thuật toán sau:

Thuật toán 1: Sử dụng 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, chúng 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 và sau đó thêm một ô mới vào danh sách. Một thao tác mất thời gian $O(k)$ trong đó k là độ dài của danh sách.

Chúng ta kết hợp các thuật toán trên bằng cách chia các thao tác thành $O(\sqrt{n})$ lô, mỗi lô bao gồm $O(\sqrt{n})$ thao tác. Khi bắt đầu mỗi lô, chúng ta thực hiện Thuật toán 1. Sau đó, chúng ta sử dụng Thuật toán 2 để xử lý các thao tác trong lô. Chúng ta xóa danh sách của Thuật toán 2 giữa các lô. Tại mỗi thao tác, khoảng cách nhỏ nhất đến một ô đen hoặc là khoảng cách được tính bởi Thuật toán 1 hoặc là khoảng cách được tính bởi Thuật toán 2.

Thuật toán kết quả hoạt động trong thời gian $O(n\sqrt{n})$. Đầu tiên, Thuật toán 1 được thực hiện $O(\sqrt{n})$ lần, và mỗi lần tìm kiếm hoạt động trong thời gian $O(n)$. Thứ hai, khi sử dụng Thuật toán 2 trong một lô, danh sách chứa $O(\sqrt{n})$ ô (bởi vì chúng 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 dưới dạng tổng của các số nguyên dương, thì một tổng như vậy luôn chứa nhiều nhất $O(\sqrt{n})$ số *phân biệt*. Lý do cho điều này là để xây dựng một tổng chứa số lượng tối đa các số phân biệt, chúng ta nên chọn các số *nhỏ*. Nếu chúng ta chọn các số $1, 2, \dots, k$, tổng kết quả là

$$\frac{k(k+1)}{2}.$$

Do đó, số lượng tối đa các số phân biệt là $k = O(\sqrt{n})$. Tiếp theo chúng ta sẽ thảo luận về hai bài toán có thể được giải quyết một cách hiệu quả bằng cách sử dụng quan sát này.

Cái túi (Knapsack)

Giả sử chúng ta được cho một danh sách các trọng lượng số nguyên có tổng là n . Nhiệm vụ của chúng ta là tìm ra tất cả các tổng có thể được tạo thành bằng cách sử dụng một tập hợp con các trọng lượng. Ví dụ, nếu các trọng lượng là $\{1, 3, 3\}$, các tổng có thể là:

- 0 (tập rỗng)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Sử dụng cách tiếp cận cái túi tiêu chuẩn (xem Chương 7.4), bài toán có thể được giải quyết như sau: chúng ta định nghĩa một hàm $\text{possible}(x, k)$ có giá trị là 1 nếu tổng x có thể được tạo thành bằng cách sử dụng k trọng lượng đầu tiên, và 0 nếu không. Vì tổng các trọng lượng là n , có nhiều nhất n trọng lượng và tất cả các giá trị của hàm có thể được tính trong thời gian $O(n^2)$ bằng cách sử dụng quy hoạch động.

Tuy nhiên, chúng ta có thể làm cho thuật toán hiệu quả hơn bằng cách sử dụng thực tế là có nhiều nhất $O(\sqrt{n})$ trọng lượng *phân biệt*. Do đó, chúng ta có thể xử lý các trọng lượng theo nhóm bao gồm các trọng lượng tương tự. Chúng ta có thể xử lý mỗi nhóm trong thời gian $O(n)$, mang lại một thuật toán thời gian $O(n\sqrt{n})$.

Ý tưởng là sử dụng một mảng ghi lại các tổng trọng lượng có thể được tạo thành bằng cách sử dụng các nhóm đã được xử lý cho đến nay. Mảng chứa n phần tử: phần tử k là 1 nếu tổng k có thể được tạo thành và 0 nếu không. Để xử lý một nhóm trọng lượng, chúng ta quét mảng từ trái sang phải và ghi lại các tổng trọng lượng mới có thể được tạo thành bằng cách sử dụng nhóm này và các nhóm trước đó.

Xây dựng chuỗi

Cho một chuỗi s có độ dài n và một tập hợp các chuỗi D có tổng độ dài là m , hãy xem xét bài toán đếm số cách s có thể được tạo thành dưới dạng một chuỗi ghép của các chuỗi trong D . Ví dụ, nếu $s = \text{ABAB}$ và $D = \{A, B, AB\}$, có 4 cách:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Chúng ta có thể giải quyết bài toán bằng quy hoạch động: Gọi $\text{count}(k)$ là số cách để xây dựng tiền tố $s[0 \dots k]$ bằng cách sử dụng các chuỗi trong D . Bây giờ $\text{count}(n - 1)$ cho ra câu trả lời cho bài toán, và chúng ta có thể giải quyết bài toán trong thời gian $O(n^2)$ sử dụng cấu trúc trie.

Tuy nhiên, chúng ta có thể giải quyết bài toán hiệu quả hơn bằng cách sử dụng băm chuỗi và thực tế là có nhiều nhất $O(\sqrt{m})$ độ dài chuỗi phân biệt trong D . Đầu tiên, chúng ta xây dựng một tập hợp H chứa tất cả các giá trị băm của các chuỗi trong D . Sau đó, khi tính một giá trị của $\text{count}(k)$, chúng ta duyệt qua tất cả các giá trị của p sao cho có một chuỗi có độ dài p trong D , tính giá trị băm của $s[k - p + 1 \dots k]$ và kiểm tra xem nó có thuộc H hay không. Vì có nhiều nhất $O(\sqrt{m})$ độ dài chuỗi phân biệt, điều này dẫn đến một thuật toán có thời gian chạy là $O(n\sqrt{m})$.

27.3 Thuật toán của Mo (Mo's algorithm)

Thuật toán của Mo (Mo's algorithm)¹ có thể được sử dụng trong nhiều bài toán yêu cầu xử lý các truy vấn phạm vi trong một mảng *tĩnh*, tức là các giá trị mảng không thay đổi giữa các truy vấn. Trong mỗi truy vấn, chúng ta được cho một phạm vi $[a, b]$, và chúng ta nên tính một giá trị dựa trên các phần tử mảng giữa các 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 của Mo xử lý các truy vấn theo một thứ tự đặc biệt đảm bảo thuật toán hoạt động hiệu quả.

Thuật toán của Mo duy trì một *phạm vi hoạt động* của mảng, và câu trả lời cho một truy vấn liên quan đến phạm vi hoạt động được biết tại mọi thời điểm. Thuật toán xử lý các truy vấn từng cái một, và luôn di chuyển các điểm cuối của phạm vi hoạt động bằng cách chèn và xóa các phần tử. Độ phức tạp thời gian của thuật toán là $O(n\sqrt{n}f(n))$ trong đó mảng chứa n phần tử, có n truy vấn và mỗi lần chèn và xóa một phần tử mất thời gian $O(f(n))$.

Bí quyết trong thuật toán của Mo là thứ tự mà các truy vấn được xử lý: 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

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ hoặc
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ và $b_1 < b_2$.

Do đó, tất cả các truy vấn có điểm cuối bên trái trong một khối nhất định được xử lý lần lượt được sắp xếp theo điểm cuối bên phải của chúng. Sử dụng thứ tự này, thuật toán chỉ thực hiện $O(n\sqrt{n})$ thao tác, bởi vì điểm cuối bên trái di chuyển $O(n)$ lần $O(\sqrt{n})$ bước, và điểm cuối bên phải di chuyển $O(\sqrt{n})$ lần $O(n)$ bước. Do đó, cả hai điểm cuối di chuyển tổng cộng $O(n\sqrt{n})$ bước trong suốt thuật toán.

Ví dụ

Ví dụ, hãy xem xét một bài toán trong đó chúng ta được cho một tập hợp các truy vấn, mỗi truy vấn tương ứng với một phạm vi trong một mảng, và nhiệm vụ của chúng ta là tính cho mỗi truy vấn số lượng các phần tử *phân biệt* trong phạm vi đó.

Trong thuật toán của Mo, các truy vấn luôn được sắp xếp theo cùng một cách, nhưng việc duy trì câu trả lời cho truy vấn phụ thuộc vào bài toán. Trong bài toán này, chúng ta có thể duy trì một mảng `count` trong đó `count[x]` chỉ ra số lần một phần tử x xuất hiện trong phạm vi hoạt động.

Khi chúng ta di chuyển từ một truy vấn này sang một truy vấn khác, phạm vi hoạt động thay đổi. Ví dụ, nếu phạm vi hiện tại là

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

và phạm vi tiếp theo là

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

sẽ có ba bước: điểm cuối bên trái di chuyển một bước sang phải, và điểm cuối bên phải di chuyển hai bước sang phải.

Sau mỗi bước, mảng `count` cần được cập nhật. Sau khi th

¹Theo [12], thuật toán này được đặt theo tên của Mo Tao, một lập trình viên thi đấu người Trung Quốc, nhưng kỹ thuật này đã xuất hiện trước đó trong tài liệu [44].

êm một phần tử x , chúng ta tăng giá trị của $\text{count}[x]$ lên 1, và nếu $\text{count}[x] = 1$ sau đó, chúng ta cũng tăng câu trả lời cho truy vấn lên 1. Tương tự, sau khi xóa một phần tử x , chúng ta giảm giá trị của $\text{count}[x]$ đi 1, và nếu $\text{count}[x] = 0$ sau đó, chúng ta cũng giảm câu trả lời cho truy vấn đi 1.

Trong bài toán này, thời gian cần thiết để thực hiện mỗi bước là $O(1)$, vì vậy tổng độ phức tạp thời gian của thuật toán là $O(n\sqrt{n})$.

Chương 28

Cây phân đoạn nâng cao

Cây phân đoạn là một cấu trúc dữ liệu đa năng có thể được sử dụng để giải quyết một số lượng lớn các bài toán thuật toán. Tuy nhiên, có nhiều chủ đề liên quan đến cây phân đoạn mà chúng ta chưa đề cập đến. Bây giờ là lúc để thảo luận về một số biến thể nâng cao hơn của cây phân đoạn.

Cho đến nay, chúng ta đã triển khai các thao tác của một cây phân đoạn bằng cách đi *từ dưới lên trên* trong cây. Ví dụ, chúng ta đã tính tổng phạm vi 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 phải triển khai các thao tác theo một cách khác, *từ trên xuống dưới*. Sử dụng 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);
}
```

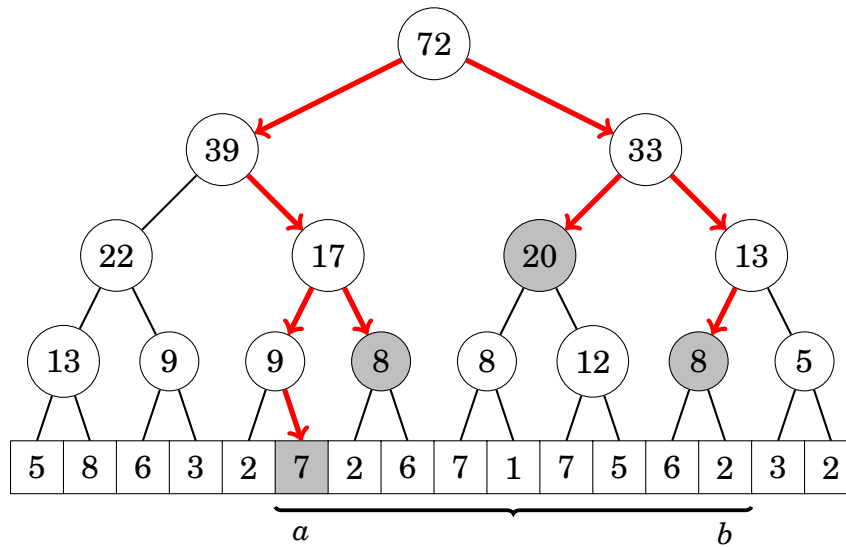
Bây giờ chúng ta có thể tính bất kỳ giá trị nào của $\text{sum}_q(a, b)$ (tổng các giá trị mảng trong phạm vi $[a, b]$) như sau:

```
int s = sum(a, b, 1, 0, n-1);
```

Tham số k chỉ ra vị trí hiện tại trong `tree`. Ban đầu k bằng 1, bởi vì chúng ta bắt đầu tại gốc của cây. Phạm vi $[x, y]$ 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 $[x, y]$ hoàn toàn nằm trong $[a, b]$, tổng có thể được tìm thấy trong `tree`. Nếu $[x, y]$ một phần nằm trong $[a, b]$, việc tìm kiếm tiếp tục một cách đệ quy đến 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 diễn ra khi tính giá trị của $\text{sum}_q(a, b)$. Các nút màu xám chỉ ra các nút nơi đệ quy dừng lại và tổng có thể được tìm thấy trong tree.



Cũng trong cách triển khai này, các thao tác mất thời gian $O(\log n)$, bởi vì tổng số nút được duyệt là $O(\log n)$.

28.1 Truyền lười (Lazy propagation)

Sử dụng **truyền lười (lazy propagation)**, chúng ta có thể xây dựng một cây phân đoạn hỗ trợ cả cập nhật phạm vi và truy vấn phạm vi trong thời gian $O(\log n)$. Ý tưởng là thực hiện các cập nhật và truy vấn từ trên xuống dưới và thực hiện các cập nhật một cách *lười biếng* để chúng được truyền xuống cây chỉ khi cần thiết.

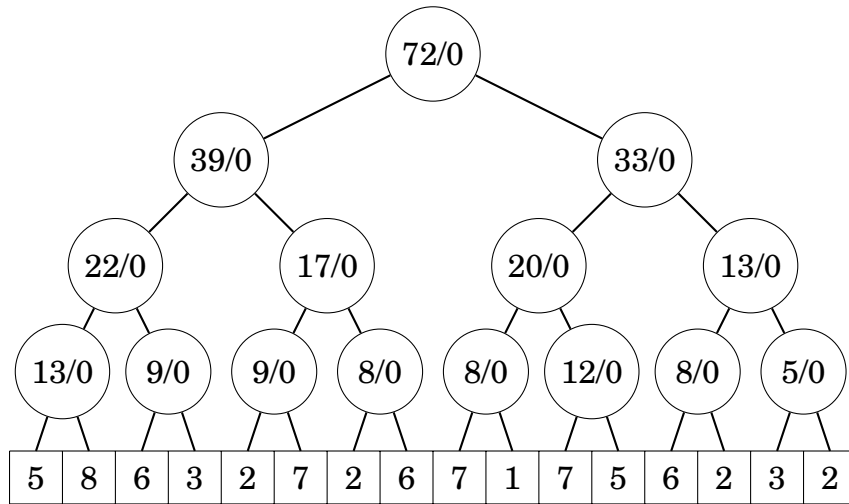
Trong một cây phân đoạn lười biếng, các nút chứa hai loại thông tin. Giống như trong một cây phân đoạn thông thường, mỗi nút chứa tổng hoặc một giá trị khác liên quan đến mảng con tương ứng. Ngoài ra, nút có thể chứa thông tin liên quan đến các cập nhật lười biếng, chưa được truyền đến các con của nó.

Có hai loại cập nhật phạm vi: mỗi giá trị mảng trong phạm vi hoặc là được *tăng* lên một giá trị nào đó hoặc được *gán* một giá trị nào đó. Cả hai thao tác có thể được triển khai bằng cách sử dụng các ý tưởng tương tự, và thậm chí có thể xây dựng một cây hỗ trợ cả hai thao tác cùng một lúc.

Cây phân đoạn lười biếng

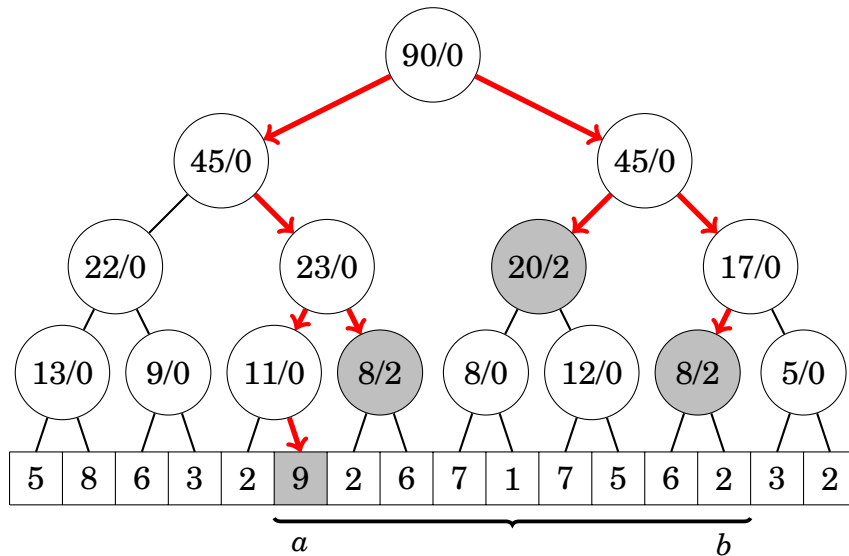
Hãy xem xét một ví dụ trong đó mục tiêu của chúng ta là xây dựng một cây phân đoạn hỗ trợ hai thao tác: tăng mỗi giá trị trong $[a, b]$ lên một hằng số và tính tổng các giá trị trong $[a, b]$.

Chúng ta sẽ xây dựng một cây trong đó mỗi nút có hai giá trị s/z : s biểu thị tổng các giá trị trong phạm vi, và z biểu thị giá trị của một cập nhật lười biếng, có nghĩa là tất cả các giá trị trong phạm vi nên được tăng lên z . Trong cây sau, $z = 0$ ở tất cả các nút, vì vậy không có cập nhật lười biếng nào đang diễn ra.



Khi các phần tử trong $[a, b]$ được tăng lên u , chúng ta đi từ gốc đến các lá và sửa đổi các nút của cây như sau: Nếu phạm vi $[x, y]$ của một nút hoàn toàn nằm trong $[a, b]$, chúng ta tăng giá trị z của nút lên u và dừng lại. Nếu $[x, y]$ chỉ một phần thuộc về $[a, b]$, chúng ta tăng giá trị s của nút lên hu , trong đó h là kích thước của giao của $[a, b]$ và $[x, y]$, và tiếp tục đi một cách đệ quy trong cây.

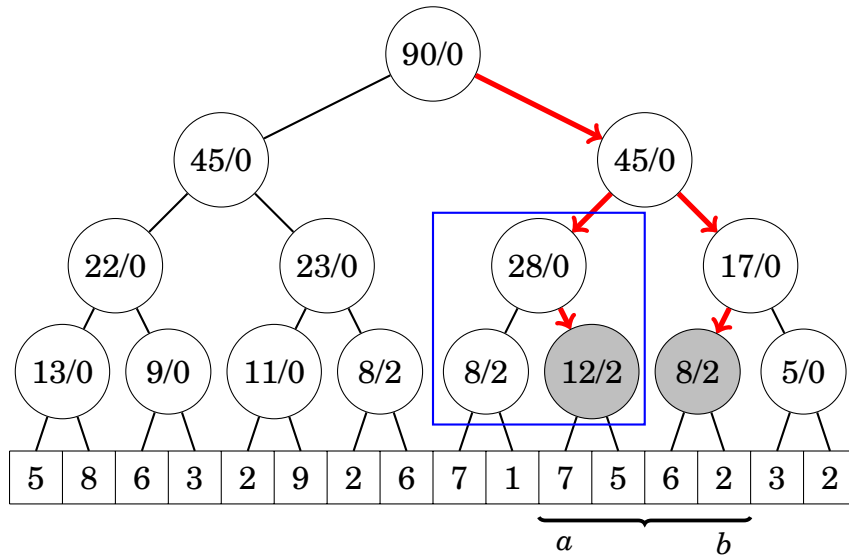
Ví dụ, hình sau cho thấy cây sau khi tăng các phần tử trong $[a, b]$ lên 2:



Chúng ta cũng tính tổng các phần tử trong một phạm vi $[a, b]$ bằng cách đi trong cây từ trên xuống dưới. Nếu phạm vi $[x, y]$ của một nút hoàn toàn thuộc về $[a, b]$, chúng ta cộng giá trị s của nút đó vào tổng. Ngược lại, chúng ta tiếp tục tìm kiếm một cách đệ quy xuống dưới trong cây.

Cả trong cập nhật và truy vấn, giá trị của một cập nhật lười biếng luôn được truyền đến các con của nút trước khi xử lý nút. Ý tưởng là các cập nhật sẽ được truyền xuống dưới chỉ khi cần thiết, điều này đảm bảo rằng các thao tác luôn hiệu quả.

Hình sau cho thấy cây thay đổi như thế nào khi chúng ta tính giá trị của $\text{sum}_a(a, b)$. Hình chữ nhật cho thấy các nút có giá trị thay đổi, bởi vì một cập nhật lười biếng được truyền xuống dưới.



Lưu ý rằng đôi khi cần phải kết hợp các cập nhật lười biếng. Điều này xảy ra khi một nút đã có một cập nhật lười biếng được gán một cập nhật lười biếng khác. Khi tính tổng, việc kết hợp các cập nhật lười biếng rất dễ dàng, bởi vì sự kết hợp của các cập nhật z_1 và z_2 tương ứng với một cập nhật $z_1 + z_2$.

Cập nhật đa thức

Các cập nhật lười biếng có thể được tổng quát hóa để có thể cập nhật các phạm vi bằng các 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 một giá trị tại vị trí i trong $[a, b]$ là $p(i - a)$. Ví dụ, cộng đa thức $p(u) = u + 1$ vào $[a, b]$ có nghĩa là giá trị tại vị trí a tăng lên 1, giá trị tại vị trí $a + 1$ tăng lên 2, và cứ thế.

Để hỗ trợ các 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 của các phần tử trong phạm vi, và các giá trị z_0, z_1, \dots, z_k là các hệ số của một đa thức tương ứng với một cập nhật lười biếng.

Bây giờ, tổng các giá trị trong một phạm vi $[x, y]$ 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 cách sử dụng các công thức tổng. Ví dụ, số hạng z_0 tương ứng với tổng $(y - x + 1)z_0$, và số hạng $z_1 u$ tương ứng với tổng

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

Khi truyền một cập nhật trong cây, các chỉ số của $p(u)$ thay đổi, bởi vì trong mỗi phạm vi $[x, y]$, các giá trị được tính cho $u = 0, 1, \dots, y - x$. Tuy nhiên, đây không phải là vấn đề, bởi vì $p'(u) = p(u + h)$ là một đa thức có bậc bằng $p(u)$. 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

Một cây phân đoạn thông thường là tĩnh, có nghĩa là mỗi nút có một vị trí cố định trong mảng và cây yêu cầu một lượng bộ nhớ cố định. Trong một **cây phân đoạn động (dynamic segment tree)**, bộ nhớ chỉ được cấp phát cho các nút thực sự được truy cập trong quá trình thuật toán, điều này có thể tiết kiệm một lượng lớn bộ nhớ.

Các nút của một cây động có thể được biểu diễn dưới dạng các cấu trúc:

```
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 phạm vi tương ứng, và left và right trỏ đến cây con trái và phải.

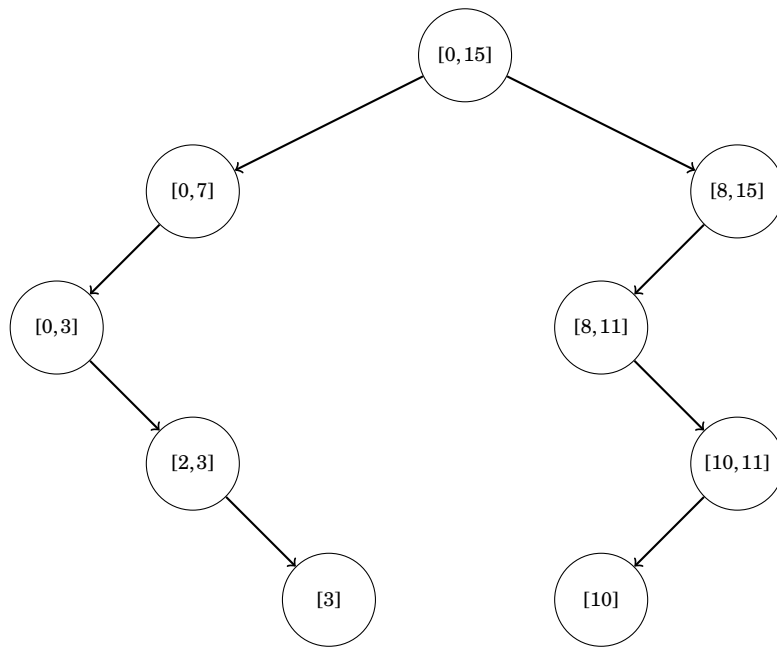
Sau đó, các nút có thể được tạo như sau:

```
// tạo nút mới  
node *x = new node(0, 0, 15);  
// thay đổi giá trị  
x->value = 5;
```

Cây phân đoạn thưa

Một cây phân đoạn động hữu ích khi mảng cơ sở là *thưa*, tức là, phạm vi $[0, n-1]$ các chỉ số được phép là lớn, nhưng hầu hết các giá trị mảng là không. Trong khi một cây phân đoạn thông thường sử dụng $O(n)$ bộ nhớ, một cây phân đoạn động chỉ sử dụng $O(k \log n)$ bộ nhớ, trong đó k là số thao tác được thực hiện.

Một **cây phân đoạn thưa (sparse segment tree)** ban đầu chỉ có một nút $[0, n-1]$ có giá trị là không, có nghĩa là mọi giá trị mảng đều là không. Sau các cập nhật, các nút mới được thêm động vào cây. Ví dụ, nếu $n = 16$ và các phần tử ở các 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, vì vậy 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.

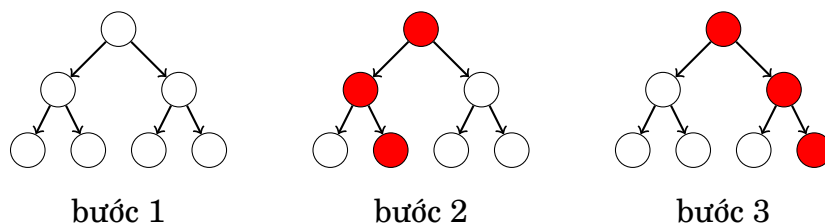
Lưu ý rằng nếu chúng ta biết tất cả các phần tử sẽ được cập nhật khi bắt đầu thuật toán, một cây phân đoạn động là không cần thiết, bởi vì chúng ta có thể sử dụng một cây phân đoạn thông thường với nén chỉ số (Chương 9.4). Tuy nhiên, điều này là không thể khi các chỉ số được tạo ra trong quá trình thuật toán.

Cây phân đoạn bền bỉ

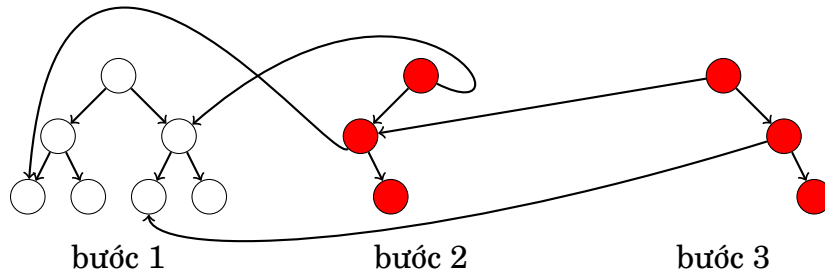
Sử dụng một cách triển khai động, cũng có thể tạo một **cây phân đoạn bền bỉ (persistent segment tree)** lưu trữ *lịch sử sửa đổi* của cây. Trong một cách triển khai như vậy, chúng ta có thể truy cập hiệu quả tất cả các phiên bản của cây đã tồn tại trong quá trình thuật toán.

Khi lịch sử sửa đổi có sẵn, chúng ta có thể thực hiện các truy vấn trong bất kỳ cây nào trước đó giống như trong một cây phân đoạn thông thường, bởi vì cấu trúc đầy đủ của mỗi cây được lưu trữ. Chúng ta cũng có thể tạo các cây mới dựa trên các cây trước đó và sửa đổi chúng một cách độc lập.

Hãy xem xét chuỗi cập nhật sau, trong đó các nút màu đỏ thay đổi và các nút khác vẫn giữ nguyên:



Sau mỗi lần cập nhật, hầu hết các nút của cây vẫn giữ nguyên, vì vậy một cách hiệu quả để lưu trữ lịch sử sửa đổi là biểu diễn mỗi cây trong lịch sử dưới dạng một sự kết hợp của các nút mới và các cây con của các cây trước đó. Trong ví dụ này, lịch sử sửa đổi có thể được lưu trữ như sau:



Cấu trúc của mỗi cây trước đó có thể được tái tạo bằng cách đi theo các con trỏ bắt đầu từ nút gốc tương ứng. Vì mỗi thao tác chỉ thêm $O(\log n)$ nút mới vào cây, có thể lưu trữ toàn bộ lịch sử sửa đổi của cây.

28.3 Cấu trúc dữ liệu

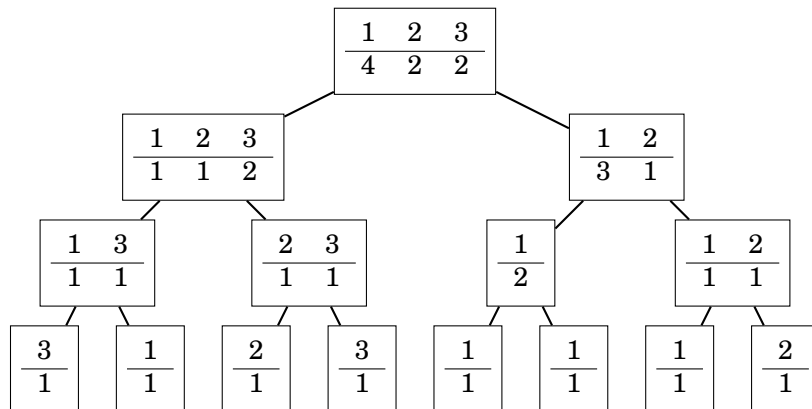
Thay vì các giá trị đơn lẻ, các nút trong một cây phân đoạn cũng có thể chứa *các cấu trúc dữ liệu* duy trì thông tin về các phạm vi tương ứng. Trong một cây như vậy, các thao tác mất thời gian $O(f(n)\log n)$, trong đó $f(n)$ là thời gian cần thiết để xử lý một nút duy nhất trong một thao tác.

Ví dụ, hãy xem xét một cây phân đoạn hỗ trợ các truy vấn có dạng "một phần tử x xuất hiện bao nhiêu lần trong phạm vi $[a, b]$?" Ví dụ, phần tử 1 xuất hiện ba lần trong phạm vi sau:

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

Để hỗ trợ các truy vấn như vậy, chúng ta xây dựng một cây phân đoạn trong đó mỗi nút được gán một cấu trúc dữ liệu có thể được hỏi một phần tử x bất kỳ xuất hiện bao nhiêu lần trong phạm vi tương ứng. Sử dụng cây này, câu trả lời cho một truy vấn có thể được tính bằng cách kết hợp các kết quả từ các nút thuộc phạm vi.

Ví dụ, cây phân đoạn sau tương ứng với mảng trên:



Chúng ta có thể xây dựng cây sao cho mỗi nút chứa một cấu trúc map. Trong trường hợp này, thời gian cần thiết để xử lý mỗi nút là $O(\log n)$, vì vậy tổng độ phức tạp thời gian của một truy vấn là $O(\log^2 n)$. Cây sử dụng $O(n \log n)$ bộ nhớ, bởi vì có $O(\log n)$ cấp và mỗi cấp chứa $O(n)$ phần tử.

28.4 Hai chiều

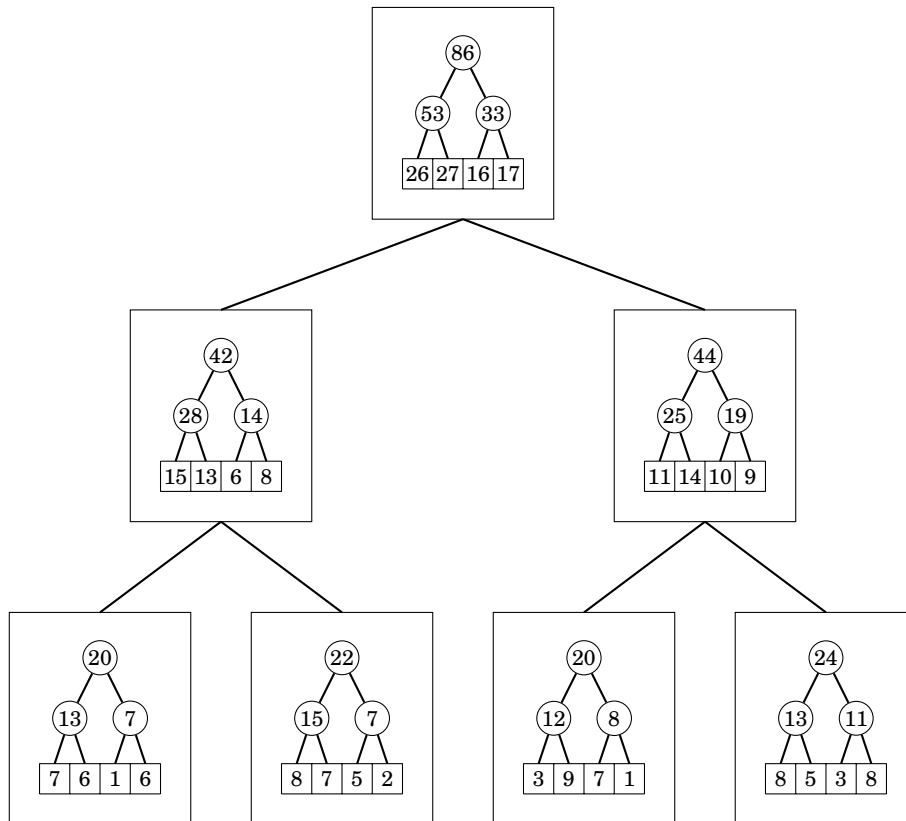
Một **cây phân đoạn hai chiều** (two-dimensional segment tree) hỗ trợ các truy vấn liên quan đến các mảng con hình chữ nhật của một mảng hai chiều. Một cây như vậy có

thể được triển khai 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 cũng có thể được tính từ cây phân đoạn sau:



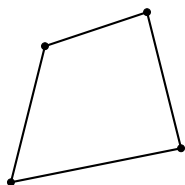
Các thao tác của một cây phân đoạn hai chiều mất thời gian $O(\log^2 n)$, bởi vì cây lớn và mỗi cây nhỏ bao gồm $O(\log n)$ cấp. Cây yêu cầu $O(n^2)$ bộ nhớ, bởi vì mỗi cây nhỏ chứa $O(n)$ giá trị.

Chương 29

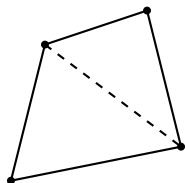
Hình học

Trong các bài toán hình học, thường khá thách thức để tìm ra cách tiếp cận bài toán sao cho có thể cài đặt lời giải một cách thuận tiện và số lượng trường hợp đặc biệt là ít.

Ví dụ như xét một bài toán trong đó ta được cho các đỉnh của một tứ giác (một đa giác có bốn đỉnh), và nhiệm vụ của ta là tính diện tích của nó. Ví dụ, một đầu vào có thể của bài toán như sau:



Một cách tiếp cận bài toán là chia tứ giác thành hai tam giác bằng một đường thẳng nối hai đỉnh đối diện:

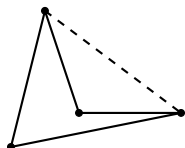


Sau đó, chỉ cần tính tổng diện tích của hai tam giác. Diện tích của một tam giác có thể được tính, ví dụ, bằng **công thức Heron** (Heron's formula)

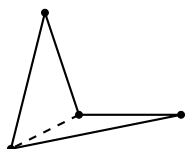
$$\sqrt{s(s-a)(s-b)(s-c)},$$

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 để giải bài toán, nhưng có một cạm bẫy: làm thế nào để chia tứ giác thành các tam giác? Hóa ra đôi khi ta không thể chỉ chọn hai đỉnh đối diện bất kỳ. Ví dụ, trong tình huống sau, đường chia nằm *bên ngoài* tứ giác:



Tuy nhiên, một cách vẽ đường khác lại được:



Với con người thì rõ ràng đường nào là lựa chọn đúng, nhưng tình huống này lại khó với máy tính.

Tuy nhiên, hóa ra ta có thể giải quyết bài toán bằng cách dùng một phương pháp khác thuận tiện hơn cho lập trình viên. Cụ thể, có một công thức tổng quát

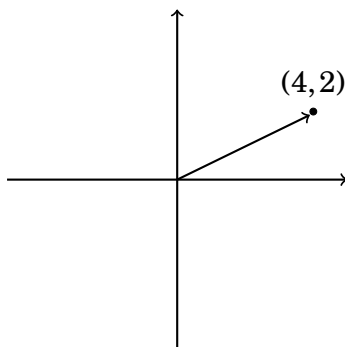
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

tính được 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 nào, và ta thậm chí có thể tổng quát hóa công thức này cho *mọi* đa giác.

29.1 Số phức

Số phức (complex number) là số có dạng $x + yi$, trong đó $i = \sqrt{-1}$ là **đơn vị ảo** (imaginary unit). Một cách diễn giải hình học của 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 trong C++ rất hữu ích khi giải các bài toán hình học. Sử dụng 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 dữ liệu của tọa độ và P là kiểu dữ liệu của điểm hoặc vectơ. Thêm vào đó, đoạn mã định nghĩa các macro X và Y có thể được sử dụng 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 ra 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 $u = (2, 2)$, và sau đó 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 dữ liệu tọa độ phù hợp thường là long long (số nguyên) hoặc long double (số thực). Sử dụng số nguyên bất cứ khi nào có thể là một ý tưởng tốt, vì các phép tính với số nguyên là chính xác. Nếu cần dùng số thực, lỗi làm tròn nên được tính đến khi so sánh các số. Một cách an toàn để kiểm tra xem các số thực a và b có bằng nhau không là so sánh chúng bằng cách dùng $|a - b| < \epsilon$, trong đó ϵ là một số nhỏ (ví dụ, $\epsilon = 10^{-9}$).

Các hàm

Trong các ví dụ sau, kiểu dữ liệu tọa độ là long double.

Hàm $\text{abs}(v)$ tính độ dài $|v|$ của một vectơ $v = (x, y)$ sử dụng công thức $\sqrt{x^2 + y^2}$. Hàm này cũng có thể được sử dụng để tính khoảng cách giữa các điểm (x_1, y_1) và (x_2, y_2) , bởi vì khoảng cách đó bằng độ dài của vectơ $(x_2 - x_1, y_2 - y_1)$.

Đoạn mã sau tính khoảng cách giữa điểm $(4, 2)$ và $(3, -1)$:

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.16228
```

Hàm $\text{arg}(v)$ 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 chỉ về phía phải là 0, và góc giảm theo chiều kim đồng hồ và tăng ngược chiều kim đồng hồ.

Hàm $\text{polar}(s, a)$ tạo ra một vectơ có độ dài là s và chỉ về góc a . Một vectơ có thể được xoay một góc a bằng cách nhân nó với một vectơ có độ dài 1 và góc a .

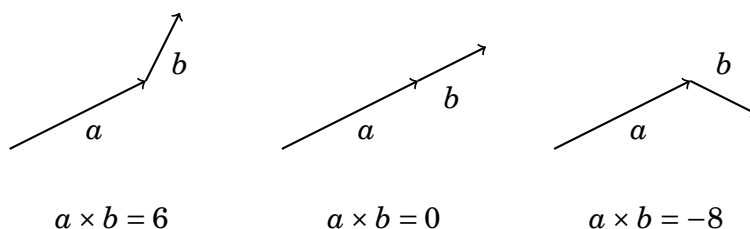
Đoạn mã sau tính góc của vectơ $(4, 2)$, xoay nó $1/2$ radian ngược chiều kim đồng hồ, và sau đó tính lại góc:

```
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 ta 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 sử dụng lớp complex:

```
P a = {4,2};
P b = {1,2};
```

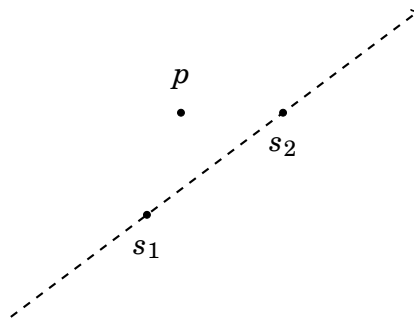
$$C_p = (\text{conj}(a) * b) \cdot Y; // 6$$

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 $x_1 y_2 - x_2 y_1$.

Vị trí điểm

Tích có hướng có thể được sử dụng để kiểm tra xem một điểm nằm ở bên trái hay bên phải của một đường thẳng. Giả sử đường thẳng đi qua các điểm s_1 và s_2 , ta đang nhìn từ s_1 đến s_2 và điểm cần kiểm tra là p .

Ví dụ, trong hình sau, p nằm ở bên trái của đường thẳng:

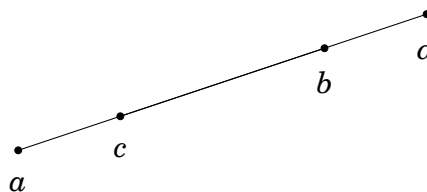


Tích có hướng $(p - s_1) \times (p - s_2)$ cho ta biết vị trí của điểm p . Nếu tích có hướng dương, p nằm ở bên trái, và nếu tích có hướng âm, p nằm ở bên phải. Cuối cùng, nếu tích có hướng bằng không, các điểm s_1 , s_2 và p nằm trên cùng một đường thẳng.

Giao điểm của đoạn thẳng

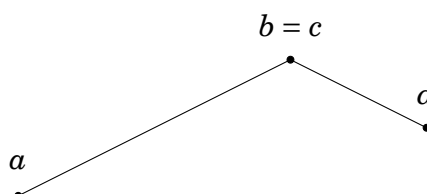
Tiếp theo ta xét bài toán kiểm tra liệu hai đoạn thẳng ab và cd có giao nhau không. Các trường hợp có thể xảy ra là:

Trường hợp 1: Các đoạn thẳng nằm trên cùng một đường thẳng và chúng chồng lên nhau. Trong trường hợp này, có vô số giao điểm. Ví dụ, trong hình sau, tất cả các điểm giữa c và b là giao điểm:



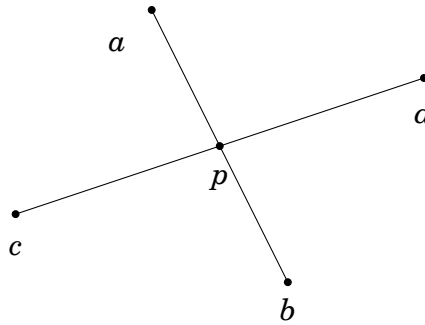
Trong trường hợp này, ta có thể sử dụng tích có hướng để kiểm tra xem tất cả các điểm có nằm trên cùng một đường thẳng không. Sau đó, ta có thể sắp xếp các điểm và kiểm tra liệu các đoạn thẳng có chồng lên nhau không.

Trường hợp 2: Các đoạn thẳng có một đỉnh chung là điểm giao duy nhất. Ví dụ, trong hình sau điểm giao là $b = c$:



Trường hợp này dễ kiểm tra, bởi vì chỉ có bốn khả năng cho điểm giao: $a = c$, $a = d$, $b = c$ và $b = d$.

Trường hợp 3: Có chính xác một giao điểm không phải là đỉnh của bất kỳ đoạn thẳng nào. Trong hình sau, điểm p là giao điểm:



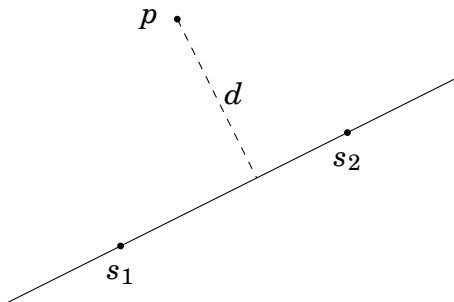
Trong trường hợp này, các đoạn thẳng giao nhau khi và chỉ khi cả hai điểm c và d đều nằm ở hai bên khác nhau của đường thẳng qua a và b , và các điểm a và b nằm ở hai bên khác nhau của đường thẳng qua c và d . Ta có thể sử dụng tích có hướng để kiểm tra điều này.

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. Sử dụng sự thật này, ta có thể suy ra một công thức để tính khoảng cách ngắn nhất từ một điểm đến 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 :

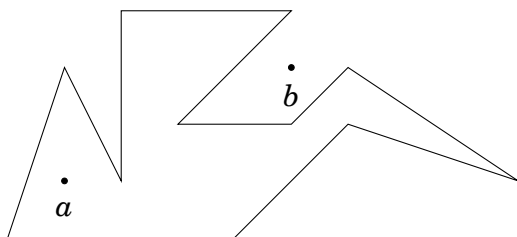


Diện tích của tam giác có các đỉnh là s_1 , s_2 và p có thể được tính theo hai cách: nó vừa bằng $\frac{1}{2}|s_2 - s_1|d$ và $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Do đó, khoảng cách ngắn nhất là

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

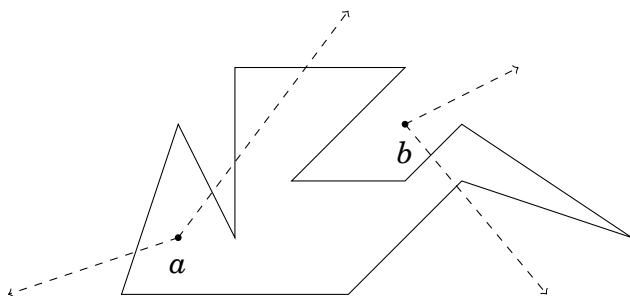
Điểm trong đa giác

Bây giờ ta hãy xét bài toán kiểm tra xem 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 quyết bài toán là phóng một *tia* từ điểm đó theo một hướng bất kỳ và tính số lần nó chạm vào 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 phóng các tia sau:



Các tia từ a chạm biên 1 và 3 lần biên của đa giác, nên a nằm bên trong đa giác. Tương tự, các tia từ b chạm biên 0 và 2 lần, nên b nằm bên ngoài đa giác.

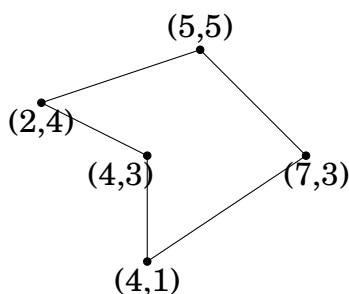
29.3 Diện tích đa giác

Một công thức tổng quát để tính diện tích của một đa giác, đôi khi được gọi là **công thức dây giày** (shoelace formula), 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 $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ theo thứ tự sao cho p_i và p_{i+1} là các đỉnh liên tiếp trên biên của đa giác, và đỉnh đầu tiên và cuối cùng là cùng một đỉnh, tức là $p_1 = p_n$.

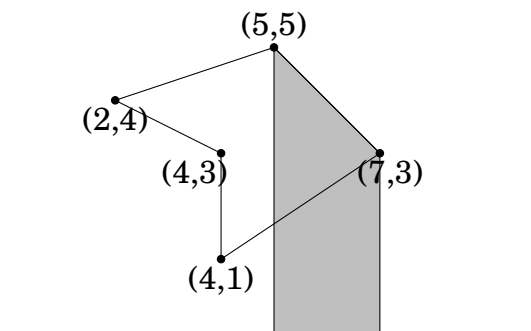
Ví dụ, diện tích của đa giác



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 đi qua các hình thang mà một cạnh là cạnh của đa giác, và cạnh kia nằm trên đường nằm 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 dương, và nếu $x_{i+1} < x_i$, diện tích âm.

Diện tích của đa giác là tổng diện tích của tất cả các hình thang như vậy, cho ta 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 giá trị tuyệt đối của tổng được lấy, bởi vì giá trị của tổng có thể dương hoặc âm, phụ thuộc vào việc ta đi theo chiều kim đồng hồ hay ngược chiều kim đồng hồ dọc theo biên của đa giác.

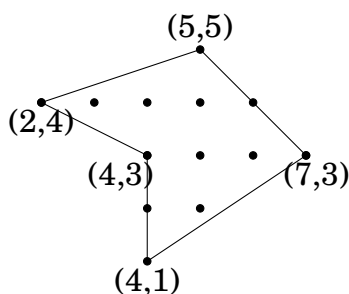
Định lý Pick

Định lý Pick (Pick's theorem) cung cấp một cách khác để tính diện tích của một đa giác nếu tất cả các 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** (distance function) định nghĩa khoảng cách giữa hai điểm. Hàm khoảng cách thông dụng nhất là **khoảng cách Euclid** (Euclidean distance) trong đó

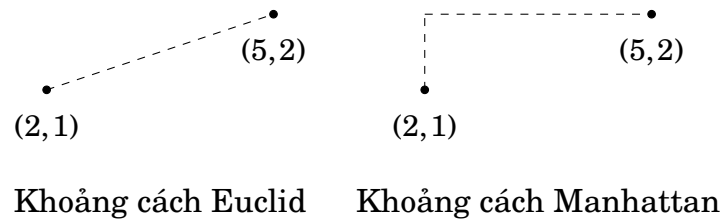
khoảng cách giữa các điểm (x_1, y_1) và (x_2, y_2) là

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

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:



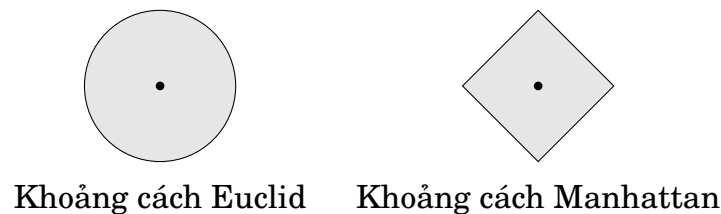
Khoảng cách Euclid giữa hai điểm là

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

và khoảng cách Manhattan là

$$|5 - 2| + |2 - 1| = 4.$$

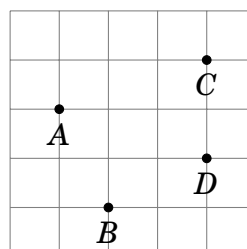
Hình sau thể hiện các vùng có khoảng cách không lớn hơn 1 từ điểm trung tâm, sử dụng khoảng cách Euclid và khoảng cách Manhattan:



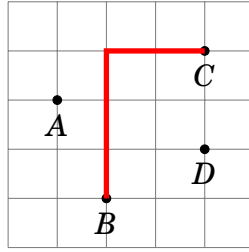
Xoay tọa độ

Một số bài toán dễ giải hơn nếu sử dụng khoảng cách Manhattan thay vì khoảng cách Euclid. Ví dụ, xét bài toán mà trong đó ta được cho n điểm trên mặt phẳng hai chiều và nhiệm vụ của ta là tính khoảng cách Manhattan lớn nhất giữa hai điểm bất kỳ.

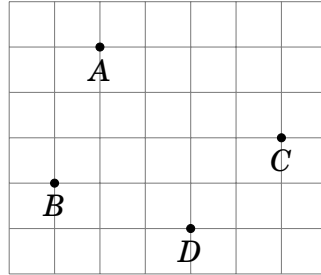
Ví dụ, xét tập điểm sau:



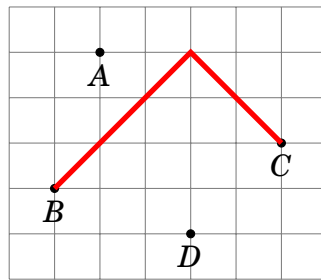
Khoảng cách Manhattan lớn nhất là 5 giữa các điểm B và C :



Một kỹ thuật hữu ích liên quan đến khoảng cách Manhattan là xoay tất cả các 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 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 để 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 $p_2 = (3, 3)$, các tọa độ sau khi xoay là $p'_1 = (1, -1)$ và $p'_2 = (6, 0)$ và khoảng cách Manhattan là

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

Các tọa độ sau khi xoay cung cấp một cách đơn giản để làm việc với khoảng cách Manhattan, bởi vì ta có thể xét tọa độ x và y một cách riêng biệt. Để tối đa hóa khoảng cách Manhattan giữa hai điểm, ta nên tìm hai điểm mà tọa độ sau khi xoay của chúng làm tối đa giá trị

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

Điều này dễ dàng, bởi vì hoặc độ chênh lệch theo chiều ngang hoặc theo chiều dọc của các tọa độ sau khi xoay phải là lớn nhất.

Chương 30

Thuật toán đường quét

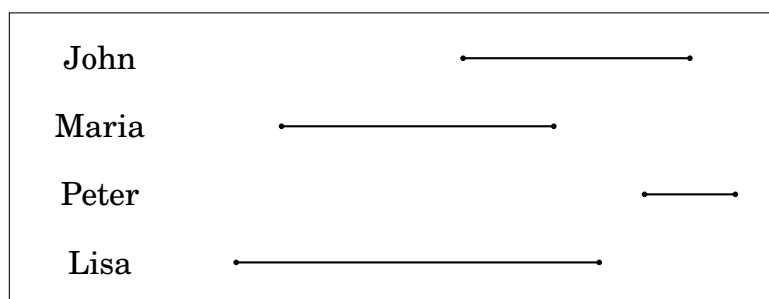
Nhiều bài toán hình học có thể được giải bằng cách sử dụng thuật toán **đường quét** (sweep line). Ý tưởng của những thuật toán này là biểu diễn một trường hợp của bài toán dưới dạng một tập các sự kiện tương ứng với các điểm trên mặt phẳng. Các sự kiện được xử lý theo thứ tự tăng dần của tọa độ x hoặc y của chúng.

Ví dụ, xét bài toán sau: Có một công ty có n nhân viên, và ta biết với mỗi nhân viên thời điểm đến và thời điểm rời khỏi văn phòng trong một ngày nhất định. Nhiệm vụ của ta là tính số lượng tối đa nhân viên có mặt cùng lúc trong văn phòng.

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 đi qua chúng và theo dõi số người trong văn phòng. Ví dụ, bảng

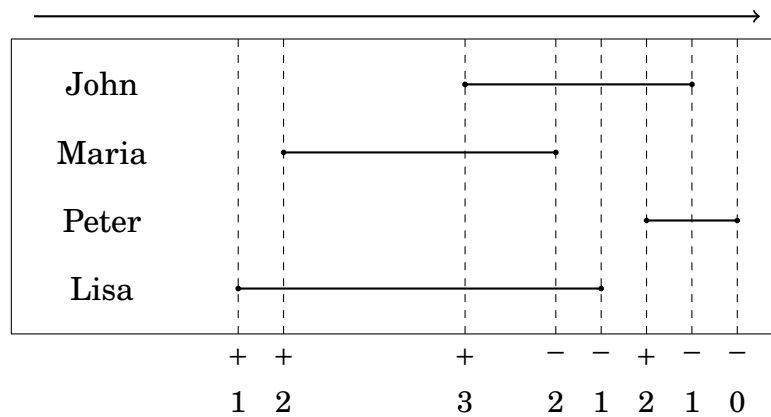
người	thời điểm đến	thời điểm đi
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

tương ứng với các sự kiện sau:



Ta đi qua các sự kiện từ trái sang phải và duy trì một bộ đếm. Mỗi khi một người đến, ta tăng giá trị của bộ đếm lên một, và khi một người rời đi, ta giảm giá trị của bộ đếm đi một. Câu trả lời của bài toán là giá trị lớn nhất của bộ đếm trong quá trình thực hiện thuật toán.

Trong ví dụ này, các sự kiện được xử lý như sau:

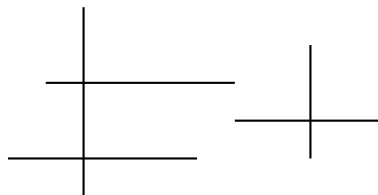


Các ký hiệu + và - chỉ ra liệu giá trị của bộ đếm tăng hay giảm, và giá trị của bộ đếm được hiển thị bên dưới. Giá trị lớn nhất của bộ đếm là 3 giữa thời điểm John đến và Maria rời đi.

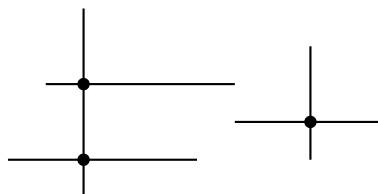
Thời gian chạy của thuật toán là $O(n \log n)$, bởi vì việc sắp xếp các sự kiện mất thời gian $O(n \log n)$ và phần còn lại của thuật toán mất thời gian $O(n)$.

30.1 Giao điểm

Cho một tập n đoạn thẳng, mỗi đoạn hoặc là nằm ngang hoặc thẳng đứng, xét bài toán đếm tổng số giao điểm. Ví dụ, khi các đoạn thẳng là



có ba giao điểm:

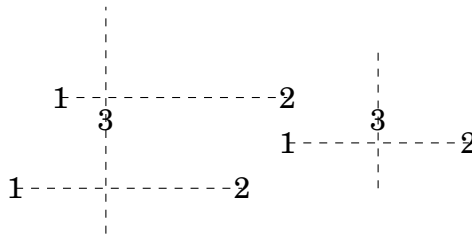


Dễ dàng giải bài toán này trong thời gian $O(n^2)$, bởi vì ta có thể đi qua tất cả các cặp đoạn thẳng có thể và kiểm tra xem chúng có giao nhau không. Tuy nhiên, ta có thể giải bài toán hiệu quả hơn trong thời gian $O(n \log n)$ bằng cách sử dụng thuật toán đường quét và một cấu trúc dữ liệu truy vấn khoảng.

Ý tưởng là xử lý các điểm cuối của các đoạn thẳng từ trái sang phải và tập trung vào ba loại sự kiện:

- (1) đoạn nằm ngang bắt đầu
- (2) đoạn nằm ngang kết thúc
- (3) đoạn thẳng đứng

Các sự kiện sau tương ứng với ví dụ:



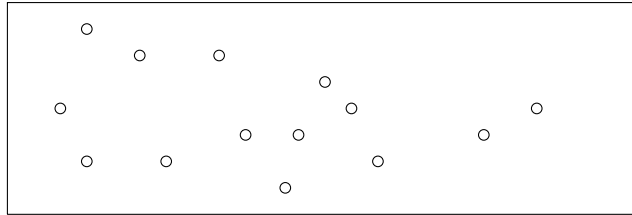
Ta đi qua các sự kiện từ trái sang phải và sử dụng một cấu trúc dữ liệu duy trì một tập các tọa độ y tại đó có một đoạn nằm ngang đang hoạt động. Tại sự kiện 1, ta thêm tọa độ y của đoạn vào tập, và tại sự kiện 2, ta xóa tọa độ y khỏi tập.

Các giao điểm được tính tại sự kiện 3. Khi có một đoạn thẳng đứng giữa các điểm y_1 và y_2 , ta đếm số lượng các đoạn nằm ngang đang hoạt động có tọa độ y nằm giữa y_1 và y_2 , và cộng số này vào tổng số giao điểm.

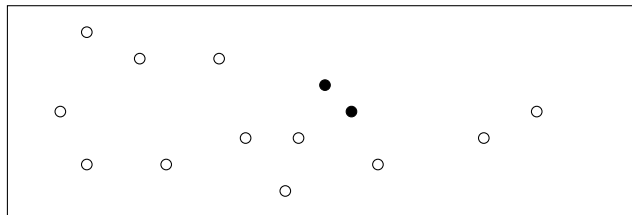
Để lưu trữ các tọa độ y của đoạn nằm ngang, ta có thể sử dụng cây chỉ số nhị phân hoặc cây phân đoạn, có thể kèm theo nén chỉ số. Khi sử dụng các cấu trúc như vậy, việc xử lý mỗi sự kiện mất thời gian $O(\log n)$, vì vậy tổng thời gian chạy của thuật toán là $O(n \log n)$.

30.2 Bài toán cặp điểm gần nhất

Cho một tập n điểm, bài toán tiếp theo của ta là tìm hai điểm có khoảng cách Euclid nhỏ nhất. Ví dụ, nếu các điểm là



ta nên tìm các điểm sau:

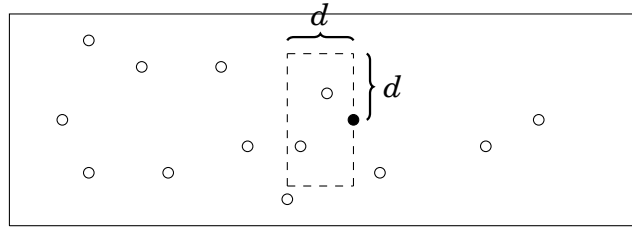


Đây là một ví dụ khác về bài toán có thể được giải trong thời gian $O(n \log n)$ bằng cách sử dụng thuật toán đường quét¹. Ta đi qua các điểm từ trái sang phải và duy trì một giá trị d : khoảng cách nhỏ nhất giữa hai điểm đã thấy cho tới thời điểm hiện tại. Tại mỗi điểm, ta tìm điểm gần nhất về bên trái. 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 trong phạm vi khoảng cách nhỏ hơn d , tọa độ x của điểm đó phải nằm trong khoảng $[x - d, x]$ và tọa độ y phải nằm trong khoảng $[y - d, y + d]$. Vì vậy, chỉ cần xét các điểm nằm trong những khoảng đó là đủ, điều này làm cho thuật toán hiệu quả.

Ví dụ, trong hình sau, vùng được đánh dấu bằng các đường nét đứt chứa các điểm có thể nằm trong phạm vi khoảng cách d từ điểm đang xét:

¹Ngoài cách tiếp cận này, còn có một thuật toán chia để trị thời gian $O(n \log n)$ [56] chia các điểm thành hai tập và đệ quy giải bài toán cho cả hai tập.



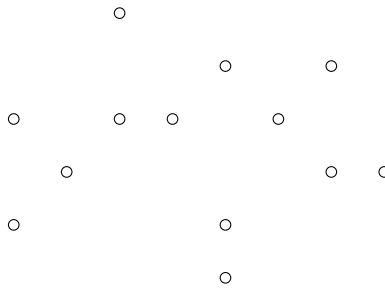
Hiệu quả của thuật toán dựa trên thực tế là vùng này luôn chỉ chứa $O(1)$ điểm. Ta có thể đi qua các điểm đó trong thời gian $O(\log n)$ bằng cách duy trì một tập các điểm có tọa độ x nằm trong khoảng $[x-d, x]$, theo thứ tự tăng dần theo tọa độ y của chúng.

Độ phức tạp thời gian của thuật toán là $O(n \log n)$, bởi vì ta đi qua n điểm và tìm cho mỗi điểm điểm gần nhất bên trái trong thời gian $O(\log n)$.

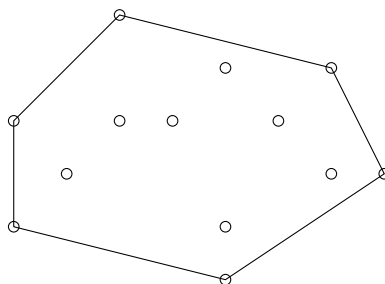
30.3 Bài toán bao lồi

Một **bao lồi** (convex hull) là đa giác lồi nhỏ nhất chứa tất cả các điểm của một tập cho trước. Tính lồi có nghĩa là một đoạn thẳng giữa bất kỳ hai đỉnh nào của đa giác đều nằm hoàn toàn bên trong đa giác.

Ví dụ, với các điểm



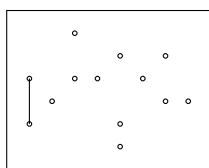
bao lồi như sau:



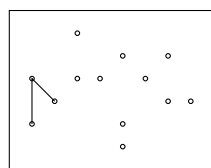
Thuật toán Andrew (Andrew's algorithm) [3] cung cấp một cách đơn giản để xây dựng bao lồi cho một tập điểm trong thời gian $O(n \log n)$. Thuật toán đầu tiên xác định điểm trái nhất và phải nhất, và sau đó xây dựng bao lồi thành hai phần: đầu tiên là phần trên và sau đó là phần dưới. Cả hai phần đều tương tự nhau, nên ta có thể tập trung vào việc xây dựng phần trên.

Đầu tiên, ta sắp xếp các điểm trước tiên theo tọa độ x và sau đó theo tọa độ y . Sau đó, ta đi qua các điểm và thêm từng điểm vào bao lồi. Luôn luôn sau khi thêm một điểm vào bao lồi, ta đảm bảo rằng đoạn thẳng cuối cùng trong bao lồi không rẽ trái. Miễn là nó rẽ trái, ta liên tục loại bỏ điểm áp chót khỏi bao lồi.

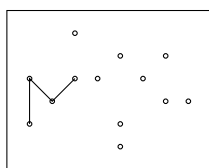
Các hình sau thể hiện cách thuật toán Andrew hoạt động:



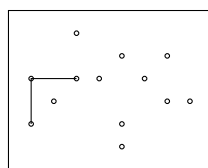
1



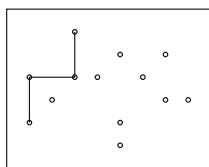
2



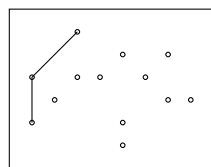
3



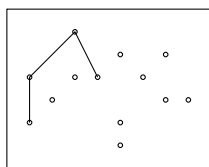
4



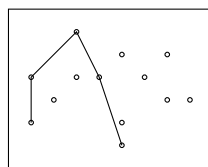
5



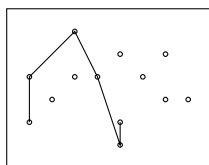
6



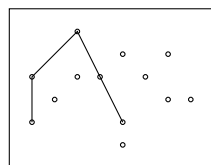
7



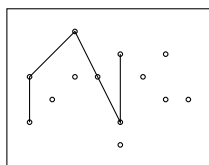
8



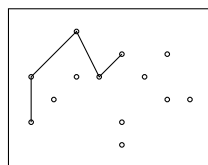
9



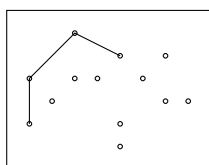
10



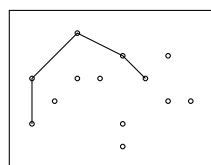
11



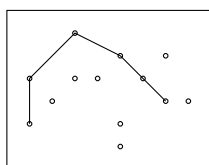
12



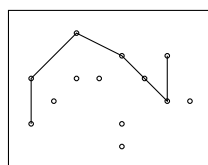
13



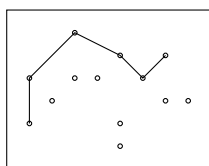
14



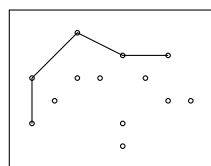
15



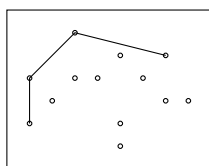
16



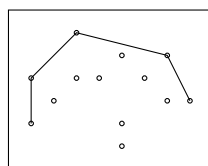
17



18



19



20

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.

