

# Competitive Programmer's Handbook

Antti Laaksonen

Draft Ngày 24 tháng 11 năm 2023



# Mục lục

<b>Preface</b>	<b>ix</b>
<b>I Basic techniques</b>	<b>1</b>
<b>1 Giới thiệu</b>	<b>3</b>
1.1 Ngôn ngữ lập trình . . . . .	3
1.2 Nhập và xuất . . . . .	4
1.3 Làm việc với số . . . . .	6
1.4 Rút gọn mã . . . . .	8
1.5 Mathematics . . . . .	11
1.6 Contests and resources . . . . .	16
<b>2 Độ phức tạp thời gian</b>	<b>19</b>
2.1 Quy tắc tính độ phức tạp . . . . .	19
2.2 Các lớp độ phức tạp . . . . .	22
2.3 Ước lượng tính hiệu quả . . . . .	23
2.4 Đoạn con có tổng lớn nhất . . . . .	24
<b>3 Sắp xếp</b>	<b>27</b>
3.1 Lý thuyết sắp xếp . . . . .	27
3.2 Sắp xếp trong C++ . . . . .	31
3.3 Tìm kiếm nhị phân . . . . .	34
<b>4 Cấu trúc dữ liệu</b>	<b>39</b>
4.1 Mảng động . . . . .	39
4.2 Cấu trúc tập hợp . . . . .	41
4.3 Cấu trúc ánh xạ . . . . .	42
4.4 Con trỏ lặp và làm việc trên dãy . . . . .	43
4.5 Một số cấu trúc dữ liệu khác . . . . .	46
4.6 So sánh với sắp xếp . . . . .	50
<b>5 Duyệt toàn bộ</b>	<b>53</b>
5.1 Sinh ra các tập hợp con . . . . .	53
5.2 Sinh hoán vị . . . . .	55
5.3 Quay lui . . . . .	56
5.4 Duyệt cận . . . . .	58

5.5	Duyệt phân tập . . . . .	61
<b>6</b>	<b>Giải thuật tham lam</b>	<b>63</b>
6.1	Bài toán đồng xu . . . . .	63
6.2	Xếp lịch . . . . .	64
6.3	Công việc và thời hạn . . . . .	66
6.4	Tối ưu tổng . . . . .	67
6.5	Nén dữ liệu . . . . .	68
<b>7</b>	<b>Quy hoạch động</b>	<b>71</b>
7.1	Bài toán đồng xu . . . . .	71
7.2	Dãy con tăng dài nhất . . . . .	76
7.3	Đường đi trên lưới . . . . .	77
7.4	Bài toán cái túi . . . . .	78
7.5	Khoảng cách chỉnh sửa . . . . .	80
7.6	Counting tilings . . . . .	81
<b>8</b>	<b>Phân tích khấu trừ</b>	<b>85</b>
8.1	Phương pháp hai con trỏ . . . . .	85
8.2	Phần tử nhỏ hơn gần nhất . . . . .	87
8.3	Cực tiểu đoạn tịnh tiến . . . . .	89
<b>9</b>	<b>Truy vấn trên đoạn</b>	<b>91</b>
9.1	Truy vấn mảng tĩnh . . . . .	92
9.2	Cây chỉ số nhị phân . . . . .	94
9.3	Cây phân đoạn . . . . .	97
9.4	Các kỹ thuật bổ sung . . . . .	101
<b>10</b>	<b>Xử lý bit</b>	<b>103</b>
10.1	Biểu diễn bit . . . . .	103
10.2	Các phép toán trên bit . . . . .	104
10.3	Biểu diễn tập hợp . . . . .	106
10.4	Tối ưu bit . . . . .	108
10.5	Quy hoạch động . . . . .	110
<b>II</b>	<b>Đồ thị</b>	<b>115</b>
<b>11</b>	<b>Cơ bản của Đồ thị</b>	<b>117</b>
11.1	Thuật ngữ trong đồ thị . . . . .	117
11.2	Biểu diễn đồ thị . . . . .	121
<b>12</b>	<b>Duyệt đồ thị</b>	<b>125</b>
12.1	Duyệt theo chiều sâu . . . . .	125
12.2	Duyệt theo chiều rộng . . . . .	127
12.3	Ứng dụng . . . . .	129

<b>13 Đường đi ngắn nhất</b>	<b>133</b>
13.1 Thuật toán Bellman–Ford . . . . .	133
13.2 Thuật toán Dijkstra . . . . .	136
13.3 Thuật toán Floyd–Warshall . . . . .	139
<b>14 Các thuật toán trên cây</b>	<b>143</b>
14.1 Duyệt cây . . . . .	144
14.2 Đường kính . . . . .	145
14.3 Mọi đường đi dài nhất . . . . .	147
14.4 Cây nhị phân . . . . .	149
<b>15 Cây khung</b>	<b>151</b>
15.1 Thuật toán Kruskal . . . . .	152
15.2 Cấu trúc dữ liệu các tập không giao nhau . . . . .	155
15.3 Thuật toán Prim . . . . .	157
<b>16 Đồ thị có hướng</b>	<b>159</b>
16.1 Sắp xếp Tô-pô . . . . .	159
16.2 Quy hoạch động . . . . .	162
16.3 Đồ thị mặt trời . . . . .	164
16.4 Phát hiện chu trình . . . . .	165
<b>17 Tính liên thông mạnh</b>	<b>169</b>
17.1 Thuật toán Kosaraju . . . . .	170
17.2 Bài toán 2SAT . . . . .	172
<b>18 Truy vấn trên cây</b>	<b>175</b>
18.1 Tìm nút tổ tiên . . . . .	175
18.2 Cây con và đường đi trên cây . . . . .	176
18.3 Tổ tiên chung gần nhất . . . . .	179
18.4 Xử lý Ngoại tuyến . . . . .	182
<b>19 Đường đi và chu trình</b>	<b>187</b>
19.1 Đường đi Euler . . . . .	187
19.2 Đường đi Hamilton . . . . .	191
19.3 Dây De Bruijn . . . . .	192
19.4 Mã đi tuần . . . . .	193
<b>20 Luồng và lát cắt</b>	<b>195</b>
20.1 Thuật toán Ford–Fulkerson . . . . .	196
20.2 Đường đi phân biệt . . . . .	200
20.3 Cặp ghép cực đại . . . . .	202
20.4 Tập đường bao phủ . . . . .	205

<b>III</b>	<b>Chủ đề nâng cao</b>	<b>209</b>
<b>21</b>	<b>Số học</b>	<b>211</b>
21.1	Số nguyên tố và Ước số . . . . .	211
21.2	Đồng dư thức . . . . .	216
21.3	Giải phương trình . . . . .	218
21.4	Kết quả khác . . . . .	220
<b>22</b>	<b>Tổ hợp</b>	<b>223</b>
22.1	Hệ số nhị thức . . . . .	224
22.2	Số Catalan . . . . .	226
22.3	Bao hàm-loại trừ . . . . .	229
22.4	Bổ đề Burnside . . . . .	230
22.5	Công thức Cayley . . . . .	231
<b>23</b>	<b>Ma trận</b>	<b>235</b>
23.1	Phép tính . . . . .	235
23.2	Hàm truy hồi tuyến tính . . . . .	238
23.3	Đồ thị và ma trận . . . . .	240
<b>24</b>	<b>Xác suất</b>	<b>243</b>
24.1	Tính toán . . . . .	243
24.2	Biến cố . . . . .	244
24.3	Biến ngẫu nhiên . . . . .	246
24.4	Chuỗi Markov . . . . .	248
24.5	Thuật toán ngẫu nhiên . . . . .	249
<b>25</b>	<b>Lý thuyết trò chơi</b>	<b>253</b>
25.1	Trạng thái trò chơi . . . . .	253
25.2	Trò chơi Nim . . . . .	255
25.3	Định lý Sprague–Grundy . . . . .	257
<b>26</b>	<b>Các thuật toán trên cây</b>	<b>261</b>
26.1	Các thuật ngữ về cây . . . . .	261
26.2	Cấu trúc Trie . . . . .	262
26.3	Băm chuỗi . . . . .	263
26.4	Thuật toán Z . . . . .	266
<b>27</b>	<b>Thuật toán chia căn</b>	<b>271</b>
27.1	Kết hợp thuật toán . . . . .	272
27.2	Tổng các số nguyên . . . . .	274
27.3	Thuật toán Mo . . . . .	275
<b>28</b>	<b>Cây phân đoạn</b>	<b>279</b>
28.1	Lan truyền thông minh . . . . .	280
28.2	Cây động . . . . .	283
28.3	Kết hợp với Cấu trúc dữ liệu khác . . . . .	286

28.4 Cây hai chiều . . . . .	286
<b>29 Hình học</b>	<b>289</b>
29.1 Số phức . . . . .	290
29.2 Điểm và đường . . . . .	292
29.3 Diện tích đa giác . . . . .	295
29.4 Hàm khoảng cách . . . . .	297
<b>30 Thuật toán đường quét</b>	<b>301</b>
30.1 Bài toán giao điểm . . . . .	302
30.2 Bài toán cặp điểm gần nhất . . . . .	303
30.3 Bài Toán bao lồi . . . . .	304
<b>Bibliography</b>	<b>307</b>
<b>Chỉ mục</b>	<b>313</b>





# Preface

The purpose of this book is to give you a thorough introduction to competitive programming. It is assumed that you already know the basics of programming, but no previous background in competitive programming is needed.

The book is especially intended for students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI) or in the International Collegiate Programming Contest (ICPC). Of course, the book is also suitable for anybody else interested in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will get a good general understanding of algorithms if you spend time reading the book, solving problems and taking part in contests.

The book is under continuous development. You can always send feedback on the book to `ahslaaks@cs.helsinki.fi`.

Helsinki, August 2019  
Antti Laaksonen



# **Phần I**

## **Basic techniques**



# Chương 1

## Giới thiệu

Lập trình thi đấu (Competitive Programming) là sự kết hợp của hai chủ đề chính: (1) thiết kế thuật toán và (2) cài đặt thuật toán

**Thiết kế thuật toán** bao gồm các kỹ năng giải quyết vấn đề và suy luận toán học. Kỹ năng phân tích bài toán và đưa ra giải pháp sáng tạo là đặc biệt cần thiết. Một thuật toán dùng để giải quyết bài toán phải vừa đúng vừa hiệu quả, và điều cốt lõi của việc giải bài, thường là tìm ra thuật toán phù hợp.

Trong lập trình thi đấu, ta cần nắm rõ các kiến thức về thuật toán. Thông thường, lời giải cho một bài toán là sự kết hợp của một vài kỹ thuật phổ biến với những ý tưởng, nhận xét mới. Các kỹ thuật, thuật toán xuất hiện trong lập trình thi đấu cũng góp phần hình thành nền tảng cho các nghiên cứu khoa học về thuật toán.

**Cài đặt thuật toán** đòi hỏi kỹ năng lập trình tốt. Trong lập trình thi đấu, lời giải được đánh giá bằng cách kiểm thử cài đặt của thuật toán thông qua các bộ dữ liệu thử. Do đó, ý tưởng thuật toán đúng là chưa đủ, mà việc cài đặt cũng cần phải chính xác.

Ngắn gọn và trực tiếp là một phong cách lập trình tốt trong thi đấu. Vì thời lượng của cuộc thi có hạn, cần lập trình nhanh. Không giống như khi lập trình phần mềm trong thực tế, các bài thi rất ngắn (mỗi chương trình giải dài nhất thường chỉ gồm vài trăm dòng mã), và không cần được bảo trì sau mỗi kỳ thi.

### 1.1 Ngôn ngữ lập trình

Ở thời điểm hiện tại, các ngôn ngữ lập trình phổ biến nhất được sử dụng trong các cuộc thi là C++, Python và Java. Ví dụ, ở Google Code Jam 2017, trong số 3000 thí sinh tốt nhất, 79 % sử dụng C++, 16 % sử dụng Python và 8 % sử dụng Java[29] Một số thí sinh sử dụng vài ngôn ngữ khác nhau cùng lúc.

Nhiều người nghĩ rằng C++ là lựa chọn tốt nhất cho lập trình thi đấu, C++ gần như luôn có sẵn trong các hệ thống thi đấu. Sử dụng C++ có hai lợi ích: đây là một ngôn ngữ cực kỳ hiệu quả, và thư viện chuẩn của C++ cung

cấp sẵn rất nhiều cấu trúc dữ liệu và thuật toán.

Tuy nhiên, bạn cũng nên thành thạo một vài ngôn ngữ lập trình và hiểu rõ về điểm mạnh của mỗi ngôn ngữ để có những lựa chọn phù hợp trong nhiều tình huống. Ví dụ, nếu cần xử lý số nguyên lớn thì Python là một lựa chọn tốt, bởi vì ngôn ngữ này cung cấp sẵn các phép toán hỗ trợ tính toán số nguyên lớn. Dẫu vậy, đa số bài toán trong các cuộc thi lập trình đều được ban ra đề tính toán, để tránh việc sử dụng một ngôn ngữ nào đó có lợi thế hơn hẳn các ngôn ngữ khác, tránh bất công.

Tất cả các ví dụ mẫu trong sách này đều được viết bằng C++, và thường xuyên sử dụng các cấu trúc dữ liệu, thuật toán trong thư viện chuẩn STL C++. Các chương trình được viết theo chuẩn C++11, đa phần các cuộc thi ngày nay đều hỗ trợ chuẩn này. Nếu bạn chưa thể lập trình bằng C++, hãy bắt đầu học ngay nhé.

## Khung sườn chương trình C++

Một mẫu chương trình C++ dùng trong lập trình thi đấu trông như thế này:

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

using namespace std;

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

Dòng `#include` ở đầu chương trình là một tính năng của trình biên dịch `g++` cho phép ta sử dụng toàn bộ các thư viện chuẩn. Do đó, không cần phải khai báo từng thư viện như `iostream`, `vector` và `algorithm`, mà các thư viện này đều được tự động khai báo đầy đủ thông qua câu lệnh trên.

Dòng `using` giúp ta sử dụng trực tiếp các lớp và hàm của thư viện chuẩn trong mã nguồn. Chẳng hạn, nếu không dùng `using` thì chúng ta phải viết `std::cout`, trong khi nếu dùng ta chỉ cần viết `cout`

Lệnh dưới đây sẽ biên dịch mã nguồn:

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

Lệnh này sẽ sinh ra một chương trình `test` từ đoạn mã nguồn nguồn `test.cpp`. Trình biên dịch sẽ theo chuẩn C++11 (`-std=c++11`), tối ưu các lệnh (`-O2`) và hiện các thông báo về các lỗi có thể có (`-Wall`).

## 1.2 Nhập và xuất

Trong hầu hết các kỳ thi, luồng nhập xuất chuẩn được dùng để đọc và ghi dữ liệu. Trong C++, luồng vào chuẩn là `cin` và luồng ra chuẩn là `cout`. Ngoài ra, ta cũng có thể sử dụng các hàm `scanf` và `printf` của C.

Đầu vào thường gồm dữ liệu số và chuỗi ký tự được phân cách bởi dấu cách hoặc dấu xuống dòng. Dữ liệu được đọc vào bằng lệnh `cin` như sau:

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

Cách viết này luôn đúng khi có ít nhất một dấu cách hay dấu xuống dòng giữa mỗi phần tử nhập vào. Lệnh phía trên có thể đọc dữ liệu ở cả hai dạng dưới đây:

```
123 456 monkey
```

```
123    456  
monkey
```

Luồng ra `cout` được dùng để in ra như sau:

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

Nhập và xuất đôi lúc là "nút thắt cổ chai" khiến chương trình chạy chậm. Ta thêm vào các dòng dưới đây ở đầu chương trình để việc nhập xuất được hiệu quả hơn:

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

Lưu ý: sử dụng ký tự xuống dòng `"\n"` nhanh hơn `endl`, vì `endl` bao gồm thêm cả thao tác "flush"<sup>1</sup>

Hàm `scanf` và `printf` của C tương tự như luồng `cin`, `cout` chuẩn của C++. Thường những lệnh này nhanh hơn một chút `cin`, `cout`, nhưng chúng khó sử dụng hơn. Các lệnh dưới đây đọc vào 2 số nguyên:

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

Đoạn mã dưới đây in ra hai số nguyên:

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

Đôi khi, chương trình yêu cầu đọc dữ liệu theo dòng, dữ liệu có thể chứa

---

<sup>1</sup>Thao tác flush giúp đảm bảo nội dung truyền vào luồng `cout` được in ra ngay lập tức. Nếu không flush, C++ sẽ sử dụng một bộ đệm (buffer). Khi nội dung cần in ra đạt tới kích thước của buffer, lúc này chúng mới được in ra một lần. Điều này giúp cho các thao I/O được nhanh vì chỉ cần in ít lần.

dấu cách. Trường hợp này ta dùng hàm `getline`:

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

Nếu không biết trước số lượng phần tử cần nhập vào, hãy dùng vòng lặp như sau:

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

Vòng lặp này đọc lần lượt từng phần tử cho tới khi không còn phần tử nào trong đầu vào.

Một số hệ thống chấm yêu cầu nhập xuất bằng tệp. Một giải pháp dễ dàng là ta vẫn sử dụng các luồng nhập xuất chuẩn như thông thường, nhưng thêm những dòng sau vào đầu chương trình.

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

Nhờ đoạn mã trên, chương trình giờ sẽ đọc vào tệp "input.txt" và ghi ra tệp "output.txt"

## 1.3 Làm việc với số

### Số nguyên

Kiểu số nguyên thường được dùng trong lập trình thì đầu nhất là `int`, đây là kiểu dữ liệu 32-bit với giá trị trong miền  $-2^{31} \dots 2^{31} - 1$ , xấp xỉ với  $-2 \cdot 10^9 \dots 2 \cdot 10^9$ . Nếu kiểu dữ liệu `int` không đủ để lưu trữ, ta có thể sử dụng kiểu số nguyên 64-bit `long long`. Kiểu dữ liệu này có miền giá trị là  $-2^{63} \dots 2^{63} - 1$ , tương đương  $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$ .

Đoạn chương trình dưới đây định nghĩa một biến `long long`:

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

Hậu tố LL chỉ định rõ kiểu dữ liệu của giá trị số là `long long`.

Một lỗi thường gặp khi sử dụng `long long` đó là ở chỗ khác trong chương trình ta vẫn dùng xen lẫn `int`. Điều này gây ra một lỗi khó phát hiện, như ví dụ dưới đây:

```
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 là kiểu `int` và kết quả của `a*a` cũng là kiểu `int`. Chính vì thế, biến `b` sẽ chứa



kết quả sai. Ta có thể sửa lỗi này bằng cách thay đổi kiểu của  $a$  thành `long long` hoặc bằng cách thay đổi biểu thức thành `(long long)a*a`.

Thông thường, dùng kiểu `long long` là đủ để làm phần lớn các bài trong kỳ thi. Tuy nhiên, cũng nên biết thêm rằng trình biên dịch `g++` còn cung cấp một kiểu dữ liệu 128-bit đó là `__int128_t` với miền giá trị trong  $-2^{127} \dots 2^{127} - 1$ , tương đương với  $-10^{38} \dots 10^{38}$ . Tuy nhiên, kiểu dữ liệu này không có sẵn trên mọi hệ thống chấm bài.

## Đồng dư thức

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

Đôi khi kết quả của bài toán là một số nguyên rất lớn nhưng chỉ cần in ra "theo mod  $m$ ", tức là, phần dư của phép chia kết quả cho  $m$  (ví dụ, "mod  $10^9 + 7$ "). Mục đích là, kể cả kết quả là một giá trị số rất lớn ta cũng chỉ cần sử dụng kiểu `int` và `long long` là đủ.

Một tính chất quan trọng của số dư là trong các phép cộng, trừ và nhân, có thể chia lấy dư cho từng số trước khi thực hiện phép tính.

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

Do đó, chúng ta có thể chia lấy dư ngay sau mỗi phép toán để giá trị các số không bao giờ bị lớn quá.

Ví dụ, đoạn chương trình sau tính  $n!$ , giai thừa của  $n$ , theo mod  $m$ :

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

Thường chúng ta muốn phần dư có giá trị trong  $0 \dots m - 1$ . Tuy nhiên, trong C++ và các ngôn ngữ khác khi chia lấy dư một số âm thì kết quả sẽ là một số âm hoặc là số 0. Để không tính ra số dư âm ta làm như sau: đầu tiên cứ tính phần dư theo cách thông thường sau đó cộng vào kết quả thêm  $m$  nếu như nó là số âm:

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

Tuy nhiên, điều này chỉ cần thiết nếu xuất hiện phép trừ trong mã nguồn, khi đó số dư có thể âm.

## Số thực

Kiểu số thực thông dụng trong lập trình thi đấu là số thực 64-bit double, và một kiểu mở rộng của trình biên dịch g++ là long double với 80-bit. Trong đa số các trường hợp thì kiểu double là đủ, nhưng long double có độ chính xác cao hơn.

Đề bài thường cho ta biết độ chính xác cần đạt được của kết quả đưa ra. Một cách đơn giản để in ra số thực là sử dụng hàm printf và chỉ định rõ số lượng chữ số phân thập phân trong xâu định dạng. Ví dụ, đoạn mã dưới đây in ra giá trị của  $x$  với 9 số phân thập phân:

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

Một vấn đề thường gặp khi dùng số thực là một vài giá trị sẽ không thể biểu diễn được một cách chính xác, và sẽ có sai lệch do làm tròn số. Ví dụ, đoạn mã dưới đây cho ra kết quả không như mong đợi:

```
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.

Không nên so sánh các số thực bằng phép toán `==`, vì có thể giá trị của hai số bằng nhau nhưng do lỗi làm tròn mà kết quả so sánh ra khác nhau. Một phương án tốt hơn để so sánh số thực là ta xem như hai số bằng nhau nếu chênh lệch giữa chúng nhỏ hơn  $\epsilon$ , với  $\epsilon$  là một giá trị đủ nhỏ.

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

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

Lưu ý rằng, trong khi các số thực nhìn chung không được biểu diễn chính xác, các số nguyên có giá trị nằm trong giới hạn của kiểu dữ liệu vẫn có thể được lưu chính xác trong các biến số thực. Ví dụ, sử dụng kiểu double, ta có thể biểu diễn chính xác mọi số nguyên có giá trị tuyệt đối không quá  $2^{53}$ .

## 1.4 Rút gọn mã

Code ngắn là một khuyến nghị trong lập trình thi đấu, vì trong thời gian cho phép, chương trình phải được hoàn thành theo cách nhanh nhất có thể. Vì điều này, các thí sinh lập trình thi đấu thường định nghĩa những tên ngắn hơn cho các kiểu dữ liệu và những phần khác của chương trình.

## Tên kiểu

Sử dụng lệnh typedef có thể định nghĩa tên mới ngắn hơn cho một kiểu dữ liệu.

Ví dụ, tên long long là một tên dài, do đó ta có thể gán tên ngắn hơn cho nó thành ll bằng:

```
typedef long long ll;
```

Sau đó, đoạn code này:

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

Có thể viết theo cách ngắn hơn như này:

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

Lệnh typedef có thể được dùng cho các kiểu dữ liệu phức tạp hơn. Ví dụ, đoạn code dưới đây đặt tên vi cho kiểu vector số nguyên và tên pi cho kiểu pair gồm hai số nguyên.

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

## Macros

Một cách khác để viết code ngắn hơn là định nghĩa các **macros**. Một macro hiểu là một chuỗi ký tự trong code sẽ được thay thế trước khi biên dịch chương trình. Trong C++, macro được định nghĩa bằng từ khóa #define keyword.

Ví dụ, chúng ta có thể định nghĩa macro như ví dụ dưới đây:

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

Sau đó, chương trình thế này:

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

sẽ được viết theo cách ngắn gọn như sau:

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

Một macro có thể chứa tham số các tham số có thể làm cho các đoạn chương trình lặp hay cấu trúc dữ liệu trở nên ngắn gọn hơn Ví dụ, chúng ta có thể định nghĩa một macro như dưới đây:

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

Sau đoạn này, chương trình này

```
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 các macro có thể gây ra lỗi, khi đó việc xác định lỗi trở nên khó khăn. Ví dụ, một macro sau tính căn bậc hai của một số:

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

Macro này *không* thường xuyên hoạt động như chúng ta mong đợi. Ví dụ, đoạn code sau

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

tương ứng với đoạn:

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

Một cách viết khác của macro này như sau:

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

Khi đó, chương trình sẽ thế này

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

tương ứng với đoạn code sau:

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

## 1.5 Mathematics

Toán học đóng một vai trò quan trọng trong lập trình thi đấu, bạn không thể trở thành một lập trình viên giỏi mà thiếu kĩ năng toán học. Phần này bàn về một số nội dung toán học quan trọng và một số công thức cần thiết được dùng ở phần sau của cuốn sách.

### Công thức tổng

Mỗi tổng đều có dạng

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

với  $k$  là một số nguyên dương. Có một công thức dạng đóng là một đa thức bậc  $k+1$

Ví dụ <sup>2</sup> Một công thức tổng nổi tiếng được gọi là **công thức Faulhaber**. Công thức này rất phức tạp, khó thể hiện được ở đây,

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

và

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

**Cấp số cộng** là một dãy số thỏa mãn điều kiện chênh lệch giữa hai phần tử cạnh nhau là một hằng số. Ví dụ,

$$3, 7, 11, 15$$

là một cấp số cộng với hằng số 4 (công sai là 4) Tổng các phần tử trong cấp số cộng có thể được tính bằng công thức:

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

với  $a$  là số đầu tiên của dãy  $b$  là số cuối cùng của dãy  $n$  là số lượng phần tử. Ví dụ:

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

Công thức được chứng minh: tổng gồm  $n$  số và giá trị của mỗi số trung bình là  $(a+b)/2$

Một **cấp số nhân** là dãy các số thỏa mãn điều kiện tỉ lệ giữa hai phần tử cạnh nhau luôn là một hằng số. Ví dụ,

$$3, 6, 12, 24$$

là một cấp số nhân với hằng số là 2 (công bội là 2) Tổng các phần tử trong cấp số nhân có thể được tính bằng công thức sau:

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

với  $a$  là số đầu tiên của dãy  $b$  là số cuối cùng của dãy  $k$  là số tỉ lệ giữa hai phần tử cạnh nhau. Ví dụ,

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

Công thức này có thể được chứng minh như sau. Gọi

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

Nhân hai vế với  $k$ , ta có:

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

và giải phương trình

$$kS - S = bk - a$$

ta có được công thức ở trên.

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

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

Một **tổng điều hòa - harmonic sum** là tổng có dạng

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

Giới hạn trên của tổng điều hòa là  $\log_2(n) + 1$ . Cụ thể là, chúng ta có thể thay đổi mỗi  $1/k$  để  $k$  trở thành một 2 mũ x sao cho không vượt quá  $k$  Ví dụ, khi  $n = 6$ , chúng ta có thể ước lượng tổng như sau:

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

Giới hạn trên này gồm có  $\log_2(n) + 1$  phần ( $1, 2 \cdot 1/2, 4 \cdot 1/4$ , etc.), và giá trị của mỗi phần tối đa là 1.

## Lý thuyết tập hợp

Một **tập hợp** là bộ các phần tử. Ví dụ, tập hợp

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

chứa các phần tử 2, 4 và 7. Biểu tượng  $\emptyset$  biểu diễn tập rỗng, và  $|S|$  là kích thước của tập hợp  $S$ , chính là số phần tử của tập hợp đó. Ví dụ; tập hợp ở ví dụ trên có  $|X| = 3$

Nếu một tập hợp  $S$  gồm một phần tử  $x$ , ta viết  $x \in S$ , ngược lại, viết  $x \notin S$ . Ví dụ, ở tập hợp phía trên

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

Các tập hợp mới có thể được tạo thành bằng cách sử dụng các phép toán trên tập hợp:

- Phép Giao **giao**  $A \cap B$  bao gồm các phần tử thuộc cả  $A$  và  $B$  Ví dụ: Nếu  $A = \{1, 2, 5\}$  và  $B = \{2, 4\}$ , thì  $A \cap B = \{2\}$ .
- **Phép hợp**  $A \cup B$  bao gồm các phần tử thuộc  $A$  hoặc thuộc  $B$  hoặc thuộc cả hai. Ví dụ, nếu  $A = \{3, 7\}$  và  $B = \{2, 3, 8\}$ , thì  $A \cup B = \{2, 3, 7, 8\}$ .
- **Phần bù**  $\bar{A}$  gồm các phần tử không thuộc tập  $A$  Các thành phần của phần bù phụ thuộc vào tập universal gồm tất cả các phần tử có thể. Ví dụ, nếu  $A = \{1, 2, 5, 7\}$  và tập universal là  $\{1, 2, \dots, 10\}$ , thì  $\bar{A} = \{3, 4, 6, 8, 9, 10\}$ .
- **Phép trừ**  $A \setminus B = A \cap \bar{B}$  gồm các phần tử trong  $A$  nhưng không có mặt trong  $B$ . Lưu ý rằng  $BB$  có thể chứa các phần tử không thuộc  $A$ . Ví dụ, nếu  $A = \{2, 3, 7, 8\}$  và  $B = \{3, 5, 8\}$ , thì  $A \setminus B = \{2, 7\}$ .

Nếu mỗi phần tử của  $A$  đồng thời cũng thuộc  $S$  chúng ta gọi  $A$  là một **tập con** của tập  $S$ , kí hiệu là  $A \subset S$ . Một tập  $S$  luôn có  $2^{|S|}$  tập con, gồm cả tập rỗng. Ví dụ, các tập con của tập  $\{2, 4, 7\}$  là

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

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

Chúng ta có thể tạo thành một tập hợp sử dụng quy tắc:

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

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

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

gồm tất cả các số nguyên chẵn.

## Logic

Giá trị của một biểu thức logic có thể là **true** (1) hoặc **false** (0). Các phép toán logic quan trọng nhất gồm  $\neg$  (**phủ định - negation**),  $\wedge$  (**hợp - conjunction**),  $\vee$  (**tuyển - disjunction**),  $\Rightarrow$  (**suy ra - implication**) và  $\Leftrightarrow$  (**tương đương - equivalence**). Bảng dưới đây thể hiện ý nghĩa của các phép toán trên:

$A$	$B$	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
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 trái ngược với  $A$ . Biểu thức  $A \wedge B$  là true nếu cả hai  $A$  và  $B$  đều true, biểu thức  $A \vee B$  là true nếu  $A$  hoặc  $B$  hoặc cả hai là true. Biểu thức  $A \Rightarrow B$  có giá trị true nếu  $A$  false, và cả  $B$  true. Biểu thức  $A \Leftrightarrow B$  true nếu  $A$  và  $B$  cùng true hoặc cùng false.

Một **vị từ** là một biểu thức có giá trị true hoặc false tùy thuộc vào giá trị của tham số. Vị từ thường được kí hiệu bằng chữ in hoa. Ví dụ, chúng ta có thể định nghĩa một vị từ  $P(x)$  có giá trị true nếu  $x$  là số nguyên tố. Sử dụng định nghĩa này,  $P(7)$  có giá trị true nhưng  $P(8)$  có giá trị false.

Một **lượng từ** kết nối một biểu thức logic với các phần tử của một tập hợp. Lượng từ quan trọng thường được sử dụng là  $\forall$  (**với mọi**) và  $\exists$  (**tồn tại**). Ví dụ,

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

nghĩa là với mỗi phần tử  $x$  trong tập hợp, có một phần tử  $y$  trong tập hợp thỏa mãn  $y$  nhỏ hơn  $x$ . Điều này đúng khi trong tập các phần tử số nguyên nhưng sai trong tập hợp các phần tử số tự nhiên.

Sử dụng kí hiệu ở trên chúng ta có thể biểu diễn nhiều loại mệnh đề logic khác nhau. Ví dụ:

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

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 có tích bằng  $x$ . Điều này đúng với tập các số nguyên.

## Hàm-Functions

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

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

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

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

**giai thừa**  $n!$  có thể được định nghĩa

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

hoặc định nghĩa một cách đệ quy:

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



**Số Fibonacci** xuất hiện ở nhiều tình huống. Có thể định nghĩa một cách đệ quy như sau:

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

Một vài số Fibonacci đầu tiên

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

tính các số Fibonacci:

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

## Logarithms

**Logarithm** của một số  $x$  được kí hiệu là  $\log_k(x)$ , với  $k$  là cơ số của logarithm. Theo định nghĩa,  $\log_k(x) = a$  khi  $k^a = x$ .

Một thuộc tính hữu ích của logarithm là  $\log_k(x)$  bằng với số lần ta chia  $x$  cho  $k$  cho tới khi kết quả bằng 1.

Ví dụ,  $\log_2(32) = 5$  vì 32 chia cho 2 5 lần để có kết quả là 1

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

Logarithm thường được sử dụng để đánh giá các thuật toán ở mỗi bước thực hiện, kích thước dữ liệu giảm một nửa. Do đó, ta có thể ước lượng độ hiệu quả của thuật toán sử dụng logarithm.

Logarithm của một tích:

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

và kết quả là,

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

Thêm vào đó, logarithm của một thương:

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

Một công thức hữu ích:

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

và sử dụng công thức này bạn có thể tính được logarithm ở cơ số bất kì nếu có cách tính một số logarithm ở một vài cơ số cố định.

**logarithm tự nhiên**  $\ln(x)$  của một số  $x$  là logarithms cơ số  $e \approx 2.71828$ . một tính chất khác của logarithms là số lượng chữ số của một số nguyên  $x$  ở cơ số  $b$  chính bằng  $\lfloor \log_b(x) + 1 \rfloor$ . Ví dụ, biểu diễn số 123 ở cơ số 2 là 1111011 và do đó  $\lfloor \log_2(123) + 1 \rfloor = 7$ .

## 1.6 Contests and resources

### IOI

The International Olympiad in Informatics (IOI) là kì thi lập trình hàng năm dành cho học sinh trung học. Mỗi quốc gia được gửi một đội gồm 4 thành viên tham gia. Thường có khoảng 300 thí sinh tham gia từ 80 quốc gia khác nhau.

Kì thi IOI gồm 2 contest, mỗi contest kéo dài 5 tiếng. Ở cả hai bài thi, thí sinh được yêu cầu giải 3 bài toán với độ khó khác nhau. Các bài thường được chia thành nhiều subtasks, mỗi subtask được một số điểm nào đó. Dù thí sinh tham gia thi theo đội, nhưng cũng có thể tham gia với tư cách cá nhân.

Hầu hết các chủ đề trong sách này là những chủ đề thường gặp trong các kì thi IOI.

Các thí sinh IOI được lựa chọn thông qua các cuộc thi cấp quốc gia. Trước khi diễn ra IOI, có thể nhiều cuộc thi cấp khu vực được tổ chức, như BOI, CEOI, APIO.

Một số quốc gia tổ chức thi online dành cho các thí sinh IOI tương lai, như COCI, USACO. Thêm vào đó, nhiều bài toán online từ cuộc thi của Ba Lan [60].

### ICPC

ICPC là cuộc thi hàng năm dành cho sinh viên các trường Đại học. Mỗi đội tham gia thi gồm có 3 sinh viên, không giống như cuộc thi IOI, các sinh viên làm việc chung với nhau, và chỉ có duy nhất một máy tính dành cho mỗi đội.

ICPC gồm nhiều pha thi, cuối cùng đội tốt nhất được mời tham dự vòng chung kết thế giới. Trong khi có hàng chục nghìn thí sinh tham gia cuộc thi và chỉ có một số lượng nhỏ được vào vòng chung kết. Vào 2017, có 133 thí sinh tham gia vòng cuối cùng.

Trong mỗi cuộc thi ICPC, các đội có 5 tiếng để giải quyết khoảng 10 bài toán. Một phương án thuật toán đúng được chấp nhận (AC) chỉ khi nó chạy đúng tất cả các test một cách hiệu quả. Trong quá trình thi, các thí sinh có thể xem kết quả của các đội khác, nhưng giờ cuối cùng, bảng điểm bị đóng băng và các đội không thể xem kết quả của lượt nộp bài cuối cùng.

Các chủ đề có thể xuất hiện trong ICPC không rõ ràng như các bài toán trong IOI. Trong mọi trường hợp, rõ ràng cần rất nhiều kiến thức cho ICPC, đặc biệt là các kiến thức, kĩ năng toán học.

### Online contests

Có nhiều cuộc thi online được tổ chức thường xuyên dành cho mọi người. Tại thời điểm này, các trang thi online nổi bật có thể kể đến là Codeforces, ở đây các cuộc thi được tổ chức hàng tuần. Ở Codeforces, các thí sinh được chia làm 2 mức độ (division): các thí sinh mới (div2) và các thí sinh có nhiều kinh

nghiệm hơn (Div1). Các trang khác như AtCoder, CS Academy, HackerRank và Topcoder.

Một số công thi cũng tổ chức các cuộc thi online và vòng chung kết là thi trực tiếp (onsite). Ví dụ như Facebook Hacker Cup, Google Code Jam và Yandex.Algorithm. Tất nhiên, các công ti cũng sử dụng các cuộc thi này để tuyển nhân viên: kết quả tốt trong cuộc thi thể hiện một cách khá chính xác về kỹ năng của một người.

## Books

Có một số cuốn tài liệu (bên cạnh cuốn này) tập trung vào mảng lập trình thi đấu và giải quyết vấn đề thuật toán như:

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

Hai cuốn sách đầu tiên dành cho người mới, còn cuốn cuối cùng là tài liệu mở rộng, nâng cao hơn.

Các cuốn sách thuật toán chung nhìn chung phù hợp với các thí sinh lập trình thi đấu. Một số cuốn phổ biến như:

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



## Chương 2

# Độ phức tạp thời gian

Trong lập trình thi đấu, độ hiệu quả của thuật toán rất quan trọng. Thông thường, ta có thể dễ dàng nghĩ ra một thuật toán cho ra kết quả đúng nhưng chạy chậm. Tuy nhiên, thử thách thật sự là sáng tạo ra một thuật đủ nhanh để giải quyết trọn vẹn bài toán. Nếu thuật toán quá chậm, bài giải chỉ có thể đạt được một phần số điểm của bài, hoặc thậm chí không được điểm nào cả.

**Độ phức tạp thời gian** của một thuật toán ước lượng thời gian chạy của nó với các bộ dữ liệu vào khác nhau. Ý tưởng là ta sẽ mô tả độ hiệu quả này dưới dạng một hàm số theo kích thước bộ dữ liệu vào. Nhờ việc tính được độ phức tạp thời gian, ta có thể xác định một thuật toán có đủ nhanh hay không mà không cần cài đặt để chạy thử.

### 2.1 Quy tắc tính độ phức tạp

Độ phức tạp thời gian của một thuật toán được kí hiệu là  $O(\dots)$  với dấu ba chấm đại diện cho một hàm số nào đó. Thông thường, biến  $n$  là kích thước bộ dữ liệu vào. Ví dụ, nếu bộ dữ liệu vào là một mảng gồm các số, thì  $n$  sẽ là kích thước của mảng, và nếu bộ dữ liệu vào là một chuỗi,  $n$  sẽ là độ dài của chuỗi đó.

#### Thuật toán chứa vòng lặp

Khi thuật toán chạy chậm, một lí do phổ biến là do nó chứa quá nhiều vòng lặp lồng nhau duyệt qua bộ dữ liệu vào. Càng nhiều vòng lặp lồng nhau thì thuật toán sẽ chạy càng chậm. Nếu có  $k$  vòng lặp lồng nhau, độ phức tạp thời gian sẽ là  $O(n^k)$ .

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

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

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

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

## Thứ bậc của độ lớn

Độ phức tạp thời gian không cho chúng ta biết chính xác số lần thực thi mã trong một vòng lặp, nó chỉ cho biết thứ bậc của độ lớn. Trong ví dụ dưới đây, đoạn mã trong các vòng lặp được thực thi  $3n$ ,  $n+5$ , và  $\lfloor n/2 \rfloor$  lần, tuy nhiên độ phức tạp của tất cả đều là  $O(n)$  (bậc 1).

```
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  $O(n^2)$  (bậc 2):

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

## Thuật toán với nhiều giai đoạn

Nếu thuật toán gồm nhiều giai đoạn liên kế nhau, tổng độ phức tạp thời gian sẽ là độ phức tạp thời gian lớn nhất trong các đoạn. Bởi vì công đoạn chậm nhất thường là "nút thắt cổ chai" tác động đến thời gian chạy của toàn chương trình.

Ví dụ, chương trình dưới đây gồm ba đoạn với độ phức tạp lần lượt là  $O(n)$ ,  $O(n^2)$  và  $O(n)$ . Vì vậy, tổng độ phức tạp thời gian sẽ 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
}

```

## Thuật toán chứa nhiều biến

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

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

```

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

```

## Thuật toán chứa đệ quy

Độ phức tạp thời gian của một hàm đệ quy phụ thuộc vào số lần gọi hàm và độ phức tạp thời gian của riêng một lần gọi hàm. Tổng độ phức tạp thời gian sẽ là tích của hai giá trị trên.

Ví dụ, xét hàm dưới đây:

```

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

```

Khi ta gọi  $f(n)$ , chương trình thực hiện  $n$  lần gọi hàm, bên cạnh đó, độ phức tạp thời gian của mỗi lần gọi là  $O(1)$ . Do đó, tổng độ phức tạp thời gian sẽ là  $O(n)$ .

Xét ví dụ khác với hàm sau:

```

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

```

Trong trường hợp này, mỗi lần gọi hàm sẽ sinh ra thêm hai lần gọi hàm khác, trừ khi  $n = 1$ .

Xét trường hợp  $g$  được gọi với tham số  $n$ . Bảng sau đây cho biết các lần gọi hàm được sinh ra từ lần gọi hàm đầu tiên này.

hàm được gọi	số lần gọi
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	$2^{n-1}$

Dựa vào bảng trên, ta có thể tìm được 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 gồm độ phức tạp thời gian phổ biến của các thuật toán:

$O(1)$  Thời gian chạy của một thuật toán có độ phức tạp **hằng số** không phụ thuộc vào kích thước dữ liệu đầu vào. Một ví dụ điển hình cho thuật toán có độ phức tạp hằng số là một công thức tính toán trực tiếp ra kết quả.

$O(\log n)$  Một thuật toán có độ phức tạp **logarit** thường gồm nhiều bước, trong đó kích thước dữ liệu đầu vào được chia đôi sau mỗi bước. Thời gian chạy của một thuật toán như trên có dạng logarit, vì  $\log_2 n$  tương ứng với số lần  $n$  phải chia 2 để đạt đến 1.

$O(\sqrt{n})$  Một thuật toán có độ phức tạp **căn bậc hai** chạy chậm hơn  $O(\log n)$ , nhưng chạy nhanh hơn  $O(n)$ . Một tính chất đặc biệt của căn bậc hai đó chính là  $\sqrt{n} = n/\sqrt{n}$ , nên về mặt nào đó, ta có thể hình dung  $\sqrt{n}$  nằm "chính giữa" dữ liệu vào.

$O(n)$  Một thuật toán có độ phức tạp **tuyến tính** duyệt qua toàn bộ phần tử của dữ liệu vào một số lần cố định. Đây thường là độ phức tạp tốt nhất khả thi, bởi vì thông thường ta đều cần đọc qua tất cả các phần tử của dữ liệu vào trước khi đưa ra đáp án.

$O(n \log n)$  Độ phức tạp này thường gợi ý thuật toán có thao tác sắp xếp dữ liệu đầu vào, vì độ phức tạp của các thuật toán sắp xếp hiệu quả là  $O(n \log n)$ . Ngoài ra, cũng có khả năng thuật toán sử dụng một cấu trúc dữ liệu nào đó với các thao tác độ phức tạp  $O(\log n)$ .

$O(n^2)$  Một thuật toán có độ phức tạp **bình phương** thường chứa hai vòng lặp lồng nhau. Một trong số các khả năng là thuật toán duyệt qua tất cả cặp phần tử trong dữ liệu vào trong thời gian  $O(n^2)$ .



$O(n^3)$  Một thuật toán có độ phức tạp **lập phương** thường chứa ba vòng lặp lồng nhau. Một trong số các khả năng là thuật toán duyệt qua tất cả bộ ba phần tử trong dữ liệu vào trong thời gian  $O(n^3)$ .

$O(2^n)$  Độ phức tạp này thường gợi ý rằng thuật toán sẽ duyệt qua tất cả các tập con của tập phần tử trong bộ dữ liệu vào. Ví dụ, các tập con của  $\{1, 2, 3\}$  là  $\emptyset$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{1, 2\}$ ,  $\{1, 3\}$ ,  $\{2, 3\}$  và  $\{1, 2, 3\}$ .

$O(n!)$  Độ phức tạp này thường gợi ý rằng thuật toán sẽ duyệt qua tất cả các hoán vị của tập phần tử trong bộ dữ liệu vào. Ví dụ, các hoán vị của tập  $\{1, 2, 3\}$  là  $(1, 2, 3)$ ,  $(1, 3, 2)$ ,  $(2, 1, 3)$ ,  $(2, 3, 1)$ ,  $(3, 1, 2)$  và  $(3, 2, 1)$ .

Một thuật toán có độ phức tạp **đa thức** nếu độ phức tạp thời gian của nó tối đa là  $O(n^k)$  với  $k$  là hằng số. Tất cả các loại độ phức tạp nêu trên trừ  $O(2^n)$  và  $O(n!)$  đều là đa thức. Trong thực tế, hằng số  $k$  thường nhỏ, vì vậy các thuật toán có độ phức tạp đa thức được xem là *hiệu quả*.

Đa số các thuật toán trong sách này đều có độ phức tạp đa thức. Tuy nhiên, vẫn còn nhiều bài toán quan trọng nhưng chưa tìm được thuật toán đa thức, nghĩa là chưa ai nghĩ ra cách giải nào hiệu quả cho chúng. Các bài **NP-khó** là một tập các bài toán quan trọng, nhưng chưa có lời giải đa thức nào được tìm ra<sup>1</sup>.

## 2.3 Ước lượng tính hiệu quả

Khi tính được độ phức tạp của thuật toán, ta có thể biết được thuật toán đó có đủ hiệu quả để giải quyết vấn đề không trước khi cài đặt nó. Trên thực tế, một chiếc máy vi tính ngày nay có thể thực hiện hàng trăm triệu thao tác trong một giây.

Ví dụ, giả sử giới hạn thời gian của bài toán nào đó là một giây và kích thước dữ liệu vào là  $n = 10^5$ . Nếu độ phức tạp thời gian là  $O(n^2)$ , thuật toán sẽ cần thực hiện khoảng  $(10^5)^2 = 10^{10}$  thao tác. Như vậy chương trình sẽ mất ít nhất vài chục giây để chạy xong, do đó thuật toán trên quá chậm để giải quyết bài toán.

Mặt khác, khi được cho kích thước dữ liệu vào, ta có thể thử *đoán* độ phức tạp thời gian cần đạt của thuật toán chuẩn. Bảng sau đây nêu một số độ phức tạp phổ biến tương ứng với từng kích thước dữ liệu vào, giả sử giới hạn thời gian là một giây.

dữ liệu vào	độ phức tạp
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ hoặc $O(n)$
$n$ rất lớn	$O(1)$ hoặc $O(\log n)$

<sup>1</sup>Một quyển sách về chủ đề này là quyển *Computers and Intractability: A Guide to the Theory of NP-Completeness* [28], của M. R. Garey và D. S. Johnson

Ví dụ, nếu kích thước dữ liệu vào là  $n = 10^5$ , ta có thể dự đoán độ phức tạp thời gian của thuật toán là  $O(n)$  hoặc  $O(n \log n)$ . Thông tin này giúp ta dễ dàng tìm lời giải hơn bởi vì ta có thể loại trừ các cách tiếp cận dẫn đến độ phức tạp lớn hơn.

Tuy nhiên, ta cần lưu ý rằng, độ phức tạp thời gian chỉ là một cách ước lượng độ hiệu quả, bởi vì nó bỏ qua *hằng số cài đặt*. Ví dụ, một thuật toán có độ phức tạp thời gian  $O(n)$  có thể thực hiện  $n/2$  hoặc  $5n$  thao tác. Điều này ảnh hưởng lớn đến thời gian chạy thực tế của thuật toán

## 2.4 Đoạn con có tổng lớn nhất

Thông thường, một bài toán có thể giải được bằng nhiều thuật toán với độ phức tạp thời gian khác nhau. Trong phần này, ta sẽ tìm hiểu một bài toán kinh điển, có hướng giải dễ hiểu trong  $O(n^3)$ . Tuy nhiên, sau khi thiết kế một thuật toán tốt hơn, ta có thể giải bài toán với độ phức tạp  $O(n^2)$  hay thậm chí  $O(n)$ .

Cho một dãy  $n$  số, nhiệm vụ của ta là tìm **đoạn con có tổng lớn nhất**, nghĩa là tìm tổng lớn nhất có thể của một đoạn các phần tử liên tiếp trong dãy<sup>2</sup>. Bài toán thú vị ở chỗ, phần tử trong dãy có thể là một số âm.

Ví dụ, trong dãy

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

đoạn con sau có tổng lớn nhất, giá trị là 10:

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

Giả sử đoạn con rỗng là hợp lệ, khi đó tổng lớn nhất của một đoạn con luôn ít nhất là 0.

### Thuật toán 1

Lời giải ta có thể nhìn ra ngay cho bài toán này, đó là duyệt qua tất cả các đoạn con trong dãy, tính tổng giá trị của mỗi đoạn và lưu lại tổng lớn nhất. Đoạn mã dưới đây mô tả cách cài đặt thuật toán:

```
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);
    }
}
```

<sup>2</sup>Sách của J. Bentley *Programming Pearls* [8] đã khiến bài toán nổi tiếng.

```

    }
}
cout << best << "\n";

```

Các biến  $a$  và  $b$  cố định vị trí đầu và cuối của đoạn con, và tổng của các phần tử được lưu bằng biến  $sum$ . Biến  $best$  duy trì tổng lớn nhất tìm được trong lúc duyệt.

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

## Thuật toán 2

Ta có thể dễ dàng cải tiến thuật toán 1 bằng cách giảm bớt một vòng lặp. Cụ thể, ta sẽ tính tổng ngay khi biên phải của đoạn con dịch chuyển. Đoạn mã sau mô tả cách làm:

```

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";

```

Như vậy, độ phức tạp thời gian chỉ còn  $O(n^2)$ .

## Thuật toán 3

Ngạc nhiên thay, ta cũng có thể giải bài toán trên với độ phức tạp  $O(n)^3$ , nghĩa là chỉ cần một vòng lặp là đủ. Ý tưởng chính là với từng vị trí trong dãy, ta đi tìm tổng lớn nhất của một đoạn con kết thúc tại vị trí đó. Sau đó, đáp án của bài toán sẽ là tổng lớn nhất trong số chúng.

Xét bài toán con tìm tổng lớn nhất của một đoạn con kết thúc tại vị trí  $k$ . Có hai trường hợp xảy ra:

1. Đoạn con chỉ gồm phần tử tại vị trí  $k$ .
2. Đoạn con bao gồm một đoạn nhỏ hơn kết thúc tại vị trí  $k - 1$ , theo sau đó là phần tử tại vị trí  $k$ .

Trong trường hợp thứ hai, do ta cần tìm đoạn con có tổng lớn nhất, đoạn con kết thúc tại vị trí  $k - 1$  cũng cần có tổng lớn nhất. Do đó, ta có thể giải quyết bài toán một cách tối ưu bằng cách tính tổng lớn nhất của các đoạn con kết thúc tại từng vị trí từ trái sang phải.

<sup>3</sup>Trong [8], thuật toán tuyến tính này được tìm ra bởi J. B. Kadane, và thuật toán cũng đôi khi được gọi là **thuật toán Kadane**.

Đoạn mã dưới đây mô tả cách cài đặt thuật toán:

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

Thuật toán chỉ cần một lần duyệt qua dữ liệu vào, do đó độ phức tạp thời gian là  $O(n)$ . Đây cũng là độ phức tạp tốt nhất có thể đạt được, vì bất kỳ thuật toán nào dùng để giải bài toán này đều cần duyệt qua mỗi phần tử trong dãy ít nhất một lần.

## So sánh tốc độ

Bây giờ ta kiểm tra thời gian chạy trên thực tế của các thuật toán. Bảng sau thể hiện thời gian chạy của các thuật toán trên với các giá trị khác nhau của  $n$  trên một máy tính hiện đại.

Trong mỗi bộ dữ liệu, đầu vào được sinh ngẫu nhiên. Ta bỏ qua thời gian đọc đầu vào của chương trình.

độ lớn của $n$	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

Bảng trên cho thấy mọi chương trình đều hiệu quả khi kích thước đầu vào nhỏ, nhưng những đầu vào lớn hơn cho thấy sự khác biệt đáng kể trong thời gian chạy giữa các thuật toán. Thuật toán 1 trở nên chậm khi  $n = 10^4$ , và thuật toán 2 cũng trở nên chậm khi  $n = 10^5$ . Chỉ có thuật toán 3 có thể xử lý được bộ dữ liệu lớn nhất một cách nhanh chóng.

# Chương 3

## Sắp xếp

**Sắp xếp** là một bài toán nền tảng trong thiết kế thuật toán. Rất nhiều thuật toán áp dụng sắp xếp là một bài toán nền tảng trong thiết kế thuật toán. Rất nhiều thuật toán áp dụng sắp xếp bởi dữ liệu sau khi được sắp xếp theo một trật tự cụ thể thì có thể được xử lý một cách dễ dàng hơn.

Ví dụ, bài toán "kiểm tra mảng có chứa hai phần tử trùng nhau không?" có thể giải được dễ dàng nhờ thuật toán sắp xếp. Nếu mảng có chứa hai phần tử trùng nhau, chúng chắc chắn sẽ nằm liền kề nhau sau khi sắp xếp, từ đó có thể dễ dàng được tìm ra. Tương tự, bài toán "tìm phần tử xuất hiện nhiều lần nhất trong một mảng?" cũng có thể giải bằng cách trên.

Có rất nhiều thuật toán sắp xếp, chúng là minh họa cụ thể cho việc áp dụng nhiều kỹ thuật thiết kế thuật toán khác nhau cho cùng một bài toán. Các thuật toán sắp xếp tổng quát và hiệu quả có độ phức tạp thời gian là  $O(n \log n)$ . Nhiều thuật toán ứng dụng sắp xếp cũng có độ phức tạp thời gian tương tự.

### 3.1 Lý thuyết sắp xếp

Bài toán cơ bản trong sắp xếp được phát biểu 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 mảng sau đây sau khi sắp xếp

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

## Thuật toán $O(n^2)$

Những thuật toán sắp xếp đơn giản đều chạy với độ phức tạp thời gian  $O(n^2)$ . Những thuật toán này đều ngắn gọn và thường gồm hai vòng lặp lồng nhau. Một thuật toán sắp xếp  $O(n^2)$  nổi tiếng là **sắp xếp nổi bọt**. Các phần tử sẽ "nổi lên" dựa vào giá trị của chúng.

Sắp xếp nổi bọt bao gồm  $n$  lượt. Ở mỗi lượt, thuật toán sẽ duyệt qua các phần tử của mảng. Mỗi khi tìm thấy hai phần tử liên tiếp không đúng trật tự, thuật toán sẽ hoán đổi vị trí của chúng. Thuật toán 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 lượt đầu tiên, phần tử lớn nhất sẽ được di chuyển tới vị trí đúng. Một cách tổng quát, sau  $k$  lượt,  $k$  phần tử lớn nhất sẽ được di chuyển tới đúng vị trí. Vì thế, sau  $n$  lượt, toàn bộ mảng sẽ đúng thứ tự.

Ví dụ, trong mảng

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

lượt 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
---	---	---	---	---	---	---	---



## Cặp nghịch thế

Sắp xếp nổi bọt là một ví dụ của các thuật toán sắp xếp mà luôn hoán đổi các phần tử *liên tiếp* trong mảng. Cần lưu ý rằng, độ phức tạp thời gian của

thuật toán trên *luôn luôn* tối thiểu là  $O(n^2)$ , vì trong trường hợp xấu nhất, ta sẽ cần tới  $O(n^2)$  lần hoán đổi để sắp xếp lại mảng.

Một khái niệm quan trọng liên quan tới các thuật toán sắp xếp là **cặp nghịch thế**: một cặp phần tử trong mảng ( $\text{array}[a], \text{array}[b]$ ) mà  $a < b$  và  $\text{array}[a] > \text{array}[b]$ ,  $a < b$  và  $\text{array}[a] > \text{array}[b]$ , hay nói cách khác là những cặp phần tử đứng sai vị trí. Ví dụ, mảng

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

có ba cặp nghịch thế: (6,3), (6,5) and (9,8). Mảng có càng nhiều cặp nghịch thế thì càng cần nhiều phép biến đổi để sắp xếp lại mảng hơn. Một mảng được sắp xếp tăng dần khi nó không còn một cặp nghịch thế nào. Ngược lại, số cặp nghịch thế sẽ là lớn nhất khi mảng được sắp xếp giảm dần.

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

Đổi chỗ một cặp hai phần tử liên tiếp đang sai thứ tự sẽ làm số cặp nghịch thế của mảng giảm đi 1. Vì vậy, nếu một thuật toán sắp xếp chỉ đổi chỗ hai phần tử liên tiếp và mỗi lần đổi chỗ xóa đi nhiều nhất là một cặp nghịch thế thì độ phức tạp thời gian của thuật toán sẽ ít nhất là  $O(n^2)$ .

## Các thuật toán $O(n \log n)$

Ta có thể sắp xếp mảng một cách hiệu quả với độ phức tạp thời gian là  $O(n \log n)$  bằng cách sử dụng những thuật toán không chỉ đơn thuần đổi chỗ hai phần tử liên tiếp. Một trong các thuật toán đó là **sắp xếp trộn**<sup>1</sup>, một thuật toán áp dụng phép đệ quy. Sắp xếp trộn sắp xếp một mảng con (đoạn con)  $\text{array}[a \dots b]$  như sau:

1. Nếu  $a = b$ , không làm gì cả, vì mảng con đã được sắp xếp.
2. Tìm vị trí của phần tử ở giữa:  $k = \lfloor (a + b)/2 \rfloor$ .
3. Gọi đệ quy để sắp xếp  $\text{array}[a \dots k]$ .
4. Gọi đệ quy để sắp xếp  $\text{array}[k + 1 \dots b]$ .
5. Trộn hai mảng con đã được sắp xếp  $\text{array}[a \dots k]$  và  $\text{array}[k + 1 \dots b]$  thành mảng con  $\text{array}[a \dots b]$  theo đúng thứ tự.

Sắp xếp trộn là một thuật toán hiệu quả, bởi nó chia đôi mảng con ở từng bước. Phép đệ quy sẽ gồm  $O(\log n)$  tầng, và xử lý từng tầng với độ phức tạp  $O(n)$ . Sở dĩ ta có thể trộn mảng con  $\text{array}[a \dots k]$  và  $\text{array}[k + 1 \dots b]$  trong độ phức tạp tuyến tính là bởi chúng đã được sắp xếp.

Ví dụ, ta sắp xếp mảng sau:

<sup>1</sup>Theo [47], sắp xếp trộn được giới thiệu lần đầu bởi J. von Neumann vào năm 1945.

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, sau khi gọi đệ quy, các mảng con sẽ được sắp xếp như sau:

1	2	3	6
2	5	8	9

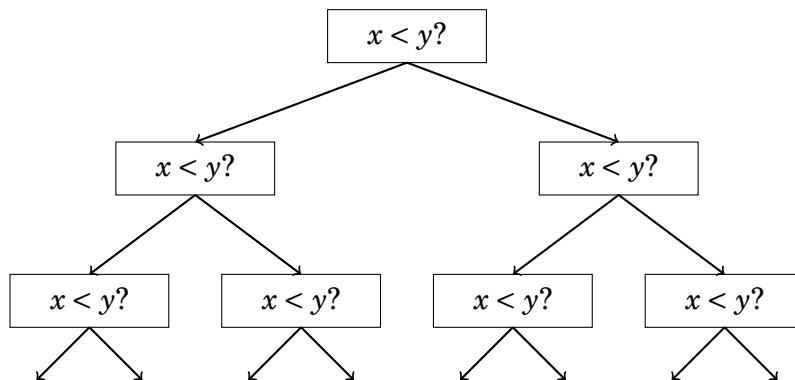
Cuối cùng, thuật toán sẽ trộn hai mảng con với nhau, tạo thành mảng được sắp xếp hoàn chỉnh:

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

## Cận dưới của sắp xếp

Liệu ta có thể sắp xếp mảng với độ phức tạp tốt hơn  $O(n \log n)$  không? Câu trả lời là *không* nếu ta chỉ xét các thuật toán sắp xếp dựa trên việc so sánh các phần tử của mảng.

Ta có thể chứng minh cận dưới của độ phức tạp thời gian bằng cách xem việc sắp xếp là một quá trình mà mỗi lần so sánh hai phần tử nào đấy sẽ cho ta biết thêm thông tin về mảng. Quá trình này sẽ tạo ra một cây như sau:



" $x < y$ ?" nghĩa là phần tử  $x$  và  $y$  đã được so sánh. Nếu  $x < y$ , ta sẽ đi về phía bên trái, ngược lại ta đi về bên phải. Khi đi theo các nhánh xuống cuối cùng (nút lá), ta sẽ có đủ thông tin để sắp xếp mảng. Kết quả của quá trình này, theo như minh họa ở cây trên, là tất cả các cách sắp xếp mảng khác nhau, tổng cộng  $n!$  cách (tương ứng với  $n!$  hoán vị của mảng ban đầu). Vì vậy, chiều cao của cây tối thiểu là

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

Ta tìm cận dưới của tổng trên bằng cách chọn  $n/2$  số hạng cuối và thay đổi giá trị của các số hạng thành  $\log_2(n/2)$ . Từ đó, ta có bất đẳng thức

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

vì vậy chiều cao của cây, hay số bước tối thiểu của một thuật toán sắp xếp trong trường hợp xấu nhất sẽ tối thiểu  $n \log n$ .



## Sắp xếp đếm phân phối

Cận dưới  $n \log n$  sẽ không đúng đối với các thuật toán sắp xếp không phụ thuộc vào so sánh các phần tử của mảng mà sử dụng những thông tin khác. Một ví dụ chính là thuật toán **sắp xếp đếm phân phối**. Thuật toán này sẽ sắp xếp mảng trong thời gian  $O(n)$  nếu tất cả phần tử trong mảng là số nguyên từ  $0 \dots c$  và  $c = O(n)$ .

Thuật toán sẽ tạo một *mảng đếm*, có chỉ số của nó là giá trị của các phần tử trong mảng gốc. Thuật toán sẽ duyệt qua mảng gốc và đếm xem mỗi giá trị sẽ xuất hiện bao nhiêu lần.

Ví dụ, mảng sau:

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

sẽ có mảng đếm như sau:

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

Ví dụ, giá trị ở vị trí 3 trong mảng đếm là 2, vì giá trị 3 đã xuất hiện 2 lần trong mảng gốc.

Việc xây dựng mảng đếm được thực hiện trong thời gian  $O(n)$ . Sau đó, ta có thể sắp xếp mảng trong  $O(n)$ , vì số lần xuất hiện của mỗi phần tử có thể lấy ra được từ mảng đếm. Vì vậy, 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 dùng được khi hằng số  $c$  đủ bé, khi đó giá trị của các phần tử trong mảng mới có thể dùng làm chỉ số trong mảng đếm.

## 3.2 Sắp xếp trong C++

Chúng ta không nên tự cài đặt một thuật toán sắp xếp trong một kì thi, vì sẽ có rất nhiều cách cài đặt tốt có sẵn trong các ngôn ngữ lập trình. Ví dụ, thư viện chuẩn của C++ có hàm `sort` có thể sắp xếp mảng và các cấu trúc dữ liệu khác rất thuận tiện.

Có nhiều lợi ích khi dùng hàm của thư viện. Thứ nhất là tiết kiệm thời gian vì ta không cần cài đặt hàm. Thứ hai, cài đặt của thư viện chắc chắn đúng và hiệu quả, chưa chắc một hàm sắp xếp tự cài đã tốt hơn.

Trong phần này chúng ta sẽ xem cách sử dụng hàm `sort` của C++. Đoạn mã sau sẽ sắp xếp một vector tăng dần:

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

Sau khi sắp xếp, vector của chúng ta sẽ trông như sau: `[2,3,3,4,5,5,8]`. Thứ tự mặc định khi sắp xếp là tăng dần, nhưng ta cũng có thể sắp xếp giảm dần 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 sẽ sắp xếp xâu s:

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

Sắp xếp một xâu tức là sắp xếp các ký tự của xâu đó. Ví dụ, xâu "monkey" sẽ được sắp xếp thành "ekmnoy".

## Toán tử so sánh

Để sử dụng hàm sort, ta cần định nghĩa một **toán tử so sánh** cho kiểu dữ liệu của các phần tử được sắp xếp. Khi sắp xếp, toán tử này sẽ được sử dụng khi cần so sánh hai phần tử nào đấy.

Phần lớn kiểu dữ liệu của C++ đều có toán tử so sánh được xây dựng sẵn, và các phần tử của những kiểu dữ liệu này có thể được sắp xếp một cách tự động. Ví dụ, kiểu số nguyên sẽ được sắp xếp dựa trên giá trị của chúng và các xâu được sắp xếp theo thứ tự từ điển.

Kiểu dữ liệu pair khi được so sánh sẽ ưu tiên giá trị first trước. Tuy nhiên, nếu first của hai pair là bằng nhau, chúng sẽ được so sánh dựa vào giá trị 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());
```

Thứ tự của các pair sau khi sắp xếp là (1,2), (1,5) and (2,3).

Tương tự, các tuple được sắp xếp ưu tiên giá trị thứ nhất, sau đó đến giá trị thứ hai...<sup>2</sup>

```
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 khi sắp xếp, thứ tự các tuple sẽ là (1,5,3), (2,1,3) and (2,1,4).

## Kiểu dữ liệu do người dùng định nghĩa

Các kiểu dữ liệu do người dùng định nghĩa không có sẵn toán tử so sánh mà cần được định nghĩa bên trong kiểu dữ liệu như một hàm có tên operator<, có tham số là một phần tử khác cùng kiểu dữ liệu. Toán tử này cần trả true nếu phần tử nhỏ hơn tham số, nếu không, trả về false.

<sup>2</sup>Lưu ý rằng ở một số trình biên dịch cũ, phải dùng hàm make\_tuple để tạo một tuple thay vì dấu ngoặc nhọn (ví dụ, make\_tuple(2,1,4) thay vì {2,1,4}).

Ví dụ, kiểu dữ liệu P sau chứa tọa độ x và y của một điểm. Toán tử so sánh được định nghĩa sao cho các điểm được sắp xếp ưu tiên theo tọa độ x trước sau đó tới 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

Ta còn có thể truyền một **hàm so sánh** bên ngoài vào trong hàm sort như một tham số. Ví dụ, hàm so sánh comp sau đây sắp xếp các xâu ưu tiên theo độ dài trước, sau đó tới thứ tự từ điển:

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

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

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

## 3.3 Tìm kiếm nhị phân

Phương pháp tổng quát để tìm một phần tử trong một mảng là dùng một vòng for duyệt qua tất cả các phần tử của mảng. Ví dụ, đoạn mã sau sẽ tìm vị trí của giá trị  $x$  trong một mảng:

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

Độ phức tạp thời gian của cách này là  $O(n)$ , vì trong trường hợp xấu nhất, cần phải duyệt qua tất cả phần tử của mảng. Nếu các phần tử không được sắp xếp theo thứ tự cụ thể nào, đây chính là cách làm tốt nhất, vì không có bất kỳ thông tin nào cho ta biết nên tìm  $x$  ở đâu trong mảng.

Tuy nhiên, nếu mảng được *sắp xếp* thì câu chuyện sẽ hoàn toàn khác. Trong trường hợp này, ta có thể tìm kiếm nhanh hơn rất nhiều, vì thứ tự

của các phần tử trong mảng cho ta thêm thông tin khi tìm kiếm. Thuật toán **tìm kiếm nhị phân** sau đây có thể tìm nhanh vị trí của một giá trị trong một mảng đã sắp xếp trong thời gian  $O(\log n)$ .

## Cách 1

Cách cài đặt thông thường của tìm kiếm nhị phân mô phỏng lại cách tìm một từ trong từ điển. Thuật toán sẽ duy trì một "phạm vi" trong mảng. Phạm vi này ban đầu bao gồm toàn bộ mảng. Tiếp đến, với mỗi bước, ta sẽ chia đôi phạm vi tìm kiếm này.

Ở mỗi bước, thuật toán sẽ xét phần tử ở giữa của phạm vi đang xét. Nếu phần tử ở giữa là giá trị cần tìm, ta kết thúc thuật toán. Ngược lại, thuật toán sẽ tiếp tục xét ở nửa trái hoặc nửa phải của phạm vi, tùy theo giá trị của phần tử ở giữa. Quá trình trên sẽ lặp lại với phạm vi nhỏ hơn này, cứ như thế cho tới khi tìm thấy, hoặc không thể chia đôi được nữa.

Ý tưởng trên có thể 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 ở vị trí k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Trong cách cài đặt này, phạm vi đang xét là  $a \dots b$ , và ban đầu phạm vi là  $0 \dots n-1$ . Thuật toán sẽ thu nhỏ phạm vi đi một nửa ở từng bước, nên độ phức tạp thời gian là  $O(\log n)$ .

## Cách 2

Một cách cài đặt khác của tìm kiếm nhị phân là duyệt qua các phần tử của mảng một cách hiệu quả. Ý tưởng ở đây là ta sẽ "nhảy" trên mảng và giảm dần tốc độ khi ta tiến tới gần vị trí cần tìm.

Thuật toán sẽ duyệt qua mảng từ trái sang phải, với độ dài bước nhảy ban đầu là  $n/2$ . Ở mỗi bước, độ dài sẽ giảm đi một nửa: đầu tiên là  $n/4$ , sau đó  $n/8$ ,  $n/16$ , vv. Thuật toán sẽ dừng khi độ dài trở về 1.

Sau khi nhảy, hoặc là mục tiêu được tìm thấy hoặc ta biết rằng nó không có trong mảng.

Cài đặt thuật toán trên như sau:

```
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 o vi tri k
}
```

Khi thuật toán hoạt động, biến  $b$  sẽ lưu độ lớn của bước nhảy hiện tại. Độ phức tạp thời gian của thuật toán là  $O(\log n)$ , vì đoạn mã trong vòng `while` được thực hiện tối đa hai lần với mỗi độ dài bước nhảy khác nhau.

## Các hàm trong C++

Thư viện chuẩn của C++ chứa nhiều hàm hoạt động dựa trên tìm kiếm nhị phân và cũng có độ phức tạp thời gian  $O(\log n)$ .

- `lower_bound` trả về một con trỏ, trỏ đến phần tử đầu tiên của mảng có giá trị không nhỏ hơn  $x$ .
- `upper_bound` trả về một con trỏ, trỏ đến phần tử đầu tiên của mảng có giá trị lớn hơn  $x$ .
- `equal_range` trả về cả hai con trỏ nêu trên.

Các hàm này sẽ giả sử rằng mảng đã được sắp xếp. Nếu như không tồn tại phần tử nào thỏa mãn, con trỏ sẽ trỏ đến phần tử ngay sau phần tử cuối cùng. Ví dụ, đoạn mã sau sẽ kiểm tra xem mảng có tồn tại phần tử giá trị  $x$  không.

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

Đoạn mã sau đây sẽ 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";
```

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

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

## Tìm nghiệm nhỏ nhất

Một ứng dụng quan trọng của tìm kiếm nhị phân là tìm vị trí mà tại đó một hàm thay đổi giá trị. Giả sử ta cần tìm giá trị  $k$  nhỏ nhất là một nghiệm hợp lệ cho một bài toán. Ta có hàm  $ok(x)$  sẽ trả về true nếu  $x$  là nghiệm hợp lệ và ngược lại trả về false. Ngoài ra, ta biết rằng  $ok(x)$  là false khi  $x < k$  và true khi  $x \geq k$ . Tình huống trên được mô tả bằng hình sau: Tình huống trên được mô tả bằng hình sau:

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

Ta có thể tìm sử dụng tìm kiếm nhị phân để tìm ra  $k$  như sau:

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

Thuật toán sẽ tìm giá trị  $x$  lớn nhất mà  $ok(x)$  là false. Khi đó, giá trị tiếp theo  $k = x + 1$  sẽ là giá trị nhỏ nhất mà  $ok(k)$  là true. Độ dài  $z$  của bước nhảy đầu tiên phải đủ lớn, lấy ví dụ như một giá trị nào đấy mà ta biết chắc  $ok(z)$  là true.

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

## Tìm giá trị lớn nhất

Ta cũng có thể sử dụng tìm kiếm nhị phân để tìm giá trị lớn nhất của một hàm mà tăng dần ban đầu và sau đó giảm dần. Ta cần tìm một vị trí  $k$  thỏa mãn

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

Ta sẽ áp dụng tìm kiếm nhị phân để tìm giá trị  $x$  lớn nhất mà  $f(x) < f(x+1)$ . Từ đây suy ra  $k = x + 1$  vì  $f(x+1) > f(x+2)$ .

Đoạn mã sau đây thực hiện 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 tìm kiếm nhị phân thông thường, thuật toán trên sẽ không thể hoạt động nếu các giá trị liên tiếp của hàm bằng nhau. Vì trong trường hợp đó ta không có cơ sở để tiếp tục tìm kiếm (nên xét nửa bên trái hay bên phải?)



# Chương 4

## Cấu trúc dữ liệu

Một **cấu trúc dữ liệu** là một cách để lưu trữ thông tin trong bộ nhớ máy tính. Chọn cấu trúc dữ liệu hợp lý cho một bài toán là điều rất quan trọng, vì mỗi cấu trúc đều có những ưu điểm và nhược điểm riêng. Khi giải bài, câu hỏi quan trọng nhất cần đặt ra là: những thao tác nào cần được tối ưu trong cấu trúc dữ liệu mà ta sẽ chọn?

Chương này sẽ giới thiệu một số cấu trúc dữ liệu quan trọng nhất trong Thư viện chuẩn (STL) của C++. Chúng ta nên tận dụng thư viện chuẩn nhiều nhất có thể, vì nó sẽ giúp ta tiết kiệm rất nhiều thời gian. Phần sau của quyển sách này sẽ nói về những cấu trúc dữ liệu phức tạp hơn không có trong thư viện chuẩn.

### 4.1 Mảng động

Một **mảng động** là một mảng mà độ dài của nó có thể thay đổi được trong lúc chạy chương trình. Cấu trúc mảng động phổ biến nhất trong C++ là kiểu dữ liệu vector, và ta có thể sử dụng gần giống như một kiểu dữ liệu mảng bình thường.

Đoạn mã dưới đây sẽ tạo ra một vector rỗng và thêm ba phần tử vào đó:

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

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

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

Hàm size trả về số phần tử trong vector. Đoạn mã dưới đây sẽ duyệt qua vector và in ra từng phần tử trong đó:

```

1 for (int i = 0; i < v.size(); i++) {
2     cout << v[i] << "\n";
3 }

```

Dưới đây là một cách ngắn hơn để duyệt qua một vector:

```

1 for (auto x : v) {
2     cout << x << "\n";
3 }

```

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

```

1 vector<int> v;
2 v.push_back(5);
3 v.push_back(2);
4 cout << v.back() << "\n"; // 2
5 v.pop_back();
6 cout << v.back() << "\n"; // 5

```

Đoạn mã sau đây tạo một vector với năm phần tử:

```

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

```

Một cách khác dùng để tạo mảng là cho sẵn số lượng phần tử, và giá trị ban đầu của mỗi phần tử:

```

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

```

```

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

```

Phần cài đặt bên trong của một vector thực ra sử dụng một mảng thường. Nếu độ dài của vector tăng và mảng đó không đủ chỗ chứa phần tử, một mảng mới sẽ được tạo để dời toàn bộ phần tử từ mảng cũ sang. Tuy nhiên, điều này không thường xuyên xảy ra, nên độ phức tạp trung bình của hàm `push_back` là  $\mathcal{O}(1)$ .

Cấu trúc chuỗi `string` cũng là một mảng động tương tự một vector. Ngoài ra, có một số cú pháp đặc biệt của chuỗi mà không hề tồn tại trong một số cấu trúc dữ liệu khác. Các chuỗi có thể được ghép nối bằng phép toán `+`. Hàm `substr(k, x)` trả về chuỗi con bắt đầu từ vị trí  $k$  có độ dài  $x$ , và hàm `find(t)` tìm vị trí xuất hiện đầu tiên của một chuỗi con  $t$ .

Đoạn mã sau đây minh họa một số thao tác trên chuỗi:

```

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

```

## 4.2 Cấu trúc tập hợp

Một **tập hợp** là một cấu trúc dữ liệu duy trì một tập các phần tử có giá trị khác nhau. Các thao tác cơ bản trên tập hợp gồm: chèn, tìm kiếm, và loại bỏ.

Thư viện chuẩn của C++ cung cấp hai cấu trúc tập hợp với cách cài đặt khác nhau: Cấu trúc set dựa trên một cây nhị phân cân bằng, và các thao tác được thực hiện trong thời gian  $\mathcal{O}(\log n)$ . Cấu trúc unordered\_set sử dụng hàm băm, và các thao tác chạy trong thời gian trung bình là  $\mathcal{O}(1)$ .

Việc lựa chọn kiểu cài đặt nào của cấu trúc tập hợp thường tùy theo ý thích của mỗi người. Điểm mạnh của set là nó duy trì thứ tự của các phần tử và cung cấp những hàm mà không có trong unordered\_set. Mặt khác, unordered\_set có thể hiệu quả hơn trong nhiều trường hợp.

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

```

1 set<int> s;
2 s.insert(3);
3 s.insert(2);
4 s.insert(5);
5 cout << s.count(3) << "\n"; // 1
6 cout << s.count(4) << "\n"; // 0
7 s.erase(3);
8 s.insert(4);
9 cout << s.count(3) << "\n"; // 0
10 cout << s.count(4) << "\n"; // 1

```

Một tập hợp có thể được sử dụng gần giống như một vector, nhưng không thể truy cập vào các phần tử sử dụng ký hiệu []. Đoạn mã dưới đây tạo ra một tập hợp, in ra số lượng phần tử trong tập, và duyệt qua từng phần tử của nó.

```

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

```

```

3  for (auto x : s) {
4      cout << x << "\n";
5  }

```

Một tính chất quan trọng của tập hợp đó là tất cả các phần tử đều *khác nhau*. Do đó, hàm `count` luôn trả về một trong hai giá trị: 0 (phần tử không có trong tập) hoặc 1 (phần tử có trong tập), và hàm `insert` không bao giờ thêm phần tử vào tập hợp nếu nó đã xuất hiện rồi. Đoạn mã dưới đây sẽ minh họa điều này:

```

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

```

C++ cũng hỗ trợ cấu trúc `multiset` (đa tập) và `unordered_multiset` hoạt động giống như `set` và `unordered_set` nhưng có thể chứa nhiều phần tử cùng một giá trị. Ví dụ như đoạn mã dưới đây, cả ba lần xuất hiện của phần tử 5 đều được thêm vào `multiset`:

```

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

```

Hàm `erase` loại bỏ toàn bộ những lần xuất hiện của một giá trị trong một `multiset`:

```

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

```

Đôi khi, chúng ta chỉ muốn loại bỏ duy nhất một phần tử có giá trị đó, khi đó chúng ta có thể làm như sau:

```

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

```

## 4.3 Cấu trúc ánh xạ

Một **ánh xạ** là một kiểu dữ liệu được tổng quát hóa từ mảng, bao gồm các cặp khóa - giá trị. Trong khi khóa trong một mảng bình thường là các số nguyên liên tiếp  $0, 1, 2, 3, \dots, n-1$ , với  $n$  là độ dài của mảng, thì các khóa của

một ánh xạ có thể thuộc bất kỳ kiểu dữ liệu nào và cũng không nhất thiết phải là các giá trị liên tiếp nhau.

Thư viện chuẩn C++ bao gồm hai cách cài đặt của kiểu dữ liệu ánh xạ, tương ứng với hai kiểu của tập hợp: cấu trúc `map` dựa trên cây nhị phân cân bằng và việc truy cập phần tử tốn thời gian  $O(\log n)$ , trong khi cấu trúc `unordered_map` thì sử dụng hàm băm và trung bình tốn  $O(1)$  để truy cập phần tử.

Đoạn mã dưới đây tạo ra một ánh xạ, với các khóa thuộc kiểu dữ liệu `string`, và các giá trị là các số nguyên.

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

Nếu bạn truy cập vào một giá trị khóa mà không có trong ánh xạ, thì khóa đó sẽ được tự động thêm vào ánh xạ với một giá trị mặc định. Ví dụ, ở đoạn mã dưới đây, khóa “aybaltu” với giá trị 0 được thêm vào ánh xạ.

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

Hàm `count` kiểm tra xem liệu một khóa có tồn tại trong ánh xạ không:

```
1 if (m.count("aybaltu")) {  
2     // khóa có tồn tại  
3 }
```

Đoạn mã dưới đây in ra toàn bộ các khóa và giá trị của một ánh xạ:

```
1 for (auto x : m) {  
2     cout << x.first << " " << x.second << "\n";  
3 }
```

## 4.4 Con trỏ lặp và làm việc trên dãy

Rất nhiều hàm trong thư viện chuẩn của C++ làm việc với con trỏ lặp (iterator)<sup>1</sup>. Một **con trỏ lặp** là một biến dùng để chỉ vào một phần tử trong một cấu trúc dữ liệu.

Một trong những trỏ lặp (iterator) thường dùng là `begin` và `end` định ra một dãy chứa hết tất cả các phần tử trong một cấu trúc dữ liệu. Con trỏ

---

<sup>1</sup>Cần phân biệt giữa iterator và con trỏ thường (như `int*` a), dù cách sử dụng khá tương đồng

`begin` chỉ vào phần tử đầu tiên của cấu trúc dữ liệu, và con trỏ `end` chỉ vào vị trí *ngay sau* phần tử cuối cùng.

Chúng ta có thể nhìn hình minh họa dưới đây:

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

Hãy để ý vào sự bất tương xứng của hai trỏ lặp này: `s.begin()` chỉ vào một phần tử trong cấu trúc dữ liệu, trong khi `s.end()` lại chỉ ra ngoài cấu trúc dữ liệu. Vì thế, một dãy các phần tử liên tiếp được biểu diễn bởi các trỏ lặp tương đồng với khái niệm *nửa khoảng* trong lý thuyết tập hợp.

## Làm việc với đoạn con

Trỏ lặp được dùng trong các hàm thư viện chuẩn C++ mà có đầu vào gồm một dãy các phần tử trong một cấu trúc dữ liệu. Thông thường, chúng ta cần xử lý tất cả phần tử trong một cấu trúc, vì vậy hai con trỏ `begin` và `end` được truyền cho những hàm này.

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

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

Những hàm này cũng có thể được dùng với một mảng bình thường. Trong trường hợp này, chúng ta sử dụng con trỏ thông thường thay vì dùng trỏ lặp.

```

1 sort(a, a+n);
2 reverse(a, a+n);
3 random_shuffle(a, a+n);

```

## Trở lặp trong tập hợp

Trở lặp thường được sử dụng để truy cập vào phần tử của một tập hợp. Đoạn mã dưới đây tạo ra một con trỏ `it` chỉ vào phần tử nhỏ nhất của một tập:

```

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

```

Một cách ngắn hơn để viết đoạn mã trên là:

```

1 auto it = s.begin();

```

Phần tử mà một trở lặp chỉ tới có thể được truy cập bằng ký hiệu `*`. Ví dụ, đoạn mã dưới đây in ra phần tử đầu tiên trong tập:

```

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

```

Trở lặp có thể được di chuyển sử dụng toán tử `++` (tiến) và `--` (lùi), cho biết con trỏ được di chuyển sang phần tử phần tử tiếp theo hay liền trước trong tập.

Đoạn mã sau đây in ra toàn bộ các phần tử theo thứ tự tăng dần:

```

1 for (auto it = s.begin(); it != s.end(); it++) {
2     cout << *it << "\n";
3 }

```

Đoạn mã dưới đây in ra phần tử lớn nhất trong tập hợp:

```

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

```

Hàm `find(x)` trả về một con trỏ lặp trỏ vào một phần tử có giá trị là `x`. Tuy nhiên, nếu trong tập hợp không chứa `x`, con trỏ được trả về sẽ là `end`.

```

1 auto it = s.find(x);
2 if (it == s.end()) {
3     // x is not found
4 }

```

Hàm `lower_bound(x)` trả về một trỏ lặp trỏ vào phần tử nhỏ nhất trong tập hợp có giá trị *lớn hơn hoặc bằng*  $x$ , và hàm `upper_bound(x)` trả về một trỏ lặp trỏ vào phần tử nhỏ nhất trong tập có giá trị *lớn hơn*  $x$ . Ở trong cả hai hàm này, nếu không có phần tử nào thỏa mãn điều kiện tương ứng, giá trị được trả về là `end`. Những hàm này không được hỗ trợ trong cấu trúc dữ liệu `unordered_set` vì cấu trúc này không duy trì thứ tự của các phần tử.

Ví dụ, đoạn mã sau đây tìm phần tử có giá trị gần với  $x$  nhất:

```
1 auto it = s.lower_bound(x);
2 if (it == s.begin()) {
3     cout << *it << "\n";
4 } else if (it == s.end()) {
5     it--;
6     cout << *it << "\n";
7 } else {
8     int a = *it; it--;
9     int b = *it;
10    if (x-b < a-x) cout << b << "\n";
11    else cout << a << "\n";
12 }
```

Đoạn mã trên đang giả sử tập hợp không rỗng, và duyệt qua toàn bộ các trường hợp sử dụng một trỏ lặp tên là `it`. Đầu tiên, con trỏ trỏ vào phần tử nhỏ nhất có giá trị ít nhất bằng  $x$ . Nếu `it` bằng `begin`, thì phần tử này chính là phần tử gần  $x$  nhất. Nếu `it` bằng `end`, phần tử lớn nhất chính là phần tử gần  $x$  nhất. Nếu không rơi vào cả hai trường hợp trên, thì phần tử gần  $x$  nhất chính là phần tử mà `it` trỏ vào, hoặc là phần tử ngay phía trước.

## 4.5 Một số cấu trúc dữ liệu khác

### Dãy bit

Một **dãy bit (bitset)** là một mảng mà giá trị của mỗi phần tử là 0 hoặc 1. Ví dụ, đoạn mã dưới đây tạo ra một dãy bit chứa 10 phần tử:

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

Lợi ích của việc dùng dãy bit là nó sử dụng ít bộ nhớ hơn mảng bình thường, vì mỗi phần tử trong một `bitset` chỉ sử dụng một bit bộ nhớ. Ví dụ,



nếu  $n$  bit được lưu trong một mảng `int`, thì cần phải sử dụng  $32n$  bit trong bộ nhớ, nhưng một `bitset` chỉ sử dụng  $n$  bit. Hơn nữa, giá trị của một `bitset` có thể được tính toán hiệu quả thông qua các phép toán trên bit, khiến cho việc tối ưu thuật toán bằng dãy bit trở nên khả thi.

Đoạn mã dưới đây cho ta thấy một cách khác để tạo ra dãy bit ở phía trên:

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

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

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

Đoạn mã dưới đây cho ta thấy một số ví dụ của các phép toán trên bit:

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

## Hàng đợi hai đầu

Một **hàng đợi hai đầu (deque)** là một mảng động với độ dài có thể được thay đổi linh động ở cả hai đầu mảng. Giống như `vector`, `deque` cũng cung cấp hai hàm `push_back` và `pop_back`, nhưng ngoài ra có thêm hàm `push_front` và `pop_front` không có trong `vector`.

Một hàng đợi hai đầu có thể được sử dụng giống như sau:

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

Cách cài đặt bên trong một `deque` phức tạp hơn của `vector`, vì vậy nên một `deque` sẽ chậm hơn một `vector`. Tuy nhiên, việc thêm hoặc bớt phần tử ở hai đầu đều tốn thời gian trung bình  $\mathcal{O}(1)$ .

## Ngăn xếp

**Ngăn xếp (stack)** là một cấu trúc cung cấp hai phép toán sau với thời gian  $\mathcal{O}(1)$ : thêm một phần tử hoặc loại bỏ một phần tử ở đầu ngăn xếp. Chúng ta cũng chỉ được truy cập phần tử ở đầu ngăn xếp.

Đoạn mã dưới đây cho ta thấy cách dùng của một ngăn xếp

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

## Hàng đợi

Một **hàng đợi (queue)** cũng cung cấp hai phép toán sau trong thời gian  $\mathcal{O}(1)$ : thêm một phần tử vào cuối hàng đợi, và loại bỏ đi phần tử ở đầu hàng đợi. Ta chỉ có thể truy cập vào phần tử đầu và cuối của một hàng đợi.

Đoạn mã dưới đây cho ta thấy cách ta có thể sử dụng một hàng đợi:

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

## Hàng đợi ưu tiên

Một **hàng đợi ưu tiên** duy trì một tập các phần tử. Các phép toán được hỗ trợ bao gồm chèn, và tùy loại hàng đợi mà ta có thể lấy ra và xóa đi phần tử lớn nhất hay nhỏ nhất. Việc thêm hoặc xóa giá trị tốn thời gian  $\mathcal{O}(\log n)$ , và lấy giá trị tốn thời gian  $\mathcal{O}(1)$ .

Dù một tập hợp có thứ tự cung cấp mọi phép toán của một hàng đợi ưu tiên một cách hiệu quả, lợi ích của việc sử dụng hàng đợi ưu tiên là cấu trúc này có hằng số trong độ phức tạp nhỏ hơn. Hàng đợi ưu tiên thường được cài đặt sử dụng cấu trúc dữ liệu Đống (Heap), là một cấu trúc đơn giản hơn nhiều so với cây nhị phân cân bằng có trong tập hợp có thứ tự.

Theo mặc định, các phần tử trong một hàng đợi ưu tiên của C++ được sắp xếp theo thứ tự giảm dần, và ta có thể tìm và xóa đi giá trị lớn nhất trong hàng đợi. Đoạn mã dưới đây minh họa điều này:

```
1 priority_queue<int> q;  
2 q.push(3);  
3 q.push(5);  
4 q.push(7);  
5 q.push(2);  
6 cout << q.top() << "\n"; // 7  
7 q.pop();  
8 cout << q.top() << "\n"; // 5  
9 q.pop();  
10 q.push(6);  
11 cout << q.top() << "\n"; // 6  
12 q.pop();
```

Nếu ta muốn tạo ra một hàng đợi ưu tiên hỗ trợ việc tìm và xóa phần tử nhỏ nhất, ta có thể làm như sau:

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

## Cấu trúc dữ liệu tùy biến (Policy-based data structures – PBDS)

Trình biên dịch g++ hỗ trợ một số cấu trúc dữ liệu không thuộc vào thư viện chuẩn của C++. Những cấu trúc này được gọi là các cấu trúc *policy-based*. Để sử dụng chúng, những dòng sau cần được thêm vào mã nguồn:

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

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

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

Giờ thì ta có thể tạo ra một tập hợp như sau:

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

Điều đặc biệt của tập này đó là chúng ta có thể truy cập bằng chỉ số vào các phần tử như một mảng đã được sắp xếp. Hàm `find_by_order` trả về một trỏ lặp trỏ vào phần tử của một vị trí cho trước.

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

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

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

Nếu phần tử không có ở trong tập, chúng ta sẽ được trả về vị trí mà nó đáng ra sẽ đứng ở trong tập:

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

Cả hai hàm này đều hoạt động trong độ phức tạp thời gian logarit.

## 4.6 So sánh với sắp xếp

Chúng ta thường có thể giải một bài toán bằng cách sử dụng các cấu trúc dữ liệu hoặc sắp xếp. Đôi khi, có những khác biệt đáng lưu ý về hiệu quả thực sự của hai cách tiếp cận này, vốn có thể bị ẩn giấu đằng sau độ phức tạp thời gian của chúng.

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

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

đáp án là 3 bởi vì ba số 2, 5 và 9 đều thuộc cả hai danh sách trên.

Một thuật toán đơn giản nhất của bài toán này là duyệt qua toàn bộ các cặp phần tử trong thời gian  $\mathcal{O}(n^2)$ , nhưng tiếp theo ta sẽ tập trung vào các thuật toán hiệu quả hơn.

### Thuật toán 1

Ta xây dựng một tập hợp các phần tử xuất hiện trong  $A$ , và sau đó duyệt qua từng phần tử trong  $B$ , rồi kiểm tra xem nó có xuất hiện trong  $A$  hay không. Thuật toán này hiệu quả vì các phần tử của  $A$  được nằm trong một tập hợp. Sử dụng cấu trúc set, độ phức tạp thời gian của thuật toán này là  $\mathcal{O}(n \log n)$ .

## Thuật toán 2

Chúng ta không cần phải duy trì một tập hợp có thứ tự, nên thay vì sử dụng cấu trúc set thì ta có thể sử dụng cấu trúc unordered\_set. Đây là một cách khá dễ để khiến cho thuật toán hiệu quả hơn, vì chúng ta cần thay đổi cấu trúc dữ liệu đằng sau thuật toán. Độ phức tạp thời gian giờ sẽ là  $\mathcal{O}(n)$ .

## Thuật toán 3

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

## So sánh độ hiệu quả

Bảng sau đây cho thấy sự hiệu quả của những thuật toán trên khi  $n$  thay đổi, và phần tử của hai danh sách là các số nguyên được chọn 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 như nhau trừ sự khác biệt về cấu trúc dữ liệu được sử dụng. Trong bài toán này, lựa chọn cấu trúc ảnh hưởng rõ rệt lên thời gian chạy, bởi vì Thuật toán 2 nhanh hơn khoảng 4-5 lần so với Thuật toán 1.

Tuy nhiên, thuật toán hiệu quả nhất lại là Thuật toán 3 chỉ dùng sắp xếp. Nó chỉ dùng một nửa thời gian so với Thuật toán 2. Điều thú vị là độ phức tạp thời gian của cả hai Thuật toán 1 và 3 đều là  $\mathcal{O}(n \log n)$ , mặc dù vậy Thuật toán 3 lại chạy nhanh hơn gấp 10 lần. Điều này có thể giải thích bằng sự thật rằng sắp xếp là một thuật toán đơn giản, và chỉ thực hiện đúng một lần vào đầu thuật toán 3, phần còn lại của thuật toán hoạt động trong thời gian tuyến tính. Ngược lại, Thuật toán 1 duy trì một cây nhị phân cân bằng phức tạp trong suốt quá trình chạy thuật toán.



# Chương 5

## Duyệt toàn bộ

**Duyệt toàn bộ** là một kĩ thuật cơ bản có thể được sử dụng để giải gần như toàn bộ bài toán lập trình. Ý tưởng là sinh ra toàn bộ phương án có thể bằng duyệt trâu và chọn ra phương án tốt nhất hoặc đếm số lượng phương án, tùy thuộc vào bài toán.

Duyệt toàn bộ là một kĩ thuật tốt nếu như có đủ giới hạn thời gian để duyệt quá hết các phương án, bởi vì duyệt thì rất dễ để cài đặt và nó luôn đem lại đáp án chính xác. Nếu như duyệt toàn bộ bị quá thời gian, các kĩ thuật khác, ví dụ như tham lam hoặc quy hoạch động, sẽ cần được sử dụng.

### 5.1 Sinh ra các tập hợp con

Đầu tiên, chúng ta hãy xét bài toán sinh ra tất cả tập con của một tập hợp gồm  $n$  phần tử. Ví dụ, Các tập hợp con của  $\{0, 1, 2\}$  là  $\emptyset$ ,  $\{0\}$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{0, 1\}$ ,  $\{0, 2\}$ ,  $\{1, 2\}$  and  $\{0, 1, 2\}$ . Có hai cách thông thường để sinh ra các tập hợp con: chúng ta có thể duyệt đệ quy hoặc là sử dụng biểu diễn nhị phân của các số nguyên.

#### Phương pháp 1

Một cách để duyệt toàn bộ tập hợp con của một tập hợp chính là sử dụng đệ quy. Hàm search dưới đây sinh ra toàn bộ tập hợp con của một tập hợp  $\{0, 1, \dots, n-1\}$ . Hàm này lưu trữ một vector subset và nó sẽ lưu những phần tử của mỗi tập hợp con. Hàm duyệt sẽ bắt đầu khi hàm được gọi với tham số 0.

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

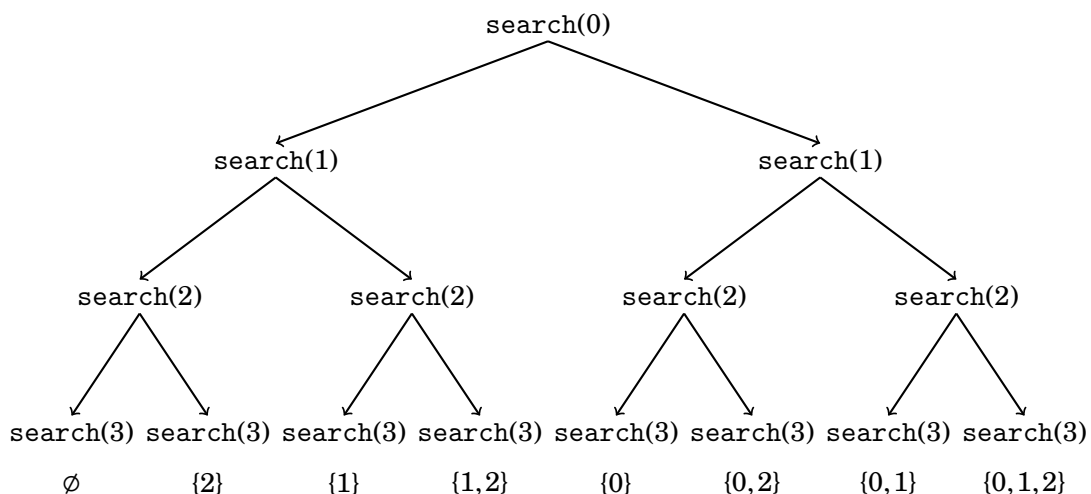
```

    }
}

```

Khi hàm `search` được gọi với tham số  $k$ , nó sẽ quyết định có đưa phần tử  $k$  và tập hợp con hoặc không, và trong cả hai trường hợp, nó gọi chính nó với tham số  $k + 1$ . Tuy nhiên, nếu  $k = n$ , mọi phần tử đã được xử lý và một tập hợp con vừa được sinh ra.

Cây dưới đây biểu diễn cho hàm khi gọi với  $n = 3$ . Chúng ta có thể luôn luôn lựa chọn nhánh cây bên trái ( $k$  không được nằm trong tập hợp con) hoặc nhánh cây bên phải ( $k$  nằm trong tập hợp con).



## Phương pháp 2

Một cách khác để sinh các tập hợp con là dựa vào biểu diễn nhị phân của các số nguyên. Mỗi tập hợp con là một tập hợp gồm  $n$  phần tử có thể được biểu diễn bởi một dãy gồm  $n$  bit, tương ứng với một số nguyên nằm giữa  $0 \dots 2^n - 1$ . Các số một trong biểu diễn dãy bit cho biết các phần tử nào được nằm trong tập hợp con.

Các liên kết thông sẽ là bit cuối cùng sẽ tương ứng với phần tử 0, bit kế cuối sẽ tương ứng với phần tử 1, và tiếp tục như thế. Ví dụ, biểu diễn nhị phân của 25 là 11001, tương ứng với tập hợp con  $\{0, 3, 4\}$ .

Phần code bên dưới duyệt qua toàn bộ tập hợp con của một tập hợp gồm  $n$  phần tử

```

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

```

Phần code bên dưới cho chúng ta thấy cách tìm các phần tử của tập hợp con tương ứng với một dãy nhị phân. Khi xử lý mỗi tập hợp con, code này xây dựng một vector để chứa các phần tử trong một tập hợp 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 Sinh hoán vị

Tiếp theo chúng ta xét bài toán sinh toàn bộ 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  $(2, 1, 0)$ . Có hai cách tiếp cận: chúng ta có thể sử dụng đệ quy hoặc lặp qua từng hoán vị

### Phương pháp 1

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

```

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

```

Mỗi lần gọi hàm thêm một phần tử mới vào trong `permutation`. Mảng `chosen` cho ta biết các phần tử nào đã được thêm vào trong hoán vị. Nếu như kích thước của `permutation` bằng với kích thước của tập hợp, một hoán vị đã được sinh ra.

### Phương pháp 2

Một phương pháp khác để sinh hoán vị chính 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

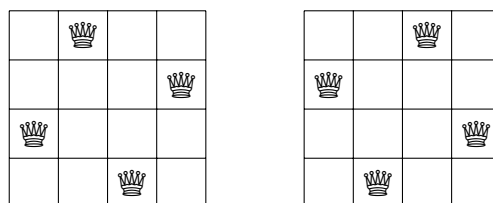
theo theo thứ tự tăng dần. Thư viện chuẩn C++ có hàm `next_permutation` có thể sử dụng cho việc này:

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

## 5.3 Quay lui

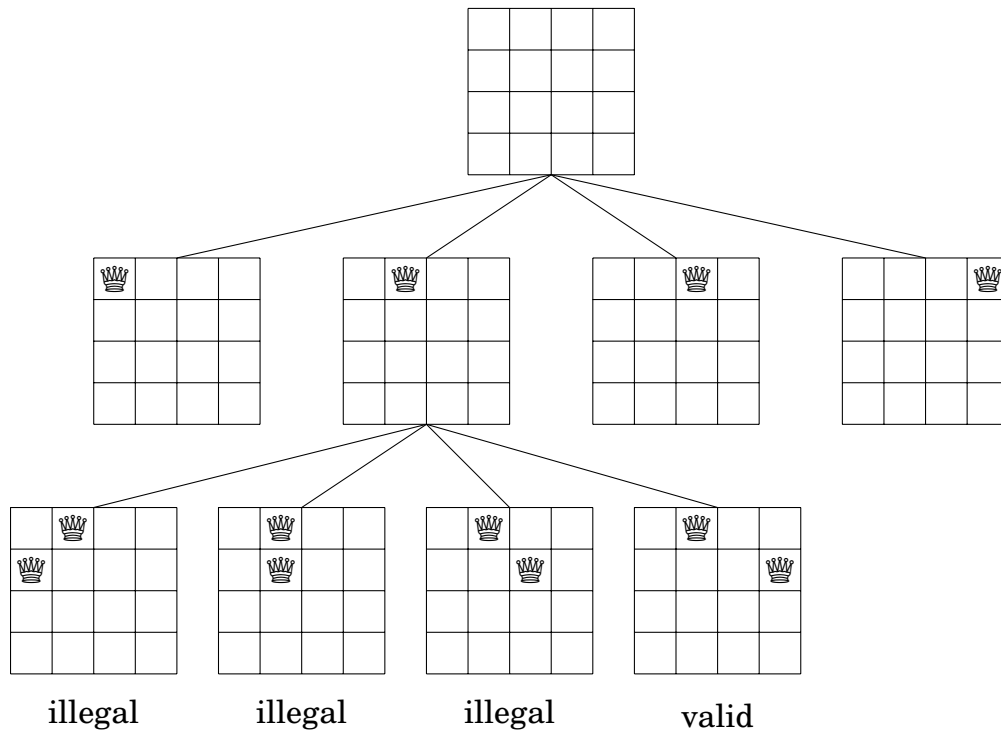
Một thuật toán **quay lui** bắt đầu với một đáp án rỗng và mở rộng đáp án theo từng bước. Việc tìm kiếm bằng đệ quy để duyệt qua tất cả các cách khác nhau mà đáp án có thể được xây dựng.

Ví dụ, xét bài toán đếm số lượng cách xếp  $n$  quân hậu trên một bàn cờ  $n \times n$  để không có hai quân hậu nào có thể ăn nhau. Khi  $n = 4$ , có hai cách xếp khác nhau:



Bài toán có thể được giải sử dụng quay lui bằng cách đặt quân hậu lên bàn cờ theo từng hàng. Cụ thể hơn, mỗi một quân hậu sẽ được đặt lên mỗi hàng sao cho không quân hậu nào có thể ăn các quân hậu đã được đặt trước đó. Một cách xếp được tìm khi toàn bộ  $n$  quân hậu đã được đặt lên bàn cờ.

Ví dụ, khi  $n = 4$ , một phân cách xếp được sinh ra bằng thuật toán đệ quy như sau:



Ở hàng dưới cùng, ba cách đặt đầu tiên là phạm luật, bởi vì các quân hậu có thể ăn nhau. Tuy nhiên, cách đặt thứ tư thì thoả mãn và nó có thể được mở rộng thành một cách xếp hoàn chỉnh bằng cách đặt thêm hai quân hậu nữa lên bàn cờ. Chỉ có duy nhất một cách để đặt hai quân hậu còn lại.

The algorithm can be implemented as follows:

```
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 việc gọi `search(0)`. Kích thước của bàn cờ là  $n \times n$ , và biến `count` chứa số lượng cách xếp.

Trong phần code, hàng và cột trên bàn cờ được đánh số từ 0 đến  $n - 1$ . Khi hàm `search` được gọi với tham số  $y$ , nó đặt một quân hậu lên hàng  $y$  và rồi tự gọi chính mình với tham số  $y + 1$ . Và rồi, khi  $y = n$ , một cách xếp đã được tìm và biến `count` được tăng thêm một.

The array `column` keeps track of columns that contain a queen, and the arrays `diag1` and `diag2` keep track of diagonals. It is not allowed to add

another queen to a column or diagonal that already contains a queen. For example, the columns and diagonals of the  $4 \times 4$  board are numbered as follows:

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

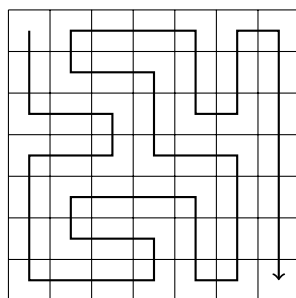
diag2

Cho  $q(n)$  là số cách để đặt đặt  $n$  quân hậu trên một bàn cờ  $nn \times n$ . Thuật toán đệ quy trên cho chúng ta biết, ví dụ,  $q(8) = 92$ . Khi  $n$  tăng, hàm tìm kiếm nhanh chóng trở nên chậm hơn, vì số lượng đáp án tăng theo cấp số nhân. Ví dụ, để tính toán  $q(16) = 14772512$  bằng việc sử dụng thuật toán trên tốn khoảng một phút trên máy tính bây giờ<sup>1</sup>.

## 5.4 Duyệt cận

Chúng ta thường có thể tối ưu đệ quy bằng cách cắt bớt cây tìm kiếm. Ý tưởng là sẽ thêm "trí tuệ" vào trong thuật toán để nó phát hiện ngay nếu một phần đáp án đang được xây dựng không thể trở thành một đáp án hoàn chỉnh. Các tối ưu như vậy có thể có một tác động lớn đến hiệu quả của việc tìm kiếm.

Xét bài toán đếm số đường đi trên một bảng  $n \times n$  xuất phát từ ô trái trên đến ô phải dưới sao cho đường đi thăm các ô chính xác một lần. Ví dụ, trên một bảng  $7 \times 7$ , có tất cả 111712 đường đi thoả mãn. Một trong số các đường đi như sau:



Ta tập trung vào trường hợp  $7 \times 7$ , bởi vì độ khó phù hợp với chúng ta. Ta bắt đầu bằng một thuật toán duyệt tường minh, sau đó tối ưu dần sử dụng các nhận xét để duyệt cận. Sau mỗi phép tối ưu, chúng ta đo đạc thời gian chạy của thuật toán và số lần gọi đệ quy để chúng ta có thể nhìn thấy rõ ràng tác dụng của mỗi phép tối ưu lên việc tìm kiếm.

<sup>1</sup>Chưa ai tìm ra cách để tính toán hiệu quả các giá trị lớn hơn của  $q(n)$ . Kỉ lục hiện giờ là  $q(27) = 234907967154122528$ , được tính vào năm 2016 [55].

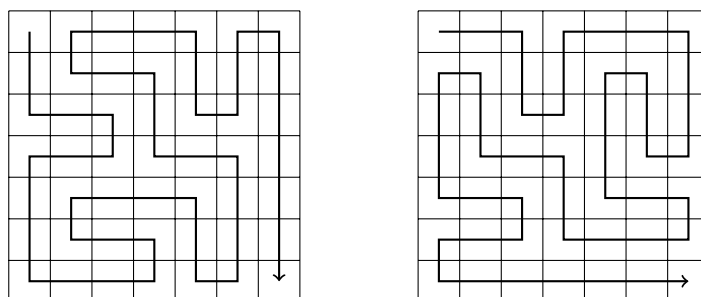
## Thuật toán cơ bản

Phiên bản đầu tiên của thuật toán không bao gồm bất kỳ tối ưu nào. Ta đơn giản sử dụng quay lui để sinh ra bất kỳ đường đi từ góc trái trên đến góc phải dưới và đếm số lượng đường đi đó.

- thời gian chạy: 483 seconds
- số lần gọi đệ quy: 76 tỉ

## Tối ưu 1

Trong mọi đáp án, chúng ta luôn bắt đầu một đi xuống hoặc đi phải. Luôn luôn có hai đường đi đối xứng với nhau theo đường chéo của bảng sau bước đầu tiên. Ví dụ, các đường đi sau đối xứng:

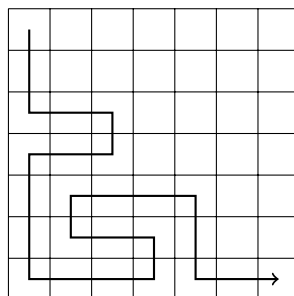


Vậy nên, chúng ta quyết định rằng luôn bắt đầu đi một bước xuống (hoặc phải), sau đó nhân số lượng đáp án cho hai.

- thời gian chạy: 244 giây
- số lần gọi đệ quy: 38 tỉ

## Tối ưu 2

Nếu đường đi đến ô phải dưới trước khi nó thăm tất cả các ô khác trên bảng, rõ ràng rằng đây không thể nào là đáp án hoàn chỉnh. Ví dụ cho trường hợp này chính là đường đi sau

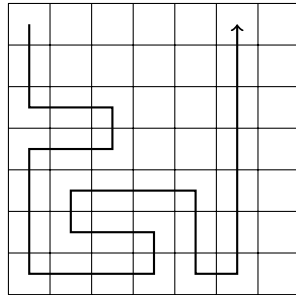


Sử dụng nhận xét này, chúng ta có thể dừng việc tìm kiếm ngay lập tức khi chúng ta đến ô phải dưới quá sớm

- thời gian chạy: 119 giây
- số lần gọi đệ quy: 20 tỉ

### Tối ưu 3

Nếu như đường đi chạm vào biên và có thể trái hoặc phải, bảng được tách ra làm hai phần chứ những ô chưa đi qua. Ví dụ, trong trường hợp sau, chúng ta có thể rẽ trái hoặc phải:

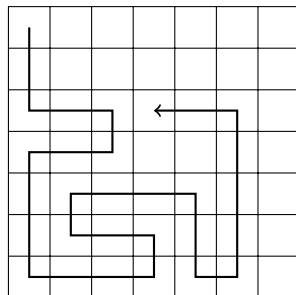


Trong trường hợp này, chúng ta không thể thăm tất cả các ô nữa, nên chúng ta có thể dừng việc tìm kiếm. Tối ưu này rất hữu dụng:

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

### Tối ưu 4

Ý tưởng của tối ưu 3 có thể được tổng quát: nếu như đường đi không thể đi tiếp nhưng có thể rẽ trái hoặc phải, bảng được chia làm hai phần chứa các ô chưa được thăm. Ví dụ, xét đường đi sau:



Rõ ràng rằng chúng ta không thể thăm tất cả các ô nữa, nên chúng ta có thể dừng việc tìm kiếm. Sau tối ưu này, hàm tìm kiếm trở nên rất nhanh.

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

Bây giờ là lúc chúng ta dừng lại và xem những gì đã đạt được. Thời gian chạy của thuật toán ban đầu là 483 giây, và sau các tối ưu, thời gian chạy bây giờ là 0.6 giây. Vậy, thuật toán đã trở nên nhanh hơn gần 1000 lần sau khi tối ưu.

Đây là một việc thường thấy trong quay lui, vì cây tìm kiếm thường rất lớn và kể cả các nhận xét cơ bản cũng có thể rút ngắn việc tìm kiếm một cách hiệu quả. Những bước tối ưu ở những bước đầu tiên của thuật toán thường hiệu quả hơn (ở trên đỉnh của cây tìm kiếm)

## 5.5 Duyệt phân tập

**Duyệt phân tập** là một kĩ thuật khi không gian tìm kiếm được chia thành hai phần tương đối bằng nhau. Thực hiện tìm kiếm riêng biệt cả hai phần, và cuối cùng kết quả của các lần tìm kiếm được kết hợp lại cùng nhau.

Kĩ thuật này có thể được sử dụng nếu có một cách hiệu quả để kết hợp các đáp án của hai cách tìm kiếm. Trong trường hợp như thế, hai lần tìm kiếm có thể cần ít thời gian hơn một lần tìm kiếm toàn bộ. Thông thường, chúng ta từ  $2^n$  sang  $2^{n/2}$  sử dụng kĩ thuật duyệt phân tập.

Như một ví dụ, xét bài toán cho một danh sách gồm  $n$  số và một số  $x$ , và chúng ta muốn tìm ra xem có thể chọn một vài số trong danh sách sao cho tổng của chúng bằng  $x$ . Ví dụ, xét danh sách  $[2, 4, 5, 9]$  và  $x = 15$ , chúng ta có thể chọn các số  $[2, 4, 9]$  để đạt được  $2 + 4 + 9 = 15$ . Tuy nhiên, nếu  $x = 10$  và danh sách vẫn giữ nguyên, không thể tạo thành tổng bằng  $x$ .

Một thuật toán đơn giản cho bài toán này là duyệt qua toàn bộ tập hợp con và kiểm tra xem tổng của bất kì tập hợp con nào bằng  $x$ . Độ phức tạp của thuật toán là  $O(2^n)$ , bởi vì có tất cả  $2^n$  tập hợp con. Tuy nhiên, sử dụng kĩ thuật duyệt phân tập, chúng ta có thể đạt được một độ phức tạp tốt hơn  $O(2^{n/2})$ <sup>2</sup>. Lưu ý là  $O(2^n)$  và  $O(2^{(n/2)})$  là độ phức tạp khác nhau vì  $2^{n/2}$  bằng  $\sqrt{2^n}$ .

Ý tưởng là chia danh sách ra làm hai danh sách  $A$  và  $B$  sao cho cả hai danh sách chứa khoảng một nửa các số. Lần tìm kiếm đầu tiên tạo ra tất cả các tập hợp con của  $A$  và lưu tổng của chúng trong danh sách  $S_A$ . Tương tự như thế, lần tìm kiếm thứ hai tạo một danh sách  $S_B$  từ  $B$ . Sau đó, chỉ cần kiểm tra xem có thể chọn một phần tử trong tập  $S_A$  và một phần tử khác trong tập  $S_B$  sao cho tổng của chúng là  $x$ . Chúng ta chỉ có thể chọn chỉ khi tồn tại một cách để tạo tổng  $x$  khi sử dụng các số trên danh sách gốc.

Ví dụ, giả sử như danh sách là  $[2, 4, 5, 9]$  và  $x = 15$ . Đầu tiên, ta chia danh sách ra thành  $A = [2, 4]$  và  $B = [5, 9]$ . Sau đó, ta tạo ra 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, bởi vì  $S_A$  chứa tổng 6,  $S_B$  chứa tổng 9 và  $6 + 9 = 15$  tương ứng với đáp án  $[2, 4, 9]$ .

Chúng ta có thể cài đặt thuật toán với độ phức tạp là  $O(2^{n/2})$ . Đầu tiên, chúng ta tạo ra hai danh sách đã được sắp xếp  $S_A$  và  $S_B$  trong  $O(2^{n/2})$  sử

---

<sup>2</sup>Ý tưởng này được giới thiệu vào năm 1974 bởi E. Horowitz và S. Sahni [39].

dụng kĩ thuật gộp. Sau đó, khi danh sách đã được sắp xếp, chúng ta có thể kiểm tra trong  $O(2^{n/2})$  nếu như tổng  $x$  có thể được tạo từ  $S_A$  và  $S_B$  hay không.

We can implement the algorithm so that its time complexity is  $O(2^{n/2})$ . First, we generate *sorted* lists  $S_A$  and  $S_B$ , which can be done in  $O(2^{n/2})$  time using a merge-like technique. After this, since the lists are sorted, we can check in  $O(2^{n/2})$  time if the sum  $x$  can be created from  $S_A$  and  $S_B$ .



## Chương 6

# Giải thuật tham lam

Một **giải thuật tham lam** xây dựng lời giải cho bài toán bằng cách luôn lựa chọn phương án trông có vẻ tốt nhất hiện tại. Một giải thuật tham lam không bao giờ hoàn lại các lựa chọn, mà luôn dựng nên lời giải cuối cùng một cách trực tiếp. Vì lí do này, các giải thuật tham lam thường rất nhanh.

Điều khó khăn trong việc thiết kế các giải thuật tham lam là tìm ra một chiến thuật tham mà luôn tạo ra phương án tối ưu cho bài toán. Những lựa chọn tối ưu cục bộ trong một thuật toán tham lam cũng phải tối ưu toàn cục. Việc chứng minh một thuật toán tham đúng thường khá khó.

### 6.1 Bài toán đồng xu

Ví dụ đầu tiên là bài toán sau: Cho trước một tập các đồng xu và nhiệm vụ của ta là tạo nên tổng tiền  $n$  bằng những đồng xu đó. Các đồng xu có giá trị là  $\text{coins} = \{c_1, c_2, \dots, c_k\}$ , và mỗi xu có thể sử dụng bao nhiêu lần tùy ý. Hỏi cần số đồng xu tối thiểu là bao nhiêu?

Ví dụ, nếu các đồng xu là đồng euro (theo đơn vị cent)

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

và  $n = 520$ , ta cần ít nhất bốn đồng. Phương án tối ưu là dùng các xu  $200 + 200 + 100 + 20$  có tổng bằng 520.

#### Thuật tham

Một thuật toán tham đơn giản cho bài trên là luôn chọn ra đồng xu lớn nhất có thể, cho tới khi đạt được tổng lượng tiền yêu cầu. Thuật này đúng trong test mẫu, bởi đầu tiên ta sẽ dùng hai đồng 200 cent, sau đó là một đồng 100 cent và cuối cùng là một đồng 20 cent. Nhưng liệu thuật toán này có luôn đúng?

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

Đầu tiên, mỗi đồng xu 1, 5, 10, 50 và 100 xuất hiện nhiều nhất một lần trong phương án tối ưu, bởi nếu phương án đó tồn tại hai đồng như vậy, ta có thể thay chúng bằng một đồng khác và ra được phương án tốt hơn. Ví dụ, nếu phương án có chứa các đồng 5 + 5, ta có thể thay chúng thành đồng 10

Tương tự, các đồng 2 và 20 xuất hiện tối đa hai lần trong một phương án tối ưu, bởi chúng ta có thể thay thế các đồng 2 + 2 + 2 thành 5 + 1 và 20 + 20 + 20 thành 50 + 10. Hơn nữa, phương án tối ưu không thể chứa tổng 2 + 2 + 1 hay 20 + 20 + 10, vì ta có thể thay chúng thành 5 hoặc 50.

Bằng những quan sát này, ta chứng minh được với mỗi xu  $x$ , không thể tạo ra tổng  $x$  hoặc tổng bất kì lớn hơn bằng cách chỉ sử dụng những đồng xu nhỏ hơn  $x$ . Ví dụ, nếu  $x = 100$ , tổng tối ưu lớn nhất từ những đồng nhỏ hơn là  $50 + 20 + 20 + 5 + 2 + 2 = 99$ . Vì thế, thuật tham: luôn chọn đồng xu lớn nhất cho ra phương án tối ưu.

Ví dụ này cho thấy việc chứng minh thuật tham đúng có thể khó, thậm chí khi bản thân thuật tham đó đơn giản.

## Trường hợp tổng quát

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

Chúng ta có thể chứng minh một thuật tham không đúng bằng cách chỉ ra một phản ví dụ mà tại đó thuật tham cho kết quả sai. Ở bài này ta dễ dàng tìm được một phản ví dụ: nếu các đồng xu là  $\{1, 3, 4\}$  và tổng cần tạo là 6, thuật tham sẽ đưa ra phương án  $4 + 1 + 1$  trong khi cách tối ưu là  $3 + 3$

Không thể biết được bài toán đồng xu tổng quát có thể giải được bằng một thuật tham bất kì hay không.<sup>1</sup>

Tuy nhiên, khi tới Chương 7 ta sẽ thấy, trong vài trường hợp, bài toán tổng quát có thể giải được một cách hiệu quả bằng một thuật toán quy hoạch động luôn cho ra kết quả đúng.

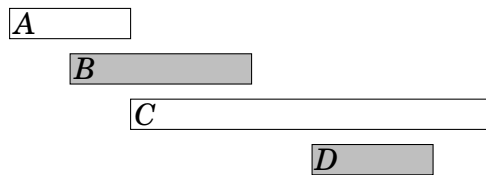
## 6.2 Xếp lịch

Nhiều bài toán xếp lịch có thể giải được bằng các thuật toán tham lam. Sau đây là một bài toán kinh điển: Cho  $n$  sự kiện, cùng với thời điểm bắt đầu và kết thúc của chúng, hãy lập một lịch trình chứa nhiều sự kiện nhất có thể. Không được chọn ra chỉ một phần của một sự kiện nào đó (mỗi sự kiện được chọn phải nằm trọn trong lịch trình). Ví dụ, xét các sự kiện sau:

sự kiện	thời điểm bắt đầu	thời điểm kết thúc
$A$	1	3
$B$	2	5
$C$	3	9
$D$	6	8

<sup>1</sup>Tuy nhiên, việc *kiểm tra* trong thời gian đa thức, rằng thuật tham trình bày ở chương này đúng với một tập xu cho trước là khả thi [53].

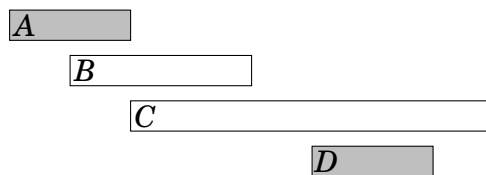
Trong trường hợp này, số lượng sự kiện tối đa là hai. Ví dụ, ta có thể chọn các sự kiện *B* và *D* như sau:



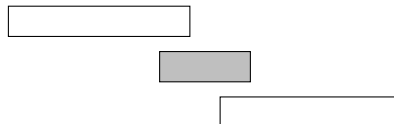
Có thể sáng tạo ra một vài thuật tham khác nhau cho bài này, nhưng thuật nào sẽ đúng trong mọi trường hợp?

## Thuật toán 1

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



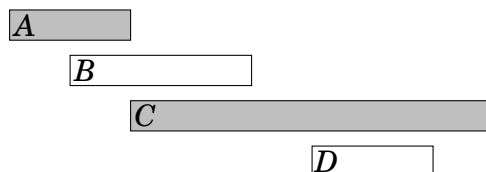
Tuy nhiên, việc lựa chọn những sự kiện ngắn không phải lúc nào cũng là một chiến thuật đúng đắn. Ví dụ, thuật trên sẽ sai trong trường hợp sau:



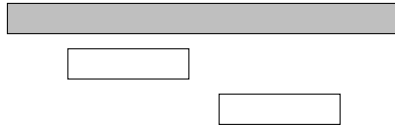
Nếu chúng ta chọn sự kiện ngắn, ta chỉ có thể chọn một. Tuy nhiên, có thể chọn cả hai sự kiện dài.

## Thuật toán 2

Ý tưởng khác là luôn lựa chọn sự kiện tiếp theo (nếu có) mà *bắt đầu* càng *sớm* càng tốt. Thuật này lựa chọn những sự kiện sau:



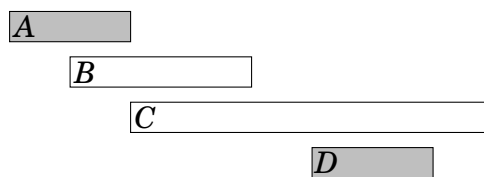
Tuy nhiên, ta cũng có thể tìm một phản ví dụ cho thuật toán này. Ví dụ, trường hợp sau, thuật trên chỉ chọn một sự kiện.



Nếu chúng ta lựa chọn sự kiện đầu tiên thì không thể chọn thêm sự kiện nào khác. Tuy nhiên, chỉ chọn hai sự kiện còn lại thì được.

### Thuật toán 3

Ý tưởng thứ ba là luôn chọn sự kiện tiếp theo (nếu có) mà *kết thúc* càng sớm càng tốt. Thuật này lựa chọn những sự kiện sau:



Hóa ra thuật này *luôn luôn* cho ra lời giải tối ưu. Lý do là, việc lựa chọn sự kiện kết thúc sớm nhất trước luôn tối ưu. Sau đó, chọn sự kiện tiếp theo với chiến thuật như trên cũng tối ưu, v.v., Cứ như vậy cho tới khi không thể chọn thêm sự kiện nào khác.

Một cách để chứng minh thuật toán đúng là xét xem nếu ta chọn trước một sự kiện kết thúc trễ hơn sự kiện kết thúc sớm nhất thì điều gì sẽ xảy ra. Lúc này, số lựa chọn cho sự kiện tiếp theo nhiều nhất cũng chỉ bằng với khi ta chọn sự kiện kết thúc sớm nhất trước. Do đó, việc chọn một sự kiện kết thúc trễ hơn không bao giờ cho ra phương án tốt hơn, vì thế thuật tham đúng.

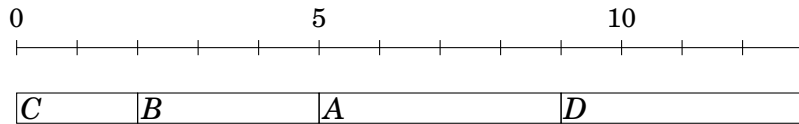
## 6.3 Công việc và thời hạn

Chúng ta cùng xét bài toán sau: Cho  $n$  công việc cùng với thời lượng, và hạn chót. Nhiệm vụ của chúng ta là chọn một thứ tự để thực hiện các công việc. Với mỗi việc, ta sẽ nhận được  $d - x$  điểm với  $d$  là hạn chót của việc đó và  $x$  là thời điểm ta hoàn thành nó. Tổng số điểm lớn nhất mà ta có thể nhận được là bao nhiêu?

Ví dụ, giả sử có các công việc như sau:

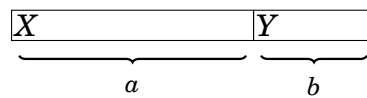
việc	thời gian	hạn chót
A	4	2
B	3	5
C	2	7
D	4	5

Trong trường hợp này, một lịch trình thực hiện công việc tối ưu trông như sau:

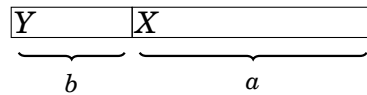


Ở phương án trên,  $C$  cho ra 5 điểm,  $B$  cho 0 điểm,  $A$  cho  $-7$  điểm và  $D$  cho  $-8$  điểm, do đó tổng điểm là  $-10$ .

Ngạc nhiên thay, lời giải tối ưu cho bài toán không hề phụ thuộc vào hạn chót. Một chiến thuật tham đúng đắn chỉ đơn giản là thực hiện các công việc theo thứ tự *tăng dần về thời lượng*. Lý do cho việc này là nếu có lúc nào mà ta thực hiện hai việc liên tiếp nhau, mà việc thứ nhất tốn nhiều thời gian hơn việc thứ hai, ta có thể có được phương án tốt hơn nếu đổi chỗ hai việc này. Ví dụ, xem lịch trình sau:



Ở đây  $a > b$ , ta đổi chỗ hai việc:



Lúc này  $X$  cho ít đi  $b$  điểm, và  $Y$  cho thêm  $a$  điểm, vì thế tổng điểm tăng một lượng  $a - b > 0$ . Trong phương án tối ưu, với hai việc liên tiếp bất kì, việc ngắn hơn phải làm trước việc lâu hơn. Do đó, các công việc phải được thực hiện theo thời lượng tăng dần.

## 6.4 Tối ưu tổng

Tiếp theo, chúng ta xét bài toán: Cho  $n$  số  $a_1, a_2, \dots, a_n$  nhiệm vụ của ta là tìm một giá trị  $x$  để tổng

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

đạt cực tiểu. Chúng ta tập trung vào hai trường hợp  $c = 1$  và  $c = 2$

### Trường hợp $c = 1$

Trong trường hợp này, cần tối thiểu hóa tổng

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

Ví dụ, nếu các số là  $[1, 2, 9, 2, 6]$ , phương án tốt nhất là chọn  $x = 2$ , cho ra tổng

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

Tổng quát, lựa chọn tốt nhất cho  $x$  chính là *trung vị* của các số, là số chính giữa sau khi sắp xếp. Ví dụ, dãy số  $[1, 2, 9, 2, 6]$  sau khi sắp xếp sẽ thành  $[1, 2, 2, 6, 9]$ , do đó trung vị là 2.

Trung vị là lựa chọn tối ưu, bởi nếu  $x$  bé hơn trung vị, có thể làm tổng nhỏ hơn bằng cách tăng  $x$ , và nếu  $x$  lớn hơn trung vị, tổng có thể trở nên nhỏ hơn bằng cách giảm  $x$ . Vì thế, phương án tối ưu là chọn  $x$  bằng với trung vị. Nếu  $n$  chẵn và có hai trung vị, thì cả hai số đó và mọi giá trị nằm giữa chúng đều là lựa chọn tối ưu.

## Trường hợp $c = 2$

Trong trường hợp này, cần tối thiểu hóa tổng

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

Ví dụ, nếu các số là  $[1, 2, 9, 2, 6]$ , phương án tối ưu là chọn  $x = 4$ , cho ra tổng

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

Tổng quát, lựa chọn tốt nhất cho  $x$  chính là *trung bình cộng* của dãy. Trong ví dụ trên, trung bình cộng là  $(1 + 2 + 9 + 2 + 6)/5 = 4$ . Kết quả này có thể rút ra được bằng cách viết lại tổng như sau:

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

Phần cuối không phụ thuộc vào  $x$ , do đó ta có thể phớt lờ nó. Phần còn lại tạo thành hàm  $nx^2 - 2xs$  trong đó  $s = a_1 + a_2 + \cdots + a_n$ . Đây là một đường cong parabol hướng lên, với nghiệm  $x = 0$  và  $x = 2s/n$ , và giá trị tối thiểu chính là trung bình cộng của các nghiệm  $x = s/n$ , tức là, trung bình cộng của các số  $a_1, a_2, \dots, a_n$ .

## 6.5 Nén dữ liệu

Trong **Mã hóa nhị phân** (Binary code), ta gán cho mỗi kí tự trong xâu một **từ khóa** (codeword) bao gồm các bit. Chúng ta có thể *nén* xâu theo mã hóa nhị phân bằng cách thay thế mỗi kí tự bởi từ khóa tương ứng. Ví dụ, bảng mã nhị phân này gán các từ khóa cho các kí tự như sau A–D:

kí tự	từ khóa
A	00
B	01
C	10
D	11

Đây là bảng mã với **độ dài hằng**, nghĩa là độ dài của mỗi từ khóa là như nhau. Ví dụ, chúng ta có thể nén xâu AABACDACA như sau:

000001001011001000

Dùng cách này, độ dài của xâu nén là 18 bit. Tuy nhiên, chúng ta có thể nén xâu hiệu quả hơn nếu sử dụng bảng mã **độ dài biến thiên**, trong đó, các từ khóa có thể có độ dài khác nhau. Do đó, ta có thể gán những từ khóa ngắn cho những kí tự xuất hiện thường xuyên và từ khóa dài cho những kí tự ít xuất hiện. Hóa ra, một bảng mã **tối ưu** cho xâu trên trông như sau:

kí tự	từ khóa
A	0
B	110
C	10
D	111

Một bảng mã tối ưu sẽ cho ra xâu nén ngắn nhất có thể. Trong trường hợp này, xâu được nén theo cách trên là

001100101110100,

chỉ có 15 bit cần dùng thay vì 18 bit. Nhờ vào bảng mã tốt hơn, ta có thể tiết kiệm được 3 bit trong xâu nén. Yêu cầu: không có từ khóa nào là tiền tố của từ khóa khác. Ví dụ, một bảng mã không được phép chứa đồng thời các từ khóa 10 và 1011. Lí do là ta muốn đảm bảo sinh được xâu gốc từ xâu nén. Nếu có một từ khóa là tiền tố của từ khóa khác, việc này không phải lúc nào cũng khả thi. Ví dụ, bảng mã sau là *không* hợp lệ:

kí tự	từ khóa
A	10
B	11
C	1011
D	111

Sử dụng bảng mã này, ta không thể biết được xâu nén 1011 tương ứng với xâu AB hay xâu C.

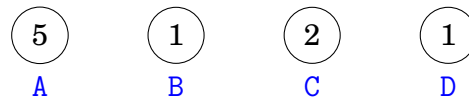
## Mã hóa Huffman

**Mã hóa Huffman**<sup>2</sup> là một thuật toán tham lam, dựng nên một cách mã hóa tối ưu để nén một xâu cho trước. Thuật toán xây dựng một cây nhị phân dựa trên tần suất của các kí tự trong xâu, và từ khóa của mỗi kí tự có thể đọc được bằng cách lần theo đường đi từ gốc tới nút tương ứng. Đi sang trái tương ứng với bit 0, đi sang phải tương ứng với bit 1

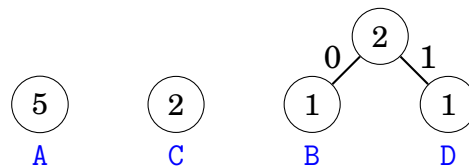
Ban đầu, mỗi kí tự trong xâu được biểu diễn bởi một nút mà trọng số của nó chính là số lần xuất hiện của kí tự đó trong xâu. Sau đó, tại mỗi bước, hai nút với trọng số nhỏ nhất được gộp lại bằng cách tạo ra một nút mới có trọng số bằng tổng trọng số của hai nút ban đầu. Quá trình tiếp diễn cho tới khi toàn bộ đỉnh đã được kết hợp.

<sup>2</sup>D. A. Huffman phát hiện phương pháp này khi giải một bài tập trong giáo trình đại học và công bố thuật toán vào năm 1952 [40].

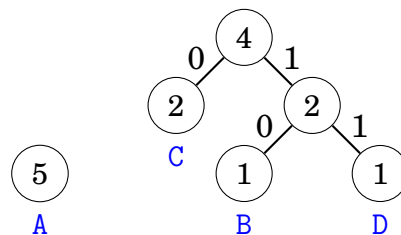
Tiếp đến, ta sẽ xem thử mã hóa Huffman tạo ra bảng mã tối ưu cho chuỗi AABACDACA như thế nào. Ban đầu, có bốn nút tương ứng với các kí tự của chuỗi:



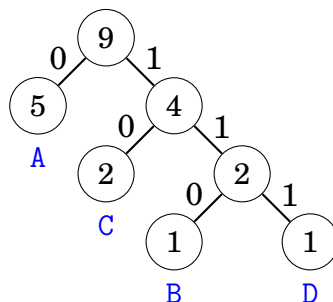
Nút tượng trưng cho kí tự A có trọng số 5 bởi kí tự A xuất hiện 5 lần trong chuỗi. Những trọng số khác được tính tương tự. Bước đầu tiên là gộp hai nút ứng với các kí tự B và D, đều có trọng số 1. Kết quả:



Sau đó, các nút với trọng số 2 được gộp lại:



Cuối cùng, hai nút còn lại được gộp:



Bây giờ mọi nút đều ở trong cây, bảng mã đã hoàn tất. Ta có thể đọc được những từ khóa sau từ cây đã dựng:

kí tự	từ khóa
A	0
B	110
C	10
D	111



# Chương 7

## Quy hoạch động

**Quy hoạch động** là một kĩ thuật kết hợp được tính đúng đắn của duyệt đầy đủ và tính hiệu quả của các thuật tham. Quy hoạch động có thể được áp dụng nếu bài toán có thể chia thành các bài toán con mà chúng có thể giải riêng, không phụ thuộc nhau.

Quy hoạch động có hai công dụng:

- **Tìm phương án tối ưu:** Ta muốn tìm một đáp số lớn nhất hoặc nhỏ nhất có thể.
- **Đếm số nghiệm:** Ta muốn tính tổng số cách có thể có.

Trước hết, ta sẽ xem cách dùng Quy hoạch động để tìm một lời giải tối ưu, sau đó ta sẽ dùng ý tưởng này để đếm số lời giải.

Hiểu được Quy hoạch động là một cột mốc quan trọng trong sự nghiệp của mỗi lập trình viên thi đấu. Trong khi ý tưởng cơ bản rất đơn giản, thử thách nằm ở việc áp dụng Quy hoạch động vào các bài toán khác. Chương này giới thiệu một số bài toán cổ điển để bắt đầu.

### 7.1 Bài toán đồng xu

Đầu tiên ta sẽ tập trung vào một bài toán mà ta đã thấy ở Chương 6: Cho một tập các mệnh giá đồng xu  $\text{coins} = \{c_1, c_2, \dots, c_k\}$  và một số tiền  $n$ , nhiệm vụ của ta là tạo ra tổng  $n$  bằng cách sử dụng ít đồng xu nhất có thể.

Trong Chương 6, ta đã giải quyết bài toán sử dụng một thuật tham luôn chọn đồng xu lớn nhất có thể. Thuật tham này đúng, ví dụ, khi các đồng xu là đồng euro, nhưng trong trường hợp tổng quát thuật tham này không đảm bảo tạo ra một lời giải tối ưu.

Bây giờ là lúc giải quyết bài toán này một cách hiệu quả bằng quy hoạch động, sao cho thuật toán đúng với bất kỳ tập đồng xu nào. Thuật toán quy hoạch động dựa trên một hàm đệ quy duyệt qua tất cả các cách tạo ra tổng, giống như thuật toán cây trầu. Tuy nhiên, thuật toán quy hoạch động hiệu quả bởi nó sử dụng *memoization* và tính toán đáp án cho mỗi bài toán con một lần duy nhất.

## Hệ thức truy hồi

Ý tưởng chính của Quy hoạch động là phát biểu bài toán một cách đệ quy để đáp án của nó có thể tính được từ đáp án của các bài toán con. Trong bài toán đồng xu, có một vấn đề đệ quy là: số đồng xu tối thiểu cần để tạo ra tổng  $x$  là bao nhiêu?

The idea in Quy hoạch động is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum  $x$ ?

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

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

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

Tính chất cơ bản của  $\text{solve}$  là một giá trị của nó có thể tính được một cách đệ quy từ các giá trị nhỏ hơn. Ý tưởng là tập trung vào đồng xu *đầu tiên* mà ta chọn để tạo ra tổng. Ví dụ, trong trường hợp trên, đồng xu đầu tiên có thể là 1, 3 hoặc 4. Nếu ta chọn đồng xu 1, việc còn lại là tạo ra tổng 9 bằng số đồng xu tối thiểu, đây là một bài toán con của bài toán gốc. Tương tự, đồng xu đầu tiên có thể là 3 hoặc 4. Do đó, ta có thể sử dụng công thức đệ quy sau để tính số đồng xu tối thiểu:

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

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

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

Giờ ta đã có thể đưa ra một hàm đệ quy tổng quát để tính số đồng xu tối

thiếu cần để tạo ra 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  $\infty$ , vì không thể tạo ra tổng âm. Tiếp theo, nếu  $x = 0$ , giá trị là 0, vì không cần đồng xu nào để tạo ra tổng rỗng. Cuối cùng, nếu  $x > 0$ , ta duyệt biến  $c$  để xét tất cả các cách chọn đồng xu đầu tiên cho tổng.

Một khi ta đã tìm được một hàm đệ quy giải quyết bài toán, ta có thể trực tiếp cài đặt một lời giải bằng C++. (hằng số INF kí hiệu 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;
}
```

Hàm này vẫn chưa hiệu quả, vì có thể số lượng cách tạo ra tổng  $x$  tăng nhanh như hàm lũy thừa. Tuy nhiên, tiếp theo đây, ta sẽ xem cách khiến cho hàm này hiệu quả bằng cách sử dụng một kỹ thuật gọi là **memoization** - (**bảng nhớ**).

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

## Dùng bảng nhớ

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

Trong bài này, ta sử dụng các mảng

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

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

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

```

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

```

Hàm này xử lý các trường hợp cơ sở  $x < 0$  và  $x = 0$  như đã nêu trên. Sau đó, hàm kiểm tra `ready[x]` để xem thử giá trị của `solve(x)` đã được lưu trong `value[x]` hay chưa, và nếu đã rồi, nó trực tiếp trả về giá trị này. Ngược lại, hàm tính toán giá trị của `solve(x)` một cách đệ quy và lưu giá trị này vào `value[x]`.

Hàm này chạy nhanh, bởi vì đáp án cho mỗi tham số  $x$  chỉ được tính đệ quy một lần. Sau khi một giá trị của `solve(x)` được lưu trong `value[x]`, nó có thể được lấy ra trực tiếp mỗi khi hàm được gọi lại với tham số  $x$ . Độ phức tạp thời gian của thuật toán là  $O(nk)$ , trong đó  $n$  là tổng cần tìm và  $k$  là số loại đồng xu.

Lưu ý rằng ta cũng có thể *lập* để xây dựng mảng `value` bằng cách sử dụng vòng lặp để tính toán tất cả giá trị của `solve` với 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, đa số lập trình viên thi đấu thích cách cài đặt này hơn, bởi vì nó ngắn và có hằng số thấp hơn. Từ bây giờ, chúng tôi cũng sử dụng cách cài đặt bằng vòng lặp trong các ví dụ. Tuy vậy, việc nghĩ ra các giải thuật Quy hoạch động thường vẫn dễ dàng hơn nếu nghĩ theo kiểu hàm đệ quy.

## Dựng ra một phương án

Đôi lúc chúng ta được yêu cầu tìm giá trị của một giải pháp tối ưu và đưa ví dụ về cách xây dựng một giải pháp như vậy. Trong bài toán đồng xu, chẳng hạn, chúng ta có thể khai báo một mảng khác để cho biết đồng xu đầu tiên trong một giải pháp tối ưu với mỗi tổng tiền.

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

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

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

Sau đó, đoạn mã sau có thể được sử dụng để in ra các đồng xu trong một giải pháp tối ưu với tổng  $n$ :

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

## Đếm số nghiệm

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

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

Lần nữa, ta có thể giải quyết bài toán một cách đệ quy. Đặt  $\text{solve}(x)$  là số cách để tạo ra tổng  $x$ . Ví dụ, nếu  $\text{coins} = \{1, 3, 4\}$ , thì  $\text{solve}(5) = 6$  và hệ thức truy hồi là

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

Tiếp đó, có thể viết 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 cách nào. Nếu  $x = 0$ , giá trị là 1, vì chỉ có một cách để tạo ra tổng rỗng. Ngược lại, ta tính tổng của tất cả các giá trị có dạng  $\text{solve}(x - c)$  với  $c$  thuộc `coins`.

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

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

Thông thường số lượng cách là quá lớn đến nỗi không cần tính chính xác nhưng chỉ cần đưa ra kết quả theo số dư  $m$ , ví dụ,  $m = 10^9 + 7$ . Điều này có thể làm được bằng cách thay đổi đoạn mã để mọi các phép tính đều lấy dư cho  $m$ . Trong đoạn mã trên, chỉ cần thêm dòng

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

sau dòng

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

Bây giờ ta đã thảo luận toàn bộ các ý tưởng cơ bản của Quy hoạch động. Vì Quy hoạch động có thể được sử dụng trong nhiều tình huống khác nhau, ta sẽ đi qua một tập các bài toán để thấy thêm các ví dụ về những khả năng của Quy hoạch động.


## 7.2 Dãy con tăng dài nhất

Bài toán đầu tiên là tìm **dãy con tăng dài nhất** trong một mảng gồm  $n$  phần tử. Đây là một dãy các phần tử của mảng, mà có độ dài lớn nhất. Nếu xét từ trái sang phải, mỗi phần tử trong dãy con 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



Đặt  $\text{length}(k)$  là độ dài của dãy con tăng dài nhất kết thúc tại vị trí  $k$ . Nếu ta tính được mọi giá trị của  $\text{length}(k)$  mà  $0 \leq k \leq n-1$ , ta sẽ tìm được độ dài của dãy con tăng dài nhất. Ví dụ, các giá trị của hàm trên cho mảng trên là:

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

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

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

Bởi mọi giá trị của hàm đều có thể được tính từ các giá trị nhỏ hơn của nó, ta có thể sử dụng Quy hoạch động. Trong đoạn mã sau, các giá trị của hàm sẽ được lưu 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 code trên chạy trong độ phức tạp  $O(n^2)$ , bởi vì nó bao gồm hai vòng lặp lồng nhau. Tuy nhiên, ta cũng có thể tính quy hoạch động một cách hiệu quả hơn trong độ phức tạp  $O(n \log n)$ . Bạn có thể tìm ra cách làm không?

## 7.3 Đường đi trên lưới

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

Ảnh sau cho thấy một đường đi tối ưu trên 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 trái trên đến góc phải dưới.

Giả sử rằng các hàng và cột của lưới được đánh số từ 1 đến  $n$ , và  $value[y][x]$  là giá trị của ô  $(y, x)$ . Ta định nghĩa  $sum(y, x)$  là tổng lớn nhất trên đường đi từ góc trên bên trái đến ô  $(y, x)$ . Giờ  $sum(n, n)$  cho ta biết tổng lớn nhất từ góc trái trên đến góc phải dưới. Ví dụ, trong hình trên,  $sum(5, 5) = 67$ .

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

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

Hệ thức truy hồi này dựa trên quan sát rằng một đường đi kết thúc tại ô  $(y, x)$  có thể đến từ ô  $(y, x-1)$  hoặc ô  $(y-1, x)$ :

			↓	
		→		

Do đó, ta có thể lựa chọn phương hướng làm tổng đạt tối đa. Ta giả sử rằng  $sum(y, x) = 0$  nếu  $y = 0$  hoặc  $x = 0$  (vì không tồn tại đường đi nào như vậy), vì thế hệ thức truy hồi cũng đúng khi  $y = 1$  hoặc  $x = 1$ .

Bởi hàm  $sum$  có hai tham số, mảng quy hoạch động cũng có hai chiều. Ví dụ, ta có thể sử dụng mảng

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

và tính 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 của thuật toán là  $O(n^2)$ .

## 7.4 Bài toán cái túi

Từ khóa **knapsack - cái túi** đề cập đến các bài toán trong đó một tập các đối tượng được cho trước, và ta phải tìm các tập con thỏa mãn một số tính



chất nào đó. Các bài toán cái túi thường có thể giải được bằng quy hoạch động.

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

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

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

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

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

Công thức này dựa trên sự thật rằng ta có thể dùng hoặc không dùng trọng lượng  $w_k$  vào trong tổng. Nếu ta dùng  $w_k$ , thì việc còn lại là tạo tổng  $x - w_k$  bằng cách sử dụng  $k - 1$  trọng lượng đầu tiên, và nếu ta không dùng  $w_k$ , thì việc còn lại là tạo tổng  $x$  bằng cách sử dụng  $k - 1$  trọng lượng đầu tiên. Trường hợp gốc

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

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

Bảng sau cho thấy mọi giá trị của hàm với các trọng số  $[1, 3, 3, 5]$  (ký hiệu "X" cho biết chân trị):

$k \backslash x$	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 toán các giá trị đó,  $\text{possible}(x, k)$  cho ta biết liệu ta có thể tạo tổng  $x$  bằng cách sử dụng *tất cả* các trọng lượng.

Đặt  $W$  là tổng của tất cả các trọng lượng. Lời giải quy hoạch động với độ phức tạp  $O(nW)$  sau đây tương ứng với hàm đệ quy:

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

Tuy nhiên, tiếp đây là một cách cài đặt tốt hơn chỉ sử dụng một mảng một chiều `possible[x]` cho biết liệu ta có thể chọn ra một tập con có tổng  $x$ . Mẹo là cập nhật mảng từ phải sang trái với mỗi trọng lượng mới:

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

Lưu ý rằng ý tưởng chung được đề cập ở đây có thể được áp dụng trong nhiều bài toán dạng cái túi. Ví dụ, nếu ta được cho trước các đối tượng cùng trọng lượng và giá trị, ta có thể tính được tổng giá trị lớn nhất của một tập con với tổng trọng lượng bất kì.

## 7.5 Khoảng cách chỉnh sửa

**Khoảng cách chỉnh sửa** hay **khoảng cách Levenshtein**<sup>1</sup> là số lượng tối thiểu các thao tác chỉnh sửa cần thiết để chuyển đổi một chuỗi thành một chuỗi khác. Những thao tác chỉnh sửa được cho phép là:

- chèn một kí tự (e.g.  $ABC \rightarrow ABCA$ )
- xóa một kí tự (e.g.  $ABC \rightarrow AC$ )
- chỉnh sửa một kí tự (e.g.  $ABC \rightarrow ADC$ )

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

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

Ta có thể tính 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  $x[a] = y[b]$ , ngược lại  $cost(a, b) = 1$ . Công thức này xét các cách chỉnh sửa chuỗi  $x$ :

<sup>1</sup>Khoảng cách được đặt tên theo V. I. Levenshtein, người đã nghiên cứu nó cùng với mã nhị phân [49].

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

Trong hai trường hợp đầu, chỉ cần một thao tác chỉnh sửa (chèn hoặc xóa). Trong trường hợp thứ ba, nếu  $x[a] = y[b]$ , ta có thể khớp hai kí tự cuối cùng mà không cần chỉnh sửa, ngược lại cần một thao tác chỉnh sửa (thay đổi).

Bảng sau cho thấy giá trị của  $\text{distance}$  trong ví dụ trên:

		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

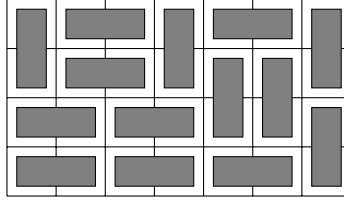
Góc dưới bên phải của bảng cho thấy khoảng cách chỉnh sửa giữa LOVE và MOVIE là 2. Bảng cũng cho thấy cách xây dựng chuỗi chỉnh sửa ngắn nhất. Trong trường hợp này, đường đi như sau:

		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

Kí tự cuối cùng của LOVE và MOVIE giống nhau, vì thế khoảng cách chỉnh sửa giữa chúng bằng khoảng cách chỉnh sửa giữa LOV và MOVI. Ta có thể sử dụng một phép chỉnh sửa để loại bỏ kí tự I khỏi MOVI. Do đó, khoảng cách chỉnh sửa lớn hơn một so với khoảng cách giữa LOV và MOV, v.v.

## 7.6 Counting tilings

Đôi lúc các trạng thái của một giải thuật quy hoạch động phức tạp hơn là các tổ hợp cố định của các số. Ví dụ, xét bài toán tính số cách khác nhau để điền vào một lưới  $n \times m$  bằng các ô gạch kích thước  $1 \times 2$  và  $2 \times 1$ . Ví dụ, một cách điền hợp lệ cho lưới  $4 \times 7$  là Sometimes the states of a Quy hoạch động solution are more complex than fixed combinations of numbers. As an example, consider the problem of calculating the number of distinct ways to fill an  $n \times m$  grid using  $1 \times 2$  and  $2 \times 1$  size tiles. For example, one valid solution for the  $4 \times 7$  grid is



và tổng số nghiệm là 781.

Bài toán này có thể giải được dùng quy hoạch động bằng cách duyệt qua từng dòng một của lưới. Mỗi dòng trong một cách lát có thể được biểu diễn dưới dạng một chuỗi gồm  $m$  kí tự từ tập  $\{\sqcup, \sqsubset, \sqcap, \sqsupset\}$ . Ví dụ, cách điền trên gồm 4 dòng tương ứng với những chuỗi sau:

- $\sqcap \sqsubset \sqsubset \sqcap \sqsubset \sqsubset \sqcap$
- $\sqcup \sqsubset \sqsubset \sqcup \sqcap \sqcap \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$

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

Một cách điền là hợp lệ nếu dòng 1 không chứa kí tự  $\sqcup$ , dòng  $n$  không chứa kí tự  $\sqcap$ , và mỗi hai dòng liên tiếp nhau là *tương thích*. Ví dụ, các dòng  $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup$  và  $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$  là tương thích, trong khi các dòng  $\sqcap \sqsubset \sqsubset \sqcap \sqsubset \sqsubset \sqcap$  và  $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$  là không tương thích.

Vì một dòng chứa  $m$  kí tự và có 4 lựa chọn cho mỗi kí tự, số lượng dòng khác nhau tối đa là  $4^m$ . Vì vậy, độ phức tạp thời gian của thuật toán là  $O(n4^{2m})$  vì ta có thể phải duyệt qua  $O(4^m)$  trạng thái có thể cho mỗi dòng, và với mỗi trạng thái, ta có  $O(4^m)$  trạng thái có thể cho dòng trước đó. Trong thực tế, ta nên xoay lưới sao cho cạnh ngắn hơn có độ dài  $m$ , vì hệ số  $4^{2m}$  chiếm đa số độ phức tạp thời gian.

Có thể cải thiện thuật toán bằng cách sử dụng một biểu diễn gọn hơn cho các dòng. Hóa ra ta chỉ cần biết những cột của dòng trước đó có chứa góc trên của một ô vuông dọc. Do đó, ta có thể biểu diễn một dòng chỉ bằng các kí tự  $\sqcap$  và  $\square$ , trong đó  $\square$  là một tổ hợp của các kí tự  $\sqcup$ ,  $\sqsubset$  và  $\sqsupset$ . Dùng cách biểu diễn này, chỉ có  $2^m$  dòng khác nhau nên độ phức tạp thời gian là  $O(n2^{2m})$ .

Một ghi chú cuối cùng, có một công thức trực tiếp khá bất ngờ để tính số cách điền<sup>2</sup>:

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

Công thức này rất nhanh, vì nó chỉ cần tính số cách điền trong độ phức tạp thời gian  $O(nm)$ , nhưng bởi vì kết quả là một tích của các số thực, một vấn

<sup>2</sup>Trùng hợp là, công thức này được phát hiện vào năm 1961 bởi hai nhóm nghiên cứu [43, 67] hoạt động độc lập.

đề khi sử dụng công thức này là làm sao để lưu trữ các kết quả trung gian một cách chính xác.



# Chương 8

## Phân tích khâu trừ

Độ phức tạp về 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 cách phân tích đơn giản không đưa ra một bức tranh chân thực về độ hiệu quả của thuật toán.

**Phân tích khâu trừ** có thể được dùng để phân tích những thuật toán có chứa các thao tác có độ phức tạp không cố định. Ý tưởng là ước lượng tổng thời gian dùng để thực hiện tất cả các phép tính như vậy trong quá trình thực hiện thuật toán, thay vì tập trung vào các phép tính riêng lẻ.

### 8.1 Phương pháp hai con trỏ

Trong **phương pháp hai con trỏ**, hai con trỏ được dùng để duyệt qua các phần tử 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 ta sẽ thảo luận hai bài toán có thể giải được bằng cách sử dụng phương pháp hai con trỏ.

#### Tổng mảng con

Ở ví dụ đầu tiên, xét một bài toán mà ta được cho một mảng gồm  $n$  số nguyên dương và một tổng đích  $x$ , và ta muốn tìm một mảng con có tổng là  $x$  hoặc thông báo 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 trong  $O(n)$  bằng cách sử dụng phương pháp hai con trỏ. Ý tưởng là duy trì các con trỏ mà trỏ đến giá trị đầu vào cuối của một mảng con. Ở mỗi lượt, con trỏ đầu di chuyển một bước sang phải, và

con trỏ cuối di chuyển sang phải chừng nào mà tổng mảng con vẫn không vượt quá  $x$ . Nếu tổng đúng bằng  $x$  thì mảng con được tìm thấy.

Ví dụ, xét mảng sau và một tổng đích  $x = 8$ :

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 đầu di chuyển một bước sang phải. Con trỏ cuối không di chuyển, bởi vì nếu ngược lại thì tổng mảng con sẽ vượt quá  $x$  ( $3 + 2 + 5 > 8$ ).

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

Lần nữa, con trỏ đầu di chuyển một bước sang phải và lần này con trỏ cuối di chuyển hai bước sang phải. Tổng 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 một thuật toán phụ thuộc vào số lượng bước mà con trỏ cuối di chuyển. Mặc dù không có giới hạn trên về số lượng bước mà con trỏ có thể di chuyển trong một lượt *duy nhất*. Nhưng ta biết rằng con trỏ di chuyển *tổng cộng*  $O(n)$  bước trong thuật toán bởi vì nó chỉ di chuyển sang phải.

Vì cả con trỏ đầu và cuối di chuyển  $O(n)$  bước trong thuật toán nên thuật toán hoạt động trong  $O(n)$ .

## Bài toán 2SUM

Một bài toán khác có thể được giải bằng cách sử dụng phương pháp hai con trỏ là bài toán sau, còn được biết với tên là **bài toán 2SUM**: cho một mảng gồm  $n$  số và một tổng đích  $x$ , tìm hai phần tử của mảng sao cho tổng của chúng là  $x$  hoặc thông báo rằng chúng không tồn tại.

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

Ví dụ, xét mảng sau và một tổng đích  $x = 12$ :



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

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

1	4	5	6	7	9	9	10
↑							↑

Sau đó con trỏ đầu di chuyển một bước sang phải. Con trỏ cuối 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 đầu di chuyển một bước sang phải. Con trỏ cuối không di chuyển và tổng trở thành  $5 + 7 = 12$ . Lúc này, lời giải 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 đầu tiên nó sắp xếp mảng trong  $O(n \log n)$  và sau đó cả hai con trỏ di chuyển  $O(n)$  bước.

Lưu ý rằng có thể giải bài toán theo một cách khác trong  $O(n \log n)$  thời gian bằng cách sử dụng tìm kiếm nhị phân. Trong cách giải này, ta duyệt qua mảng và đối với mỗi phần tử của mảng, ta cố gắng tìm một phần tử khác mà tổng là  $x$ . Ta có thể thực hiện  $n$  lần tìm kiếm nhị phân, mỗi lần mất  $O(\log n)$ .

Một bài toán khó hơn là **bài toán 3SUM**, bài toán yêu cầu tìm *ba* phần tử của mảng có tổng là  $x$ . Sử dụng ý tưởng của thuật toán trên, bài toán này có thể được giải trong  $O(n^2)$ <sup>1</sup>. Bạn biết cách làm không?

## 8.2 Phần tử nhỏ hơn gần nhất

Phân tích khâu trừ thường được dùng để ước lượng số lượng thao tác trên một cấu trúc dữ liệu nào đó. Các phép tính có thể được phân phối không đồng đều vì vậy hầu hết các phép tính xuất hiện trong một giai đoạn nhất định của thuật toán mà số lượng thao tác bị giới hạn.

Ví dụ, xét bài toán tìm **phần tử nhỏ hơn gần nhất** với mỗi phần tử của mảng, tức là, phần tử đầu tiên nhỏ hơn mà đứng trước phần tử đó trong mảng. Có thể không tồn tại phần tử nào như vậy, trong trường hợp đó thuật toán sẽ phải thông báo điều này. Tiếp theo ta sẽ xem cách mà bài toán có thể được giải hiệu quả bằng cấu trúc ngăn xếp.

<sup>1</sup>Trong một thời gian dài, người ta cho rằng việc giải bài toán 3SUM nhanh hơn  $O(n^2)$  là không thể. Tuy nhiên, năm 2014, người ta phát hiện ra [30] rằng điều này không còn đúng nữa.

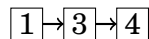
Ta duyệt qua mảng từ trái qua phải và duy trì một ngăn xếp gồm các phần tử của mảng. Tại mỗi vị trí mảng, ta loại bỏ các phần tử khỏi đỉnh ngăn xếp cho đến khi phần tử đỉnh nhỏ hơn phần tử hiện tại hoặc khi ngăn xếp đã rỗng. Sau đó, ta thông báo rằng phần tử ở đỉnh là phần tử nhỏ hơn gần nhất của phần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

Ví dụ, xét mảng sau:

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

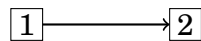
Đầu tiên, các phần tử 1, 3 và 4 được thêm vào ngăn xếp bởi vì mỗi phần tử đều lớn hơn phần tử đứng trước. Vì vậy, phần tử nhỏ hơn gần nhất của 4 là 3 và phần tử nhỏ hơn gần nhất của 3 là 1.

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



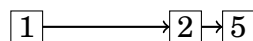
Phần tử tiếp theo là 2 nhỏ hơn hai phần tử ở đỉnh của ngăn xếp. Vì vậy, các phần tử 3 và 4 bị loại bỏ khỏi ngăn xếp và sau đó, phần tử 2 được thêm vào ngăn xếp. Phần tử nhỏ hơn gần nhất của nó là 1:

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



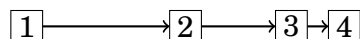
Sau đó, phần tử 5 lớn hơn phần tử 2, vì vậy nó sẽ được thêm vào ngăn xếp, và phần tử nhỏ hơn gần nhất của nó là 2:

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



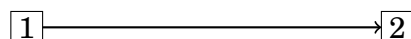
Sau đó, phần tử 3 bị loại bỏ khỏi ngăn xếp và các phần tử 3 và 4 được thêm vào ngăn xếp:

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



Cuối cùng, mọi phần tử trừ phần tử 1 bị loại bỏ khỏi ngăn xếp và phần tử cuối cùng phần tử 2 được thêm vào ngăn xếp:

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



Độ hiệu quả của thuật toán phụ thuộc vào tổng số lượng thao tác trên ngăn xếp. Nếu phần tử hiện tại lớn hơn phần tử ở đỉnh trong ngăn xếp, nó được thêm trực tiếp vào ngăn xếp, điều này được cho là hiệu quả. Tuy nhiên, đôi khi ngăn xếp có thể chứa vài phần tử lớn hơn và phải mất thời gian để loại bỏ chúng. Tuy vậy, mỗi phần tử được thêm *đúng một lần* vào ngăn xếp và bị loại bỏ *không quá một lần* khỏi ngăn xếp. Vì vậy, mỗi phần tử tạo ra  $O(1)$  thao tác trên ngăn xếp và thuật toán hoạt động trong  $O(n)$ .

### 8.3 Cực tiểu đoạn tịnh tiến

Một **đoạn tịnh tiến** là một mảng con có kích thước không đổi di chuyển từ trái sang phải trong mảng. Tại mỗi vị trí đoạn, ta muốn tính toán một số thông tin về các phần tử nằm trong đoạn. Trong phần này, ta tập trung vào bài toán về việc duy trì **cực tiểu đoạn tịnh tiến**, nghĩa là ta sẽ thông báo giá trị nhỏ nhất nằm trong mỗi đoạn.

Cực tiểu đoạn tịnh tiến có thể được tính toán bằng cách sử dụng một ý tưởng tương tự mà ta đã dùng để tính toán các phần tử nhỏ hơn gần nhất. Ta duy trì một hàng đợi mà trong đó mỗi phần tử lớn hơn phần tử đứng trước và phần tử đầu tiên luôn luôn tương ứng với phần tử nhỏ nhất nằm trong đoạn. Sau mỗi bước di chuyển, ta loại bỏ các phần tử khỏi cuối hàng đợi cho đến khi phần tử cuối nhỏ hơn phần tử mới hoặc khi hàng đợi rỗng. Ta cũng loại bỏ phần tử đầu tiên của hàng đợi nếu nó không nằm trong đoạn nữa. Cuối cùng, ta thêm phần tử mới vào cuối hàng đợi.

Ví dụ, xét mảng sau:

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

Giả sử kích thước của đoạn tịnh tiến là 4. Tại vị trí đoạn đầu tiên, phần tử nhỏ nhất là 1:

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

1 → 4 → 5

Sau đó, đoạn 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.

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

1 → 3

Sau đó, đoạn di chuyển lần nữa và phần tử nhỏ nhất 1 không nằm trong đoạn nữa. Vì vậy, nó bị loại bỏ khỏi hàng đợi và giá trị nhỏ nhất bây giờ là 3. Đồng thời phần tử mới 4 được thêm vào hàng đợi.

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

3 → 4

Phần tử tiếp theo 1 nhỏ hơn mọi phần tử trong hàng đợi. Vì vậy, mọi phần tử bị loại bỏ khỏi hàng đợi và nó sẽ chỉ chứa phần tử 1:

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

1

Cuối cùng, đoạn di chuyển đến vị trí cuối cùng. Phần tử 2 được thêm vào hàng đợi, nhưng giá trị nhỏ nhất nằm trong đoạn vẫn là 1.

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

1 → 2

Bởi vì mỗi phần tử của mảng được thêm vào hàng đợi chính xác một lần và bị loại bỏ khỏi hàng đợi không quá một lần nên thuật toán hoạt động trong  $O(n)$ .

## Chương 9

# Truy vấn trên đoạn

Trong chương này, chúng ta sẽ nói về các cấu trúc dữ liệu cho phép xử lý các truy vấn trên đoạn. Trong một **truy vấn trên đoạn**, nhiệm vụ của chúng ta là tính toán một giá trị dựa vào một mảng con của một mảng. Các truy vấn trên 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, xét đoạn  $[3, 6]$  trong dãy 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 trên đoạn là sử dụng vòng lặp duyệt hết tất cả các giá trị của mảng trong một đoạn. Ví dụ, hàm dưới đây có thể được sử dụng để xử lý các truy vấn tổng trên một đoạn:

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

Hàm này xử lý trong độ phức tạp  $O(n)$ , với  $n$  là kích thước của mảng. Vì vậy, chúng ta có thể xử lý  $q$  truy vấn trong độ phức tạp  $O(nq)$  bằng cách sử dụng hàm này. Tuy nhiên, nếu cả  $n$  và  $q$  đều lớn, cách tiếp cận này sẽ bị chậm. Nhưng may mắn là có những cách xử lý các truy vấn trên đoạn hiệu quả hơn rất nhiều.

## 9.1 Truy vấn mảng tĩnh

Chúng ta đầu tiên tập trung vào một tình huống khi mà mảng là *tĩnh*, tức là các giá trị của mảng không bao giờ được cập nhật ở các truy vấn. Trong trường hợp này, một cách đủ tốt đó là xây dựng một cấu trúc mảng tĩnh có thể cho chúng ta biết kết quả của các truy vấn có thể.

### 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**. Mỗi giá trị của mảng tổng tiền tố bằng tổng các giá trị trong mảng ban đầu cho đến vị trí đó, tức là, giá trị ở vị trí  $k$  là  $\text{sum}_q(0, k)$ . Mảng tổng tiền tố có thể được xây dựng trong độ phức tạp  $O(n)$ .

Ví dụ, xét mảng sau:

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

Mảng tổng tiền tố tương ứng như sau:

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

Bởi vì mảng tổng tiền tố chứa tất cả các giá trị của  $\text{sum}_q(0, k)$ , chúng ta có thể tính toán bất kỳ giá trị nào của  $\text{sum}_q(a, b)$  trong độ phức tạp  $O(1)$  như sau:

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

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

Ví dụ, xét đoạn  $[3, 6]$ :

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

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

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

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

Việc tổng quát hoá ý tưởng này lên chiều cao hơn là hoàn toàn khả thi. Ví dụ, chúng ta có thể xây dựng một mảng tổng tiền tố hai chiều và nó có thể được sử dụng để tính tổng của bất kỳ hình chữ nhật con nào trong độ

phức tạp  $O(1)$ . Mỗi tổng trong một mảng tương ứng với một đoạn con bắt đầu từ góc trái trên của mảng.

Hình ảnh dưới đây mô tả ý tưởng:

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

Tổng của mảng con màu xám được tính bằng công thức

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

với  $S(X)$  là tổng các giá trị trong một mảng con hình chữ nhật bắt đầu từ góc trái trên đến vị trí của  $X$ .

## Truy vấn giá trị nhỏ nhất

Các truy vấn giá trị nhỏ nhất khó xử lý hơn các truy vấn tổng. Mặc dù vậy, có một phương pháp tiền xử lý trong độ phức tạp  $O(n \log n)$  khá đơn giản và sau đó chúng ta có thể trả lời bất kỳ truy vấn giá trị nhỏ nhất nào trong độ phức tạp  $O(1)$ <sup>1</sup>. Lưu ý rằng bởi vì truy vấn giá trị nhỏ nhất và lớn nhất có thể được xử lý như nhau, chúng ta sẽ tập trung vào truy vấn giá trị nhỏ nhất.

Ý tưởng đó là tiền xử lý tất cả các giá trị của  $\min_q(a, b)$  với  $b - a + 1$  (độ dài của đoạn) là một lũy thừa của hai. Ví dụ, với mảng

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

các giá trị được tính như sau:

$a$	$b$	$\min_q(a, b)$	$a$	$b$	$\min_q(a, b)$	$a$	$b$	$\min_q(a, b)$
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						

<sup>1</sup>Kỹ thuật này đã được giới thiệu trong [7] và đôi khi được gọi là phương pháp **bảng thưa**. Ngoài ra còn có các kỹ thuật tinh vi hơn [22] với độ phức tạp tiền xử lý chỉ là  $O(n)$ , nhưng những thuật toán như vậy không cần thiết trong lập trình thi đấu.

Số lượng các giá trị được tiền xử lý là  $O(n \log n)$ , bởi vì có  $O(\log n)$  độ dài đoạn là các lũy thừa của hai. Các giá trị được tính toán hiệu quả bằng công thức truy hồi

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

void  $b - a + 1$  là một lũy thừa của hai và  $w = (b - a + 1)/2$ . Tính hết tất cả các giá trị với độ phức tạp  $O(n \log n)$ .

Sau đó, bất kỳ giá trị  $\min_q(a, b)$  nào đều có thể được tính trong độ phức tạp  $O(1)$  bằng giá trị nhỏ nhất của hai giá trị tiền xử lý. Gọi  $k$  là lũy thừa của hai lớn nhất 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 xem là hợp bởi đoạn  $[a, a + k - 1]$  và  $[b - k + 1, b]$ , cả hai đoạn đều có độ dài  $k$ .

Ví dụ, xét đoạn  $[1, 6]$ :

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

Độ dài của đoạn là 6, và lũy thừa của hai lớn nhất mà không vượt quá 6 là 4. Vì vậy đoạn  $[1, 6]$  là hợp bởi hai đoạn  $[1, 4]$  và  $[3, 6]$ :

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

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

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

## 9.2 Cây chỉ số nhị phân

Một **cây chỉ số nhị phân** hoặc **Fenwick tree**<sup>2</sup> có thể thấy một biến thể động của một mảng tổng tiền tố. Nó hỗ trợ hai thao tác độ phức tạp  $O(\log n)$  trên một mảng: xử lý truy vấn tổng trên một đoạn và cập nhật một giá trị.

Lợi ích của một cây chỉ số nhị phân đó là nó cho phép chúng ta cập nhật hiệu quả các giá trị của mảng giữa các truy vấn tính tổng. Điều này là không khả thi khi dùng mảng tổng tiền tố, bởi vì sau mỗi lần cập nhật, mảng tổng tiền tố cần phải được xây dựng lại toàn bộ trong độ phức tạp  $O(n)$ .

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



## Cấu trúc

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

Gọi  $p(k)$  là lũy thừa của hai lớn nhất là ước của  $k$ . Ta lưu một cây chỉ số nhị phân như một mảng *tree* sao cho

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

Tức là, mỗi vị trí  $k$  chứa tổng của 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  $\text{sum}_q(5, 6)$ .

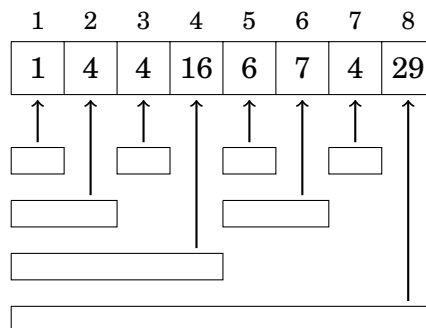
Ví dụ, xét mảng sau:

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

Cây chỉ số nhị phân tương ứng như sau:

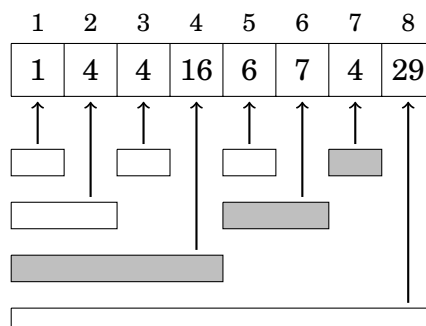
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

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



Sử dụng cây chỉ số nhị phân, bất kỳ giá trị của  $\text{sum}_q(1, k)$  có thể được tính trong độ phức tạp  $O(\log n)$ , bởi vì một đoạn  $[1, k]$  luôn có thể được chia thành  $O(\log n)$  đoạn với các tổng được lưu trong cây.

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



Vì vậy, ta có thể tính giá trị tổng tương ứng như sau:

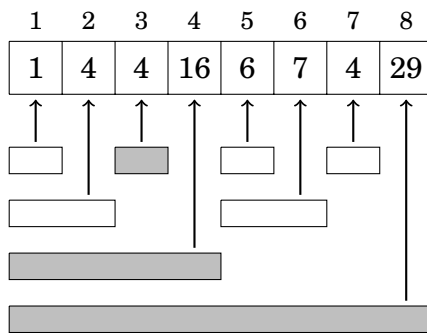
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

để tính giá trị  $\text{sum}_q(a, b)$  với  $a > 1$ , ta có thể sử dụng kĩ thuật đã dùng trong mảng tổng tiền tố:

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

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

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



Bởi vì mỗi phần tử của mảng thuộc  $O(\log n)$  đoạn trong cây chỉ số nhị phân, các giá trị trong cây được cập nhật trong độ phức tạp  $O(\log n)$ .

## Cài đặt

Các phép tính trong một cây chỉ số nhị phân có thể được cài đặt hiệu quả bằng các phép tính bit. Điểm mấu chốt đó là chúng ta có thể tính bất kỳ giá trị  $p(k)$  bằng công thức

$$p(k) = k \& -k.$$

Hàm sau đây 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 đây tăng giá trị của mảng tại vị trí  $k$  lên  $x$  ( $x$  có thể dương hoặc âm):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

Độ phức tạp tính toán của cả hai hàm là  $O(\log n)$ , bởi vì các hàm truy cập đến  $O(\log n)$  giá trị của cây chỉ số nhị phân, và mỗi bước chuyển đến vị trí tiếp theo tốn độ phức tạp  $O(1)$ .

### 9.3 Cây phân đoạn

Một **cây phân đoạn**<sup>3</sup> 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 phân đoạn có thể hỗ trợ các truy vấn tính tổng, truy vấn giá trị nhỏ nhất và lớn nhất và nhiều truy vấn khác mà cả hai thao tác đều làm việc trong độ phức tạp  $O(\log n)$ .

So sánh với một cây chỉ số nhị phân, lợi ích của một cây phân đoạn đó là nó là một cấu trúc dữ liệu tổng quát hơn. Trong khi những cây chỉ số nhị phân chỉ có thể hỗ trợ các truy vấn tính tổng<sup>4</sup>, các cây phân đoạn còn hỗ trợ các truy vấn khác. Mặt khác, một cây phân đoạn yêu cầu nhiều bộ nhớ hơn và khó hơn một chút trong việc cài đặt.

#### Cấu trúc

Một cây phân đoạn là một cây nhị phân mà các nút ở bậc thấp nhất của cây ứng với các phần tử của mảng, và các nút khác chứa thông tin cần thiết cho việc xử lý các truy vấn trên đoạn.

Trong phần này, chúng ta giả định rằng kích thước của mảng là một lũy thừa của hai và được đánh chỉ số từ 0, bởi vì điều này sẽ tiện hơn trong việc xây dựng một cây phân đoạn cho mảng đó. Nếu kích thước của mảng không phải là một lũy thừa của hai, ta luôn có thể thêm các phần tử phụ vào cuối nó.

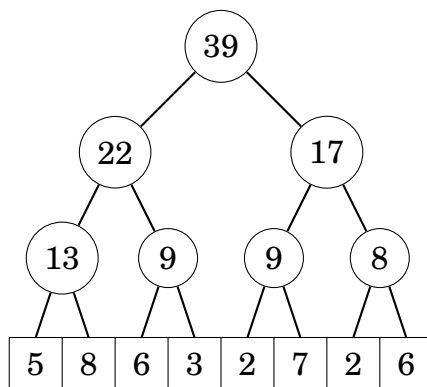
Ta sẽ đầu tiên nói về những cây phân đoạn hỗ trợ các truy vấn tổng. Ví dụ, xét mảng sau:

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

Cây phân đoạn tương ứng như sau:

<sup>3</sup>Cách cài đặt bottom-up trong chương này được đề cập trong [62]. Giống với các cấu trúc được sử dụng vào cuối những năm 1970 để giải quyết các vấn đề hình học [?].

<sup>4</sup>Thực ra, sử dụng *hai* cây chỉ số nhị phân có thể hỗ trợ các truy vấn giá trị nhỏ nhất [16], nhưng nó phức tạp hơn việc sử dụng một cây phân đoạn.

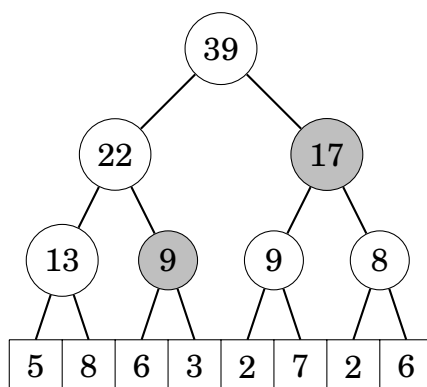


Mỗi nút bên trong cây tương ứng với một đoạn của mảng có kích thước là một lũy thừa của hai. Trong cây ở trên, giá trị của mỗi nút bên trong là một tổng của các giá trị tương ứng của mảng, và nó có thể được tính bằng tổng của giá trị của nút con trái và phải của nó.

Nhận thấy rằng bất kỳ đoạn  $[a, b]$  nào có thể được chia thành  $O(\log n)$  đoạn có các giá trị được lưu vào các nút của cây. Ví dụ, xét đoạn  $[2, 7]$ :

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

Ở đây  $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$ . Trong trường hợp này, hai nút sau của cây tương ứng với đoạn:

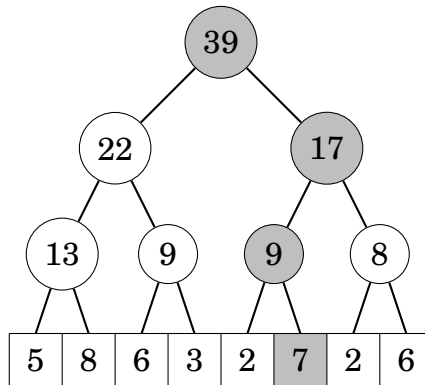


Vì vậy, một cách khác để tính tổng là  $9 + 17 = 26$ .

Khi tổng được tính bằng các nút ở vị trí cao nhất có thể ở trong cây, có tối đa hai nút trên mỗi bậc của cây là được cần tới. Vì vậy, tổng số nút là  $O(\log n)$ .

Sau một lần cập nhật mảng, ta cần cập nhật hết các nút có giá trị phụ thuộc vào giá trị được cập nhật. Điều này có thể làm được bằng cách duyệt trên đường đi từ phần tử được cập nhật của mảng đến nút trên cùng và cập nhật các nút trên đường đi.

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

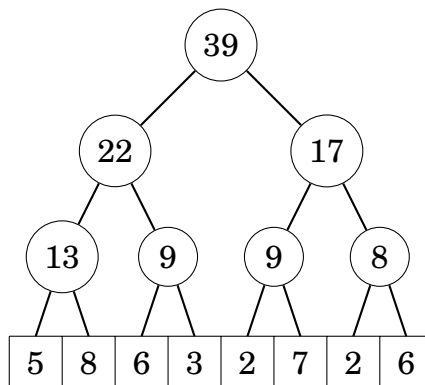


Đường đi từ vị trí thấp nhất và cao nhất luôn chứa  $O(\log n)$  nút, vì vậy mỗi lần cập nhật thay đổi  $O(\log n)$  nút trong cây.

## Cài đặt

Ta lưu một cây phân đoạn như một mảng gồm  $2n$  phần tử với  $n$  là kích thước của mảng ban đầu và là một lũy thừa của hai. Các nút của cây được lưu từ đầu đến cuối:  $tree[1]$  là nút trên cùng,  $tree[2]$  và  $tree[3]$  là các con của nó, và cứ như thế. Cuối cùng, các giá trị từ  $tree[n]$  đến  $tree[2n - 1]$  tương ứng với các giá trị của mảng ban đầu ở bậc thấp nhất của cây.

Ví dụ, cây phân đoạn



được lưu như sau:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Sử dụng các biểu diễn này, cha của  $tree[k]$  là  $tree[\lfloor k/2 \rfloor]$ , và các con của nó là  $tree[2k]$  và  $tree[2k + 1]$ . Lưu ý rằng điều này suy ra rằng vị trí của một nút là chẵn nếu nó là con trái và lẻ nếu nó là con phải.

Hàm sau đây tính giá trị của  $sum_q(a, b)$ :

```
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 dịch chuyển đến bậc cao hơn trong cây, và trước đó, các giá trị của các nút không thuộc về đoạn cao hơn và cộng vào tổng.

Hàm sau đây 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];
    }
}

```

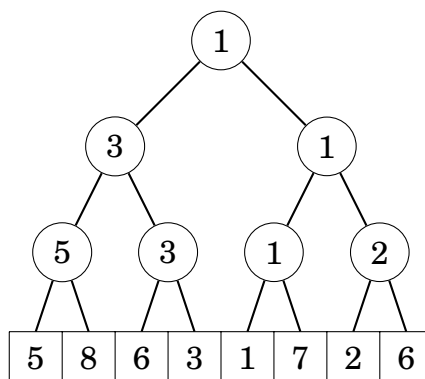
Đầu tiên hàm cập nhật giá trị ở bậc thấp nhất của cây. Sau đó, hàm cập nhật các giá trị của tất cả các nút bên trong cây, cho đến khi nó đến được nút trên cùng của cây.

Cả hai hàm trên làm việc trong độ phức tạp  $O(\log n)$ , bởi vì một cây phân đoạn gồm  $n$  nút chứa  $O(\log n)$  bậc, và các hàm dịch lên một bậc cao hơn ở trong cây tại mỗi bước.

## Các truy vấn khác

Cây phân đoạn có thể hỗ trợ tất cả các truy vấn trên đoạn mà nó có thể được chia từ một đoạn thành hai thành phần, tính kết quả độc lập cho cả hai phần và sau đó kết hợp một cách hiệu quả các kết quả. Ví dụ của những truy vấn đó là giá trị nhỏ nhất, lớn nhất, ước chung lớn nhất, và các phép tính trên bit and, or và xor.

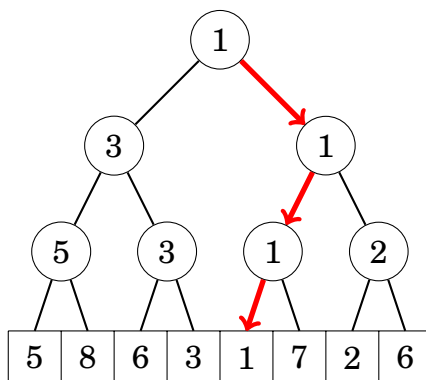
Ví dụ, cây phân đoạn sau đây hỗ trợ các truy vấn giá trị nhỏ nhất:



Trong trường hợp này, tất cả các nút của cây chứa giá trị nhỏ nhất trong đoạn tương ứng của mảng. Nút trên cùng của cây chứa giá trị nhỏ nhất của toàn bộ mảng. Các thao tác có thể được cài đặt như trước đó, nhưng thay vì tổng, giá trị nhỏ nhất được tính.

Cấu trúc của một cây phân đoạn còn cho phép ta sử dụng chặt nhị phân cho các phần tử ở mảng. Ví dụ, nếu cây hỗ trợ các truy vấn giá trị nhỏ nhất, ta có thể tìm vị trí của một phần tử có giá trị nhỏ nhất trong độ phức tạp  $O(\log n)$ .

Ví dụ, trong cây ở trên, một phần tử có giá trị nhỏ nhất 1 có thể được tìm thấy bằng cách duyệt đường đi hướng xuống từ nút trên cùng:



## 9.4 Các kỹ thuật bổ sung

### Nén chỉ số

Một giới hạn trong các cấu trúc dữ liệu được xây dựng dựa trên một mảng đó là các phần tử được đánh chỉ số bằng các số nguyên liên tiếp. Khó khăn xảy ra khi các chỉ số lớn được cần đến. Ví dụ, nếu ta muốn sử dụng chỉ số  $10^9$ , mảng cần chứa  $10^9$  phần tử yêu cầu một lượng lớn bộ nhớ.

Tuy nhiên, ta thường có thể vượt qua giới hạn này bằng cách **nén chỉ số**, với các chỉ số ban đầu được thay thế bằng các chỉ số 1, 2, 3, v.v. Điều này có thể làm được nếu ta biết trước hết tất cả các chỉ số cần dùng trong thuật toán.

Ý tưởng đó là thay thế mỗi chỉ số ban đầu  $x$  bằng  $c(x)$  với  $c$  là một hàm nén các chỉ số. Ta cần thứ tự của các chỉ số không thay đổi, nên nếu  $a < b$ , thì  $c(a) < c(b)$ . Điều này cho phép ta thực hiện 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, ta đã cài đặt các cấu trúc dữ liệu hỗ trợ các truy vấn trên đoạn và cập nhật các giá trị riêng lẻ. Bây giờ ta xét một tình huống ngược lại, khi mà ta cần cập nhật các đoạn và lấy các giá trị riêng lẻ. Ta tập trung vào một thao tác tăng tất cả các phần tử trong một đoạn  $[a, b]$  lên  $x$ .

Thật ngạc nhiên, ta có thể sử dụng các cấu trúc dữ liệu được trình bày trong chương này cho tình huống này. Để làm được điều này, ta xây dựng một **mảng chênh lệch** có các giá trị chỉ ra độ chênh lệch giữa hai giá trị liên tiếp trong mảng ban đầu. Vì vậy, mảng ban đầu là mảng tổng tiền tố của dãy chênh lệch. Ví dụ, xét mảng sau đây:

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

Mảng chênh lệch của mảng trên là 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 chênh lệch.

Lợi ích của mảng chênh lệch đó là ta có thể cập nhật một đoạn trong mảng ban đầu bằng cách thay đổi chỉ hai phần tử trong mảng chênh lệch. Ví dụ, nếu ta muốn tăng các giá trị ở giữa vị trí 1 và 4 của mảng ban đầu lên 5, nó là đủ để tăng giá trị của mảng chênh lệch ở vị trí 1 lên 5 và giảm giá trị ở vị trí 5 xuống 5. Kết quả là 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  $x$ , ta tăng giá trị tại vị trí  $a$  lên  $x$  và giảm giá trị tại vị trí  $b + 1$  xuống  $x$ . Vì vậy, ta chỉ cần cập nhật các giá trị riêng lẻ và xử lý các truy vấn tính tổng, vậy nên ta có thể sử dụng một cây chỉ số nhị phân hoặc một cây phân đoạn.

Một vấn đề khó hơn đó là hỗ trợ cho cả các truy vấn trên đoạn và cập nhật đoạn. Ở chương 28 ta sẽ thấy điều này hoàn toàn khả thi.





Ví dụ, biểu diễn bit của số  $-43$  thuộc kiểu `int` là

1111111111111111111111111111010101.

Biểu diễn không dấu chỉ có thể lưu được các số không âm, bù lại cận trên của các giá trị được tăng lên so với biểu diễn có dấu. Một biến không dấu  $n$  bit có thể lưu được các số nguyên trong phạm vi từ  $0$  đến  $2^n - 1$ . Ví dụ, trong C++, một biến `unsigned int` có thể lưu các số nguyên trong phạm vi từ  $0$  đến  $2^{32} - 1$ .

Hai cách biểu diễn trên có một sự liên quan đến nhau: một số  $-x$  có dấu tương đương với một số  $2^n - x$  không dấu. Ví dụ, đoạn code sau thể hiện một số  $x = -43$  có dấu tương đương với một số  $y = 2^{32} - 43$  không dấu.

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

Nếu một số lớn hơn cận trên của biểu diễn nhị phân, thì số đó sẽ bị tràn. Trong một biểu diễn có dấu, số liền sau của  $2^{n-1} - 1$  là  $-2^{n-1}$ , và trong một biểu diễn không dấu, số liền sau của  $2^n - 1$  là  $0$ . Ví dụ, xét đoạn code sau:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Ban đầu, giá trị khởi tạo của  $x$  là  $2^{31} - 1$ . Đây là giá trị lớn nhất mà một biến `int` có thể lưu được, do đó giá trị tiếp theo của  $2^{31} - 1$  là  $-2^{31}$ .

## 10.2 Các phép toán trên bit

### Toán tử `and`

Toán tử **and**  $x \& y$  trả về một số mà vị trí của các bit 1 tương ứng với các vị trí mà cả  $x$  và  $y$  đều có bit 1. Ví dụ,  $22 \& 26 = 18$ , vì

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

Sử dụng toán tử `and`, ta có thể kiểm tra tính chẵn lẻ của một số  $x$ , vì  $x \& 1 = 0$  nếu  $x$  chẵn, và  $x \& 1 = 1$  nếu  $x$  lẻ. Tổng quát hơn,  $x$  chia hết cho  $2^k$  khi và chỉ khi  $x \& (2^k - 1) = 0$ .

## Toán tử or

Toán tử **or**  $x \mid y$  trả về một số mà các vị trí bit 1 tương đương với các vị trí mà  $x$  hoặc  $y$  có bit 1. Ví dụ,  $22 \mid 26 = 30$ , vì

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

## Toán tử xor

Toán tử **xor**  $x \wedge y$  trả về một số mà các vị trí bit 1 tương đương với các vị trí mà chính xác một trong hai số  $x$  và  $y$  có bit 1. Ví dụ,  $22 \wedge 26 = 12$ , vì

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

## Toán tử not

Toán tử **not**  $\sim x$  trả về một số mà tất cả bit của  $x$  được đảo lại, cụ thể, theo công thức  $\sim x = -x - 1$ . Ví dụ,  $\sim 29 = -30$ .

Kết quả của toán tử not ở cấp độ bit phụ thuộc vào độ dài của biểu diễn nhị phân, bởi vì toán tử sẽ thao tác đảo tất cả các bit. Ví dụ, nếu số là một số int 32-bit, kết quả sẽ như sau:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000000011101 \\ \sim x & = & -30 \quad 111111111111111111111111111110010 \end{array}$$

## Dịch chuyển bit

Phép dịch trái bit  $x \ll k$  chèn thêm  $k$  bit 0 vào cuối số, và phép dịch phải bit  $x \gg k$  bỏ đi  $k$  bit cuối của số. Ví dụ,  $14 \ll 2 = 56$ , bởi vì 14 và 56 tương đương với 1110 và 111000. Tương tự,  $49 \gg 3 = 6$ , vì 49 và 6 tương đương với 110001 và 110.

Để ý rằng  $x \ll k$  tương đương với việc nhân  $x$  với  $2^k$ , và  $x \gg k$  tương đương với việc lấy phần nguyên của phép chia  $x$  cho  $2^k$ .

## Các ứng dụng

Một số có dạng  $1 \ll k$  có một bit 1 tại vị trí  $k$  và các bit còn lại đều là 0, vì thế ta có thể dùng các số như vậy để lấy giá trị từng bit trong một số nào đó. Cụ thể, bit thứ  $k$  của một số là 1 khi và chỉ khi  $x \& (1 \ll k)$  khác 0. Đoạn code sau in biểu diễn nhị phân của một biến  $x$  thuộc kiểu int:

```
for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}
```

Ta cũng có thể chỉnh sửa từng bit của một số theo ý tưởng tương tự. Ví dụ, công thức  $x \mid (1 \ll k)$  sẽ gán bit thứ  $k$  của  $x$  thành 1, công thức  $x \& \sim(1 \ll k)$  sẽ gán bit thứ  $k$  của  $x$  thành 0, và công thức  $x \wedge (1 \ll k)$  sẽ đảo bit thứ  $k$  của  $x$ .

Công thức  $x \& (x - 1)$  gán bit 1 thấp nhất của  $x$  thành 0, và công thức  $x \& -x$  gán tất cả bit 1 thành 0, trừ bit 1 cuối cùng. Công thức  $x \mid (x - 1)$  đảo tất cả các bit sau bit 1 cuối cùng. Lưu ý rằng một số dương  $x$  là một lũy thừa của 2 khi và chỉ khi  $x \& (x - 1) = 0$ .

## Các hàm bổ sung

Trình biên dịch g++ cung cấp một số hàm sau đây:

- `__builtin_clz(x)`: trả về số lượng chữ số 0 ở đầu biểu diễn
- `__builtin_ctz(x)`: trả về số lượng chữ số 0 ở cuối biểu diễn
- `__builtin_popcount(x)`: trả về số bit 1 trong biểu diễn
- `__builtin_parity(x)`: tính chẵn lẻ của số lượng bit 1 trong biểu diễn

Các hàm có thể được gọi như sau:

```
int x = 5328; // 000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Các hàm trên chỉ hỗ trợ kiểu biến `int`. Tuy nhiên, ta có thể gọi phiên bản `long` của hàm bằng cách thêm hậu tố `ll` vào tên hàm.

## 10.3 Biểu diễn tập hợp

Mỗi tập con của một tập hợp  $\{0, 1, 2, \dots, n - 1\}$  có thể được biểu diễn bằng một số nguyên  $n$  bit với mỗi bit 1 thể hiện phần tử có xuất hiện trong tập con hay không. Đây là một cách hiệu quả để biểu diễn các tập hợp, do tất cả phần tử chỉ cần một bit trong bộ nhớ và các thao tác trên tập hợp có thể được đưa về cài đặt bằng các thao tác trên bit.

Ví dụ, kiểu `int` là một kiểu dữ liệu 32-bit, một số `int` có thể biểu diễn bất kì tập con nào của tập hợp  $\{0, 1, 2, \dots, 31\}$ . Biểu diễn nhị phân của tập  $\{1, 3, 4, 8\}$  là

000000000000000000000000100011010,

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

## Cài đặt tập hợp

Đoạn code sau khai báo một biến `int x` có thể biểu diễn một tập con của tập  $\{0, 1, 2, \dots, 31\}$ . Sau đó, đoạn code 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 code sau in ra tất cả các phần tử thuộc tập hợp

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

## Thao tác trên tập hợp

Các thao tác trên tập hợp có thể được cài đặt tương tự như các thao tác trên bit:

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

Ví dụ, đoạn code sau ban đầu tạo các tập hợp  $x = \{1, 3, 4, 8\}$  và  $y = \{3, 6, 8, 9\}$ , sau đó tạo tập  $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$ :

```
int x = (1<<1) | (1<<3) | (1<<4) | (1<<8);
int y = (1<<3) | (1<<6) | (1<<8) | (1<<9);
int z = x | y;
cout << __builtin_popcount(z) << "\n"; // 6
```

## Duyệt các tập con

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

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

Đoạn code sau duyệt qua các tập con có đúng  $k$  phần tử:

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

Đoạn code sau duyệt qua các tập con của một tập  $x$ :

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

## 10.4 Tối ưu bit

Đa số các thuật toán có thể được tối ưu bằng xử lý bit. Những cách tối ưu này không làm thay đổi độ phức tạp thuật toán, tuy nhiên chúng có thể có tác động không nhỏ vào thời gian chạy của chương trình. Trong phần này ta sẽ tìm hiểu về các ví dụ của cách tối ưu trên.

### Khoảng cách Hamming

**Khoảng cách Hamming** giữa hai chuỗi  $a$  và  $b$  cùng độ dài là số lượng vị trí mà kí tự giữa hai chuỗi khác nhau. Ví dụ,

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

Xét bài toán sau: cho một danh sách  $n$  xâu nhị phân độ dài  $k$ , hãy tính khoảng cách Hamming nhỏ nhất giữa hai xâu bất kì trong danh sách. Ví dụ, đáp án cho danh sách  $[00111, 01101, 11110]$  là 2, bởi vì

- $\text{hamming}(00111, 01101) = 2$ ,
- $\text{hamming}(00111, 11110) = 3$ , và
- $\text{hamming}(01101, 11110) = 3$ .

Lời giải trực quan nhất cho bài toán chính là duyệt qua tất cả các cặp xâu và tính khoảng cách Hamming giữa chúng, dẫn đến độ phức tạp thời gian  $O(n^2k)$ . Hàm trong đoạn code sau đây có thể được dùng để tính khoảng cách:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

Tuy nhiên, nếu  $k$  nhỏ, ta có thể tối ưu đoạn code bằng cách lưu xâu nhị phân vào các số nguyên, rồi dùng các thao tác trên bit để tính khoảng cách Hamming. Cụ thể, nếu  $k \leq 32$ , ta có thể lưu các xâu vào các biến `int` rồi dùng hàm sau để tính khoảng cách:

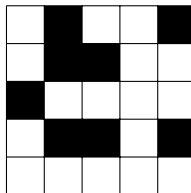
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

Trong hàm trên, toán tử xor trả về một xâu nhị phân có các bit 1 ở vị trí mà  $a$  và  $b$  khác nhau. Sau đó, số bit 1 được tính thông qua hàm `__builtin_popcount`.

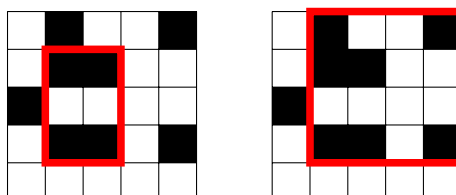
Để so sánh hai cách cài đặt, ta sinh một danh sách ngẫu nhiên 10000 xâu nhị phân độ dài 30. Với đoạn code đầu tiên, chương trình mất 13.5 giây để tính, sau khi được tối ưu bằng bit, chương trình chỉ còn chạy trong 0.5 giây. Do đó, đoạn code sử dụng bit để tối ưu chạy nhanh gấp khoảng 30 lần so với đoạn code trước đó.

## Đếm hình chữ nhật con

Một ví dụ khác, xét bài toán sau: Cho một lưới ô vuông kích thước  $n \times n$  trong đó mỗi ô được tô màu đen (1) hoặc trắng (0), hãy tính số lượng hình chữ nhật con trên lưới sao cho bốn góc hình chữ nhật đều được tô đen. Ví dụ, lưới ô vuông sau



có hai hình chữ nhật con thỏa mãn:



Ta có thể giải bài toán trong  $O(n^3)$  như sau: Duyệt qua tất cả  $O(n^2)$  cặp dòng, với mỗi cặp  $(a, b)$  đếm số lượng cột có ô đen ở cả hai dòng trong  $O(n)$ . Đoạn code sau giả sử `color[y][x]` là màu của ô vuông ở dòng  $y$  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, các cột tìm được sẽ cho ra `count(count - 1)/2` hình chữ nhật với các góc màu đen, do ta có thể chọn bất kì hai cột khác nhau trong số đó để tạo thành một hình chữ nhật thỏa mãn.

Để cải tiến thuật toán, ta chia lưới ô vuông thành các nhóm cột, mỗi nhóm gồm  $N$  cột liên tiếp. Sau đó, lưu mỗi dòng thành một danh sách các số nguyên  $N$ -bit biểu diễn cho màu của các ô vuông. Giờ ta có thể xử lí  $N$  cột trong cùng một lúc bằng xử lí bit. Trong đoạn code sau, `color[y][k]` biểu diễn màu của một nhóm  $N$  màu dưới dạng bit.

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

Độ phức tạp của thuật toán chỉ còn  $O(n^3/N)$ .

Ta sinh một lưới ô vuông ngẫu nhiên kích thước  $2500 \times 2500$  để so sánh hai cách cài đặt trên. Đoạn code đầu tiên chạy trong 29.6 giây, trong khi đó, đoạn code sử dụng cách tối ưu bit cho thời gian chạy là 3.1 giây với  $N = 32$  (kiểu `int`) và 1.7 giây với  $N = 64$  (kiểu `long long`).

## 10.5 Quy hoạch động

Xử lí bit giúp ta dễ dàng cài đặt các thuật toán quy hoạch động có trạng thái là một tập con của tập các phần tử, vì các trạng thái này có thể được lưu vào các số nguyên. Tiếp theo, ta sẽ tìm hiểu về cách ứng dụng xử lí bit vào quy hoạch động.

### Lựa chọn tối ưu

Xét bài toán sau: Cho danh sách giá tiền của  $k$  món hàng qua  $n$  ngày. Mỗi ngày bạn được phép mua tối đa một món. Hãy xác định chi phí tối thiểu để mua mỗi món hàng đúng một lần. Xét test ví dụ sau ( $k = 3$  và  $n = 8$ ):

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



Trong test ví dụ trên, chi phí tối thiểu là 5:

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

Gọi  $\text{price}[x][d]$  là giá của món hàng  $x$  vào ngày  $d$  (chẳng hạn, trong test ví dụ trên  $\text{price}[2][3] = 7$ ). Sau đó, gọi  $\text{total}(S, d)$  là chi phí tối thiểu để mua một tập con  $S$  của các món hàng sau  $d$  ngày. Khi đó, đáp án cho bài toán là  $\text{total}(\{0 \dots k-1\}, n-1)$ .

Ban đầu,  $\text{total}(\emptyset, d) = 0$ , do một tập rỗng không mất chi phí nào, và  $\text{total}(\{x\}, 0) = \text{price}[x][0]$ , do chỉ có một cách để mua một món hàng vào ngày đầu tiên. Khi đó, ta có công thức truy hồi sau:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

Ở công thức trên, ta có thể lựa chọn không mua hàng vào ngày  $d$ , hoặc mua một món  $x$  thuộc tập  $S$ . Trong trường hợp thứ hai, ta cần loại  $x$  khỏi tập  $S$  và cộng thêm giá của món  $x$  vào tổng chi phí.

Bước tiếp theo, ta đi tính giá trị của hàm bằng quy hoạch động. Ta lưu giá trị của hàm vào một mảng hai chiều

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

với  $K$  và  $N$  là các hằng số vừa đủ to. Chiều thứ nhất của mảng dùng để lưu biểu diễn nhị phân của từng tập con.

Đầu tiên, trường hợp  $d = 0$ , ta có thể xử lí như sau:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Then, the recurrence translates into the following code:

```
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 trên là  $O(n2^k k)$ .

## Từ hoán vị đến tập con

Với quy hoạch động, ta thường có thể chuyển cách duyệt một hoán vị về duyệt các tập con<sup>1</sup>. Ưu điểm của việc này là số lượng hoán vị,  $n!$ , lớn hơn nhiều so với số lượng tập con,  $2^n$ . Ví dụ, nếu  $n = 20$ , thì  $n! \approx 2.4 \cdot 10^{18}$  và  $2^n \approx 10^6$ . Vì vậy, việc duyệt qua các tập con luôn hiệu quả hơn duyệt các hoán vị.

Xét bài toán sau: Cho một chiếc thang máy với sức chịu tải đa là  $x$ , và  $n$  người biết trước cân nặng muốn đi từ tầng trệt lên tầng cao nhất. Hãy xác định thứ tự lên thang máy tối ưu sao cho số lượt lên thang máy là ít nhất.

Ví dụ, giả sử  $x = 10$ ,  $n = 5$  và cân nặng của mọi người là như sau:

người	cân nặng
0	2
1	3
2	3
3	5
4	6

Trong trường hợp này, số lượt lên thang máy ít nhất là 2. Một thứ tự tối ưu là  $\{0, 2, 3, 1, 4\}$ , với lượt thứ nhất gồm những người  $\{0, 2, 3\}$  (tổng cân nặng là 10), và lượt hai gồm  $\{1, 4\}$  (tổng cân nặng là 9).

Bài toán có thể được giải một cách đơn giản trong  $O(n!n)$  bằng cách duyệt qua tất cả các hoán vị của  $n$  người. Tuy nhiên, ta có thể dùng quy hoạch động để đạt độ phức tạp tốt hơn là  $O(2^n n)$ . Ý tưởng đó là ta tính cho mỗi tập con hai giá trị: số lượt lên thang máy tối thiểu và tổng cân nặng nhỏ nhất của nhóm cuối cùng lên thang máy.

Gọi  $\text{weight}[p]$  là cân nặng của người  $p$ . Ta định nghĩa hai hàm:  $\text{rides}(S)$  là số lượt lên thang máy cho một tập con  $S$ , và  $\text{last}(S)$  là tổng cân nặng tối thiểu của nhóm cuối cùng. Ví dụ, trong bộ dữ liệu phía trên

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

vì số lượt tối ưu là  $\{1, 4\}$  và  $\{3\}$ , và lượt cuối có tổng cân nặng là 5. Tất nhiên, mục tiêu cuối cùng của chúng ta là tính giá trị của  $\text{rides}(\{0 \dots n-1\})$ .

Ta đi xác định công thức truy hồi của hàm và cài đặt quy hoạch động. Ý tưởng đó là ta duyệt qua những người thuộc tập con  $S$  và chọn người  $p$  - người cuối cùng lên thang máy một cách tối ưu. Mỗi lựa chọn sẽ dẫn đến một bài toán con với kích thước nhỏ hơn. Nếu  $\text{last}(S \setminus p) + \text{weight}[p] \leq x$ , ta có thể thêm người  $p$  vào lượt đi thang máy cuối cùng. Ngược lại, ta phải thêm một lượt mới, ban đầu chỉ gồm người  $p$ .

Khi cài đặt quy hoạch động, ta khai báo một mảng

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

sao cho mỗi tập con  $S$  lưu một pair  $(\text{rides}(S), \text{last}(S))$ . Ta khởi tạo giá trị cho tập rỗng như sau:

<sup>1</sup>This technique was introduced in 1962 by M. Held and R. M. Karp [34].

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

Sau đó, ta có thể điền phần còn lại của mảng như sau:

```
for (int s = 1; s < (1<<n); s++) {  
    // initial value: n+1 rides are needed  
    best[s] = {n+1,0};  
    for (int p = 0; p < n; p++) {  
        if (s&(1<<p)) {  
            auto option = best[s^(1<<p)];  
            if (option.second+weight[p] <= x) {  
                // add p to an existing ride  
                option.second += weight[p];  
            } else {  
                // reserve a new ride for p  
                option.first++;  
                option.second = weight[p];  
            }  
            best[s] = min(best[s], option);  
        }  
    }  
}
```

Lưu ý rằng vòng lặp ở trên đảm bảo với mỗi cặp tập con  $S_1$  và  $S_2$  sao cho  $S_1 \subset S_2$ , ta xét  $S_1$  trước khi xét  $S_2$ . Do đó, các giá trị quy hoạch động được tính theo đúng thứ tự.

## Đếm tập con

Bài toán cuối cùng của chương này được phát biểu như sau: Gọi  $X = \{0 \dots n-1\}$ , và mỗi tập con  $S \subset X$  được dán nhãn bằng một số nguyên  $\text{value}[S]$ . Với mỗi tập con  $S$ , hãy tính

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

, tức tổng giá trị các nhãn của các tập con của  $S$ .

Ví dụ, giả sử  $n = 3$  và các nhãn có 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 trên, ta có

$$\begin{aligned} \text{sum}(\{0,2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

Bởi vì có tổng cộng  $2^n$  tập con, ta có thể giải bằng cách duyệt qua tất cả các cặp tập con trong  $O(2^{2n})$ . Tuy nhiên, với quy hoạch động, ta có thể giải bài toán trong  $O(2^n n)$ . Ý tưởng đó là ràng buộc lại các phần tử có thể bỏ khỏi  $S$ .

Gọi  $\text{partial}(S, k)$  là tổng giá trị của các tập con của  $S$  với ràng buộc là chỉ có các phần tử  $0 \dots k$  có thể được loại bỏ khỏi  $S$ .

Ví dụ,

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

vì ta chỉ có thể bỏ các phần tử  $0 \dots 1$ . Ta có thể tính giá trị của mảng `sum` thông qua giá trị của `partial`, do

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

Trường hợp cơ bản của hàm là

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

do trong trường hợp này, không có phần tử nào có thể bỏ khỏi  $S$ . Sau đó, trong trường hợp tổng quát, ta có công thức truy hồi sau:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Ở đây, ta chú ý vào giá trị  $k$ . Nếu  $k \in S$ , ta có hai lựa chọn: giữ  $k$  lại trong  $S$  hoặc bỏ  $k$  ra khỏi  $S$ .

Có một cách cài đặt rất hay để tính tổng các tập con. Đầu tiên, ta khai báo một mảng

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

để lưu tổng cần tính của từng tập con. Mảng được khởi tạo như sau:

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

Sau đó, ta có thể tính các phần tử trong 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 code phía trên được dùng để tính giá trị của  $\text{partial}(S, k)$  với  $k = 0 \dots n - 1$  vào mảng `sum`. Do  $\text{partial}(S, k)$  luôn được tính dựa vào  $\text{partial}(S, k - 1)$ , nên ta có thể tái sử dụng mảng `sum` và có một cách cài đặt rất tối ưu.

## **Phần II**

### **Đồ thị**



# Chương 11

## Cơ bản của Đồ thị

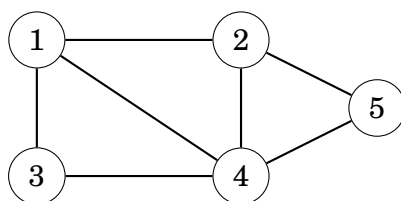
Rất nhiều bài toán lập trình có thể giải được bằng cách mô hình hóa thành đồ thị. Ví dụ điển hình của đồ thị là một mạng lưới gồm các đường đi và thành phố trong một quốc gia. Đôi lúc, mô hình đồ thị sẽ bị giấu đi và sẽ không dễ để nhận ra hướng giải này.

Phần sách này sẽ trình bày các thuật toán đồ thị và tập trung chủ yếu vào các chủ đề quan trọng trong lập trình thi đấu. Ở chương này, ta sẽ tìm hiểu các khái niệm liên quan tới đồ thị, và các cách khác nhau để biểu diễn đồ thị trong các thuật toán.

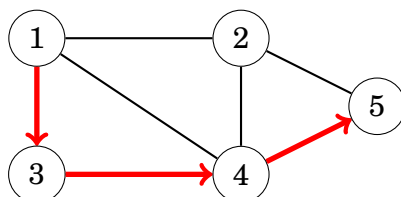
### 11.1 Thuật ngữ trong đồ thị

Một **đồ thị** bao gồm các **đỉnh** và các **cạnh**. Trong sách này, ta quy ước biến  $n$  là số lượng đỉnh trong một đồ thị, và biến  $m$  là số lượng cạnh. Các đỉnh sẽ được đánh số bằng các số nguyên  $1, 2, \dots, n$ .

Ví dụ, đồ thị sau gồm có 5 đỉnh và 7 cạnh:



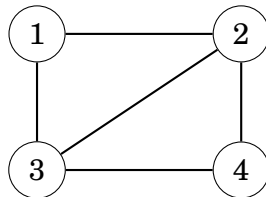
Một **đường đi** sẽ nối từ đỉnh  $a$  tới đỉnh  $b$  bằng các cạnh của đồ thị. **Độ dài** của một đường đi là số cạnh nằm trên đường đi đó. Ví dụ, đồ thị ở trên chứa một đường đi  $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  có độ dài 3 từ đỉnh 1 tới đỉnh 5:



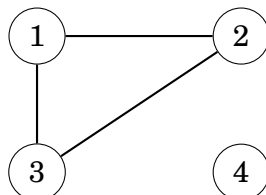
Một đường đi là **chu trình** nếu đỉnh đầu tiên trùng với đỉnh cuối cùng. Ví dụ, đồ thị trên chứa một chu trình  $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ . Một đường đi là **đường đi đơn** nếu mỗi đỉnh thuộc đường đi xuất hiện tối đa 1 lần.

## Tính liên thông

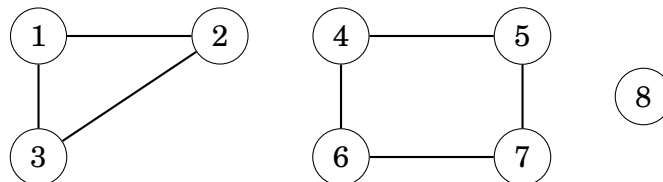
Một đồ thị là **liên thông** nếu giữa hai đỉnh bất kỳ luôn tồn tại ít nhất một đường đi. Ví dụ, đồ thị sau là liên thông:



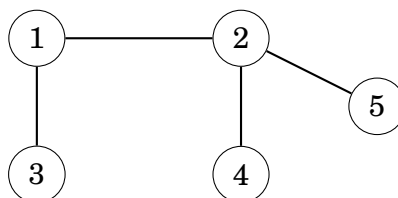
Đồ thị sau không liên thông vì ta không thể đi từ đỉnh 4 tới bất kỳ đỉnh nào khác.



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



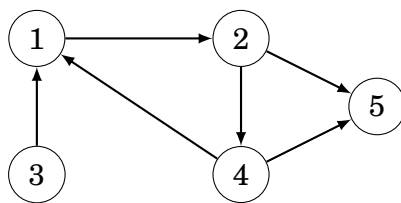
**Cây** là một đồ thị liên thông gồm  $n$  đỉnh và  $n - 1$  cạnh. Tồn tại duy nhất một đường đi giữa hai đỉnh bất kỳ trong cây. Ví dụ, đồ thị sau đây là một cây:



## Hướng của cạnh

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

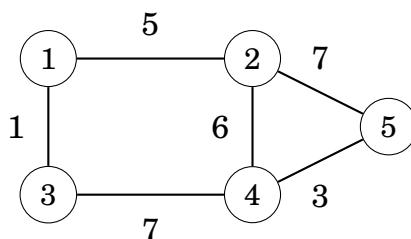




Đồ thị trên chứa một đường  $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$  từ đỉnh 3 tới đỉnh 5, nhưng không có đường đi ngược lại từ đỉnh 5 tới đỉnh 3.

## Trọng số cạnh

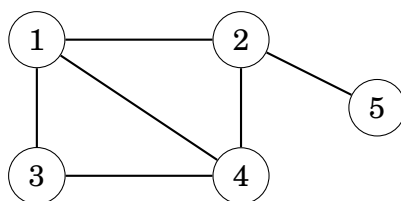
Trong một đồ thị **có trọng số**, mỗi cạnh sẽ được gán một **trọng số**. Các trọng số thường được coi như độ dài của cạnh. Ví dụ, đồ thị sau đây là đồ thị có trọng số:



Độ dài của một đường đi trong đồ thị có trọng số là tổng trọng số của các cạnh thuộc 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 thứ hai là đường đi **ngắn nhất** từ đỉnh 1 tới đỉnh 5.

## Đỉnh kề và bậc của đỉnh

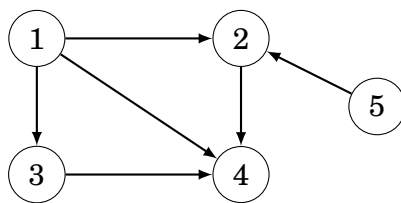
Hai đỉnh được gọi là **kề nhau** nếu tồn tại một cạnh giữa chúng. **Bậc** của một đỉnh là số lượng đỉnh kề nó. Ví dụ, trong đồ thị sau, các đỉnh kề của đỉnh 2 là 1, 4, và 5, nên bậc của nó là 3.



Tổng bậc của các đỉnh trong một đồ thị luôn luôn là  $2m$ , trong đó  $m$  là số lượng cạnh, vì mỗi cạnh sẽ tăng bậc của đúng hai đỉnh thêm một. Do đó, tổng bậc của các đỉnh sẽ luôn là số chẵn.

Một đồ thị **chính quy** là đồ thị mà bậc của các đỉnh giống nhau và bằng một hằng số  $d$ . Một đồ thị **đầy đủ** là đồ thị mà bậc của các đỉnh là  $n - 1$ , hay nói cách khác, đồ thị chứa tất cả các cạnh có thể nối được giữa các đỉnh.

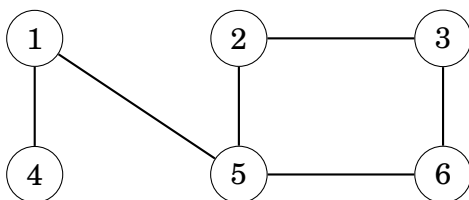
Trong một đồ thị có hướng, **bậc vào** của một đỉnh là số lượng cạnh đi vào đỉnh đó, và **bậc ra** của một đỉnh là số lượng cạnh đi ra khỏi đỉnh đó. Ví dụ, trong đồ thị sau, bậc vào của đỉnh 2 là 2, còn bậc ra của đỉnh 2 là 1.



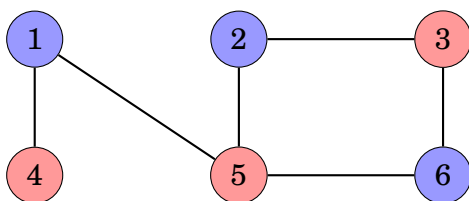
## Tô màu

Một cách **tô màu** đồ thị là một cách gán cho mỗi đỉnh một màu sao cho không có hai đỉnh kề nào có màu giống nhau.

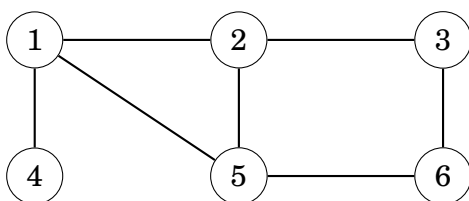
Một đồ thị là **hai phía** nếu tồn tại cách tô màu đồ thị đó chỉ với hai màu. Người ta chứng minh được rằng, một đồ thị là đồ thị hai phía khi và chỉ khi nó không chứa bất kỳ chu trình nào có độ dài lẻ. Ví dụ, đồ thị



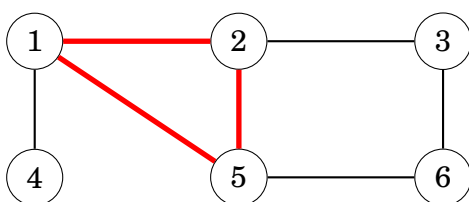
là đồ thị hai phía, và nó có thể được tô màu như sau:



Tuy nhiên, đồ thị

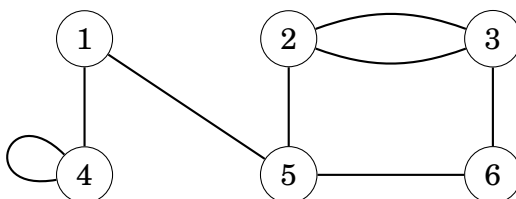


không phải đồ thị hai phía, vì ta không thể tô màu chu trình gồm 3 đỉnh sau chỉ bằng hai màu:



## Đơn đồ thị

Một **đơn đồ thị** là đồ thị mà không có cạnh nào có điểm đầu trùng với điểm cuối, và không có nhiều cạnh khác nhau cùng nối giữa hai đỉnh bất kỳ. Thông thường ta sẽ xem như các đồ thị đều là đơn đồ thị. Ví dụ, đồ thị sau đây *không phải* đơn đồ thị:



## 11.2 Biểu diễn đồ thị

Có nhiều cách để biểu diễn đồ thị trong các thuật toán. Việc lựa chọn cấu trúc dữ liệu nào sẽ dựa vào kích cỡ của đồ thị và cách hoạt động của thuật toán. Tiếp theo chúng ta sẽ tìm hiểu 3 cách biểu diễn thông dụng.

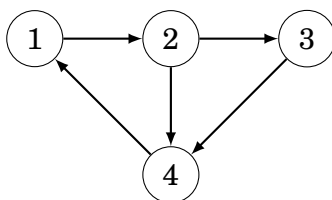
### Biểu diễn bằng danh sách kề

Với cách biểu diễn này, mỗi đỉnh  $x$  trong đồ thị sẽ có một **danh sách kề** chứa các đỉnh mà có cạnh nối từ  $x$  tới nó. Biểu diễn bằng danh sách kề là cách phổ biến nhất để biểu diễn đồ thị, và ta có thể áp dụng cách này để cài đặt hiệu quả phần lớn các thuật toán.

Ta có thể lưu trữ các danh sách kề một cách thuận tiện bằng cách khai báo một mảng chứa các vector như sau:

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

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



có thể được lưu trữ như sau:

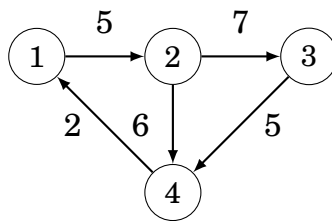
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Nếu đồ thị là vô hướng, ta có thể lưu trữ tương tự, nhưng mỗi cạnh cần phải được thêm theo cả hai hướng.

Với đồ thị có trọng số, cấu trúc dữ liệu của ta có thể được biến đổi như sau:

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

Ở đây, danh sách kề của đỉnh  $a$  sẽ chứa cặp  $(b, w)$  nếu có một cạnh nối từ đỉnh  $a$  tới đỉnh  $b$  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 đó chính là ta có thể tìm các đỉnh kề của đỉnh cho trước một cách hiệu quả. Ví dụ, vòng lặp sau đây có thể duyệt qua tất cả các đỉnh kề của đỉnh  $s$ .

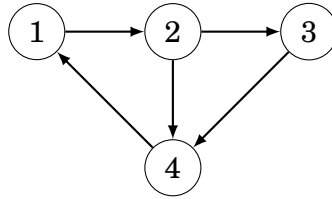
```
for (auto u : adj[s]) {
    // process node u
}
```

## Biểu diễn bằng ma trận kề

Một **ma trận kề** là một mảng hai chiều cho biết đồ thị chứa những cạnh nào. Ta có thể thông qua ma trận kề mà kiểm tra nhanh sự tồn tại của cạnh bất kỳ. Ma trận kề có thể được lưu trữ dưới dạng một mảng

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

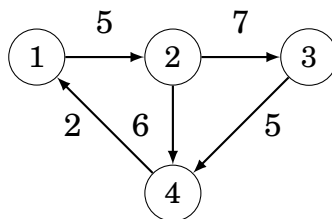
trong đó mỗi giá trị  $adj[a][b]$  cho biết đồ thị có chứa cạnh từ đỉnh  $a$  tới đỉnh  $b$  hay không. Nếu đồ thị có chứa cạnh này thì  $adj[a][b] = 1$ , ngược lại  $adj[a][b] = 0$ . Ví dụ, đồ thị



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, ma trận kề có thể biến đổi để có thể lưu trọng số của cạnh nếu cạnh đó tồn tại. Bằng cách biểu diễn này, đồ thị



tương đương 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

Điểm trừ của biểu diễn bằng ma trận kề là ma trận sẽ chứa  $n^2$  phần tử mà phần lớn trong số đó thường là 0. Do đó, cách biểu diễn trên không thể áp dụng cho các đồ thị lớn.

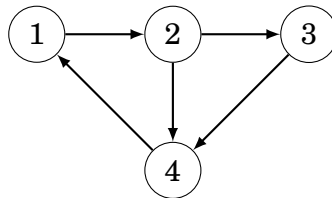
## Biểu diễn bằng danh sách cạnh

Một **danh sách cạnh** sẽ chứa tất cả các cạnh của đồ thị theo một thứ tự nào đó. Đây là một cách biểu diễn đồ thị tiện lợi cho các thuật toán cần duyệt qua tất cả các cạnh của đồ thị và không cần phải tìm các cạnh kề một đỉnh nào đó.

Danh sách cạnh có thể được lưu trữ trong một vector

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

trong đó mỗi cặp  $(a,b)$  cho biết tồn tại một cạnh từ đỉnh  $a$  tới đỉnh  $b$ . Do đó, đồ thị



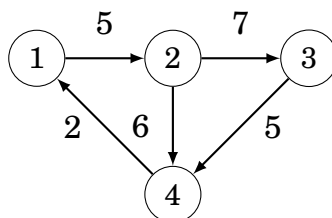
có thể được biểu diễn như sau:

```
edges.push_back({1,2});  
edges.push_back({2,3});  
edges.push_back({2,4});  
edges.push_back({3,4});  
edges.push_back({4,1});
```

Nếu đồ thị có trọng số, ta có thể biến đổi cấu trúc dữ liệu như sau:

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

Mỗi phần tử trong danh sách sẽ có dạng  $(a,b,w)$ , tương ứng với tồn tại một cạnh từ đỉnh  $a$  tới đỉnh  $b$  với trọng số  $w$ . Ví dụ, đồ thị



có thể biểu diễn như sau<sup>1</sup>:

```
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});
```

---

<sup>1</sup>Trong một số trình dịch cũ, ta phải sử dụng hàm `make_tuple` thay cho dấu ngoặc nhọn (ví dụ, `make_tuple(1,2,5)` thay cho `{1,2,5}`).

# Chương 12

## Duyệt đồ thị

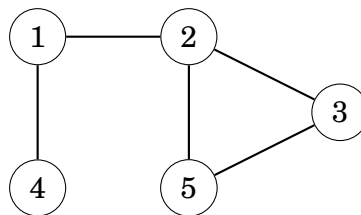
Trong chương này, chúng ta sẽ thảo luận về hai thuật toán nền tảng trong đồ thị: thuật toán duyệt theo chiều sâu (DFS) và thuật toán duyệt theo chiều rộng (BFS). Hai thuật toán này đều nhận đầu vào là một đỉnh xuất phát trên đồ thị và thăm tất cả những đỉnh có thể đến được từ đỉnh này. Sự khác nhau giữa hai thuật toán này là thứ tự thăm các đỉnh.

### 12.1 Duyệt theo chiều sâu

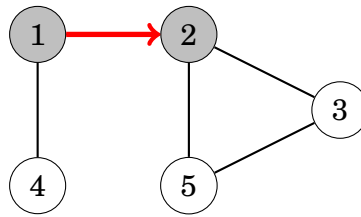
**Duyệt theo chiều sâu** là một kỹ thuật duyệt đồ thị đơn giản. Thuật toán bắt đầu từ một đỉnh trên đồ thị, và đi thăm qua tất cả các đỉnh khác mà có thể đến từ đỉnh bắt đầu bằng cách đi qua các cạnh của đồ thị. Duyệt chiều sâu luôn đi theo một đường đi đơn trong đồ thị chừng nào vẫn còn tìm thấy một đỉnh chưa được thăm. Sau đó, nó quay trở về những đỉnh liền trước và bắt đầu duyệt những phần khác của đồ thị. Thuật toán này đánh dấu lại các đỉnh đã đi qua, vì vậy chỉ thăm mỗi đỉnh đúng một lần.

#### Ví dụ

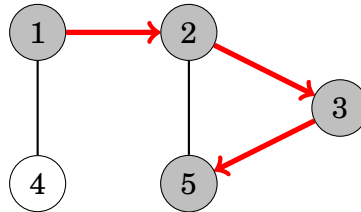
Hãy xem cách hoạt động của duyệt chiều sâu trong đồ thị sau:



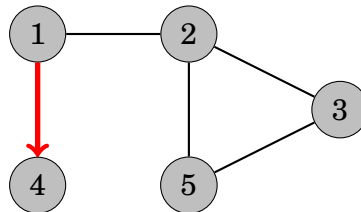
Chúng ta có thể bắt đầu từ một đỉnh bất kì ở trên đồ thị, ta sẽ bắt đầu từ đỉnh 1. Đầu tiên chúng ta tìm thấy đỉnh 2:



Sau đó, đỉnh 3 và 5 sẽ được thăm:



Đỉnh kề cạnh với đỉnh 5 là đỉnh 2 và 3, nhưng cả 2 đỉnh chúng ta đều đã thăm, vì vậy chúng ta sẽ quay lại đỉnh được thăm trước đó, 5 quay lại 3, đỉnh kề với 3 chúng ta cũng đã thăm vì vậy ta lại quay về 2, tương tự đỉnh kề 2 cũng đã thăm hết ta quay lại 1, đỉnh 4 kề với 1 chưa được thăm nên ta sẽ đến thăm đỉnh 4:



Sau đó, tìm kiếm kết thúc bởi vì tất cả đỉnh đã được thăm. Độ phức tạp của thuật toán duyệt theo chiều sâu là  $O(n + m)$ , ở đây  $n$  là số đỉnh và  $m$  là số cạnh của đồ thị, bởi vì mỗi đỉnh và mỗi cạnh của đồ thị chỉ duyệt qua một lần.

## Cài đặt

Duyệt theo chiều sâu có thể cài đặt đơn giản bằng đệ quy. Hàm `dfs` sau bắt đầu duyệt chiều sâu từ một đỉnh cho trước. Giả sử rằng đồ thị được lưu dưới dạng một mảng các danh sách kề.

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

và cần duy trì mảng

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

đánh dấu những đỉnh đã thăm. Ban đầu, toàn bộ giá trị đều là `false`, và khi ta đến thăm đỉnh  $s$ , giá trị của `visited[s]` trở thành `true`. Có thể cài đặt như sau:



```

void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}

```

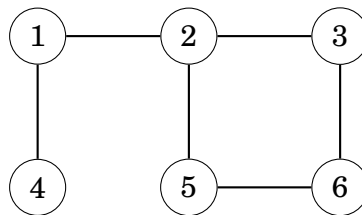
## 12.2 Duyệt theo chiều rộng

**Duyệt theo chiều rộng** sẽ thăm các đỉnh theo thứ tự tăng dần về khoảng cách so với đỉnh bắt đầu. Vì thế, ta có thể tính được khoảng cách từ đỉnh bắt đầu tới mọi đỉnh khác bằng phép duyệt theo chiều rộng. Tuy nhiên, duyệt theo chiều rộng khó cài đặt hơn so với duyệt theo chiều sâu.

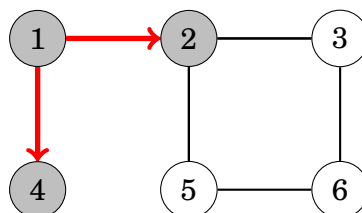
Duyệt chiều rộng duyệt qua các đỉnh theo từng mức. Đầu tiên, thuật toán sẽ duyệt qua những đỉnh có khoảng cách tới đỉnh bắt đầu là 1, sau đó là những đỉnh có khoảng cách là 2, cứ như vậy... Quá trình này lặp lại cho tới khi mọi đỉnh đều đã được thăm.

### Ví dụ

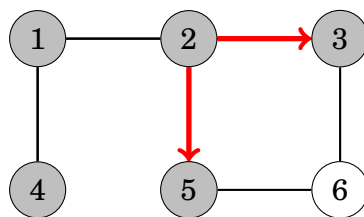
Hãy xem cách hoạt động của duyệt theo chiều rộng qua đồ thị sau:



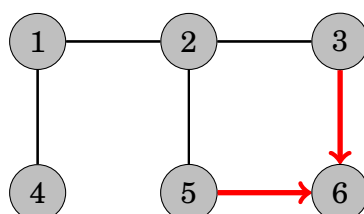
Giả sử đỉnh bắt đầu là 1. Đầu tiên, ta thăm tất cả đỉnh có thể tới từ đỉnh 1 chỉ qua 1 cạnh duy nhất (khoảng cách là 1):



Sau đó, ta thăm đỉnh 3 và 5:



Cuối cùng, ta thăm đỉnh 6:



Bây giờ chúng ta đã tính được khoảng cách từ đỉnh bắt đầu tới toàn bộ đỉnh trên đồ thị. Xem bảng sau:

Đỉnh	Khoảng cách
1	0
2	1
3	2
4	1
5	2
6	3

Giống như duyệt theo chiều sâu, độ phức tạp của thuật toán duyệt theo chiều rộng là  $O(n + m)$ , trong đó  $n$  là số đỉnh và  $m$  là số cạnh của đồ thị.

## Cài đặt

Duyệt theo chiều rộng khó cài đặt hơn so với duyệt theo chiều sâu, bởi vì thuật toán có thể thăm các đỉnh nằm ở nhiều phần khác nhau trên đồ thị. Một cách cài đặt điển hình là sử dụng hàng đợi để lưu lại các đỉnh. Ở mỗi bước, đỉnh tiếp theo trong hàng đợi sẽ được thăm.

Giả sử đồ thị được lưu dưới dạng danh sách kề, đoạn mã dưới đây sử dụng 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 khoảng cách tăng dần. Đỉnh mới luôn luôn được thêm vào cuối của hàng đợi, và đỉnh đầu tiên sẽ là đỉnh tiếp theo được thăm. Mảng  $visited$  cho biết các đỉnh đã được thăm hay chưa, và mảng  $distance$  sẽ lưu khoảng cách từ đỉnh bắt đầu tới toàn bộ đỉnh trên đồ thị.

Duyệt theo chiều rộng có thể cài đặt như sau (bắt đầu duyệt từ đỉnh  $x$ ):

```

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

```

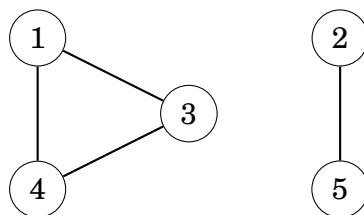
## 12.3 Ứng dụng

Sử dụng các thuật toán duyệt trên, chúng ta có thể kiểm tra nhiều tính chất của đồ thị. Thông thường chúng ta có thể sử dụng cả duyệt theo chiều sâu và duyệt theo chiều rộng. Nhưng trong thực tế, dùng duyệt theo chiều sâu sẽ tốt hơn, bởi vì nó dễ cài đặt hơn. Trong những ứng dụng sau, ta giả sử rằng đồ thị vô hướng.

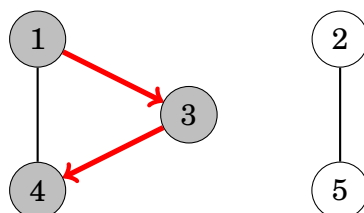
### Kiểm tra liên thông

Đồ thị được gọi là liên thông nếu luôn tồn tại đường đi giữa hai đỉnh bất kì trên đồ thị. Như vậy, chúng ta có thể kiểm tra xem đồ thị có liên thông không bằng cách bắt đầu tại một đỉnh bất kì và xem thử ta có thể đi tới được tất cả những đỉnh khác hay không.

Ví dụ, trong đồ thị



duyet theo chiều sâu bắt đầu từ đỉnh 1 và đi thăm các đỉnh như sau:

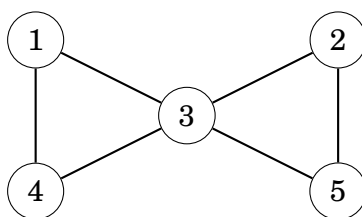


Vì không thăm được tất cả các đỉnh, chúng ta có thể kết luận rằng đồ thị không phải là đồ thị liên thông. Tương tự, chúng ta có thể tìm được tất cả thành phần liên thông của đồ thị bằng cách lặp qua mọi đỉnh, nếu đỉnh đó chưa thuộc thành phần liên thông nào thì bắt đầu tìm kiếm chiều sâu từ đó.

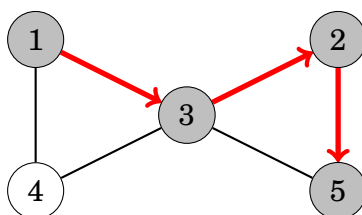
## Tìm chu trình

Một đồ thị chứa chu trình nếu trong quá trình duyệt qua các đỉnh chúng ta tìm thấy một đỉnh có đỉnh kề (khác với đỉnh liền trước trên đường đi) đã được thăm từ trước.

Ví dụ, đồ thị



chứa 2 chu trình và chúng ta có thể tìm ra một trong số chúng như sau:



Sau khi đi từ đỉnh 2 đến đỉnh 5 chúng ta nhận thấy rằng đỉnh 3 kề với 5 đã được thăm trước đó. Vì vậy, đồ thị chứa một chu trình đi qua đỉnh 3, đó là,  $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$ .

Một cách khác để kiểm tra xem đồ thị có chứa chu trình hay không chỉ đơn giản là tính số lượng đỉnh và cạnh trong mỗi thành phần liên thông. Nếu một thành phần liên thông chứa  $c$  đỉnh và không có chu trình, nó phải chứa chính xác  $c - 1$  cạnh (tức nó phải là một cây). Nếu có  $c$  cạnh hoặc hơn, thành phần liên thông đó chắc chắn chứa chu trình.

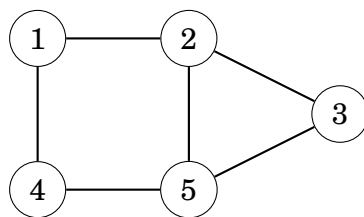
## Kiểm tra tính hai phía

Một đồ thị là hai phía nếu nó có thể được tô bằng hai màu khác nhau sao cho không có hai đỉnh kề nhau có cùng màu.

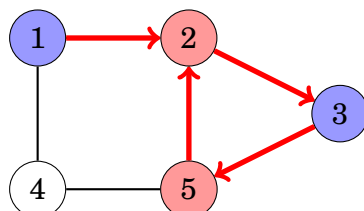
Thật ra việc kiểm tra xem một đồ thị có phải là đồ thị hai phía hay không rất dễ, nếu sử dụng các thuật toán duyệt đồ thị.

Ý tưởng là tô đỉnh bắt đầu bằng màu xanh, và tất cả các đỉnh kề nó bằng màu đỏ, lại tô màu xanh cho những đỉnh kề cận tiếp theo (kề với đỉnh có màu đỏ),... Nếu trong quá trình duyệt ta bắt gặp hai đỉnh kề nhau có cùng màu thì đây không phải là đồ thị hai phía. Ngược lại đây là đồ thị hai phía và ta đã tìm được một cách tô màu.

Ví dụ, đồ thị



không phải là hai phía, vì phép duyệt bắt đầu từ đỉnh 1 sẽ diễn ra như sau:



Chúng ta nhận ra rằng màu của hai đỉnh 2 và 5 là đỏ, trong khi chúng lại kề nhau trong đồ thị. Vì vậy, đây không phải là đồ thị hai phía.

Thuật toán này luôn đúng, bởi khi chỉ có 2 màu có thể tô được thì màu của đỉnh đầu tiên trong một thành phần liên thông sẽ quyết định màu của mọi đỉnh còn lại trong thành phần đó. Đỉnh bắt đầu có màu xanh hoặc đỏ đều không ảnh hưởng đến thuật toán.

Lưu ý rằng trong trường hợp tổng quát, rất khó để kiểm tra xem các đỉnh của đồ thị có thể được tô bằng  $k$  màu hay không sao cho không có hai đỉnh kề nào có cùng màu. Kể cả khi  $k = 3$ , chưa ai biết tới thuật toán nào hiệu quả hơn, bài toán đó là NP-hard.



# Chương 13

## Đường đi ngắn nhất

Tìm đường đi ngắn nhất giữa hai đỉnh của một đồ thị là một bài toán quan trọng có nhiều ứng dụng thực tiễn. Ví dụ, trong một mạng lưới giao thông, có một bài toán tự nhiên là tính độ dài đường đi ngắn nhất giữa hai thành phố, khi đã biết trước độ dài mỗi con đường.

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

### 13.1 Thuật toán Bellman-Ford

**Thuật toán Bellman-Ford**<sup>1</sup> tìm đường đi ngắn nhất từ một đỉnh sang tất cả đỉnh khác trong đồ thị. Thuật toán có thể xử lý mọi loại đồ thị nếu như trong đồ thị không chứa chu trình có tổng trọng số âm. Nếu đồ thị có chứa chu trình âm, thuật toán này có thể phát hiện điều đó.

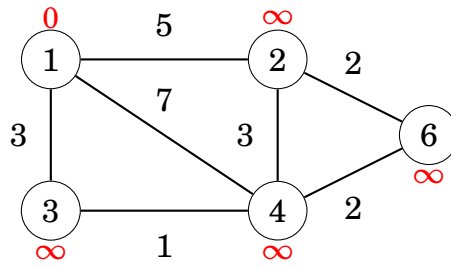
Thuật toán lưu lại khoảng cách từ đỉnh bắt đầu đến tất cả các đỉnh. Khi bắt đầu, khoảng cách đến đỉnh bắt đầu là 0 và khoảng cách đến tất cả các đỉnh còn lại là vô cùng. Thuật toán sẽ giảm các khoảng cách bằng cách tìm các cạnh có thể rút ngắn đường đi cho đến khi không thể giảm nữa.

#### Ví dụ

Ta hãy xem cách hoạt động của thuật toán Bellman-Ford trong đồ thị sau:

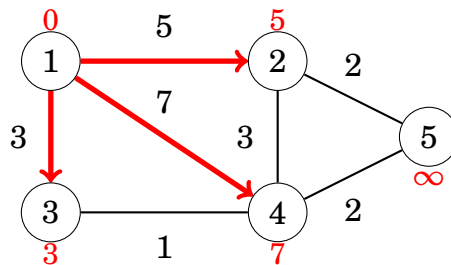
---

<sup>1</sup>Thuật toán được đặt tên dựa trên R. E. Bellman and L. R. Ford, hai người đã xuất bản bài báo độc lập với nhau vào năm 1958 và 1956, [5, 24].

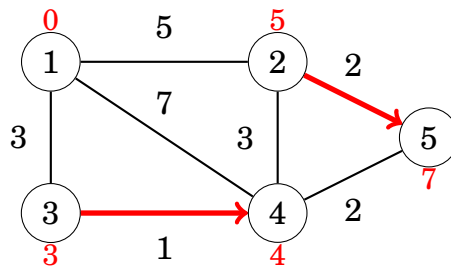


Mỗi đỉnh trong đồ thị được gán một khoảng cách. Ban đầu, khoảng cách đến đỉnh bắt đầu là 0, và khoảng cách đến tất cả các đỉnh còn lại là vô cùng.

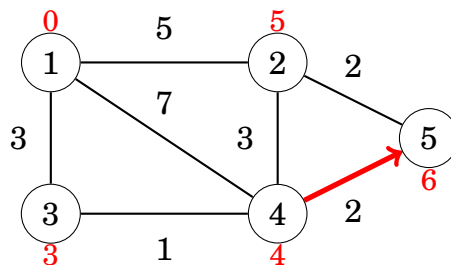
Thuật toán sẽ tìm các cạnh có thể giảm khoảng cách. Đầu tiên, các cạnh từ nút 1 có thể giảm khoảng cách



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



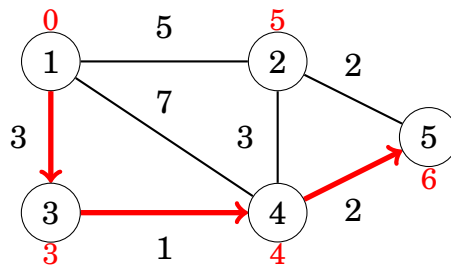
Cuối cùng, còn một sự thay đổi nữa:



Sau đó, không còn cạnh nào có thể giảm khoảng cách nữa. Có nghĩa là các khoảng cách đã được xác định, và chúng ta hoàn thành việc tính toán đường đi ngắn nhất từ đỉnh xuất phát đến các đỉnh khác trong đồ thị.

Ví dụ, một đường đi ngắn nhất từ đỉnh 1 đến đỉnh 5, gồm 3 cạnh:





## Cài đặt

Phần cài đặt của thuật toán Bellman-Ford xác định các đường đi ngắn nhất từ đỉnh  $x$  đến tất cả các đỉnh khác trong đồ thị. Đoạn mã sau giả định rằng đồ thị được lưu dưới dạng một danh sách cạnh edge chứa các bộ số có dạng  $(a, b, w)$ , nghĩa là có một cạnh nối từ đỉnh  $a$  đến đỉnh  $b$  với trọng số  $w$

Thuật toán bao gồm  $n - 1$  lần lặp, và trong mỗi lần lặp thuật toán duyệt qua tất cả các cạnh trên đồ thị và thử rút ngắn khoảng cách. Thuật toán sẽ xây dựng mảng `distance` chứa các khoảng cách từ  $x$  đến tất cả các đỉnh trên đồ thị. Hằng số `INF` tượng trưng cho khoảng cách vô cùng.

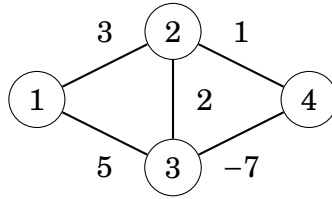
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Độ phức tạp của thuật toán là  $O(nm)$ , bởi vì thuật toán bao gồm  $n - 1$  lần lặp và mỗi lần duyệt qua toàn bộ  $m$  cạnh trong một lần lặp. Nếu như không có chu trình âm trong đồ thị, mọi khoảng cách sẽ chính xác sau  $n - 1$  lần lặp, bởi vì mọi đường đi ngắn nhất chỉ gồm tối đa  $n - 1$  cạnh.

Trên thực tế, các khoảng cách chính xác thường sẽ được tìm thấy nhanh hơn, trước khi đủ  $n - 1$  lần lặp. Vì vậy, một cách để khiến thuật toán hiệu quả hơn chính là dừng thuật toán nếu không còn khoảng cách nào có thể giảm được trong một lần lặp.

## Chu trình âm

Thuật toán Bellman-ford còn có thể được sử dụng để kiểm tra xem đồ thị có chứa chu trình âm hay không. Ví dụ, đồ thị



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

Nếu như đồ thị chứa một chu trình âm, chúng ta có thể giảm độ dài mọi đường đi chứa chu trình đó vô hạn lần bằng cách đi lặp lại theo chu trình liên tục. Vì vậy, khái niệm đường đi ngắn nhất không có ý nghĩa trong trường hợp này.

Một chu trình âm có thể được xác định sử dụng thuật toán Bellman-Ford bằng cách chạy thuật toán với  $n$  lần lặp. Nếu như trong vòng lặp cuối cùng, thuật toán làm giảm khoảng cách đến một đỉnh nào đó thì đồ thị chứa chu trình âm. Lưu ý rằng thuật toán này có thể tìm kiếm chu trình âm trên toàn bộ đồ thị bất kể đỉnh xuất phát là đỉnh nào.

## Thuật toán SPFA

**Thuật toán SPFA** ("Thuật toán tìm đường đi ngắn nhất nhanh hơn (Shortest Path Faster Algorithm)") [20] là một biến thể của thuật toán Bellman-Ford, nó thường tối ưu hơn thuật toán gốc. Thuật toán SPFA không duyệt qua tất cả các cạnh ở mỗi lần lặp, tuy nhiên thay vào đó, nó chọn các cạnh để xét một cách thông minh hơn.

Thuật toán lưu giữ một hàng đợi các đỉnh có thể được sử dụng để giảm khoảng cách. Đầu tiên, thuật toán thêm đỉnh xuất phát  $x$  vào hàng đợi. Sau đó, thuật toán luôn luôn xử lý đỉnh đầu tiên trong hàng đợi, và khi có một cạnh  $a \rightarrow b$  làm giảm khoảng cách, đỉnh  $b$  được thêm vào hàng đợi.

Tốc độ của thuật toán SPFA phụ thuộc vào cấu trúc của đồ thị: thuật toán thường sẽ tốt nhưng trong trường hợp tệ nhất thì độ phức tạp thuật toán vẫn là  $O(nm)$ . Có thể tạo ra những đồ thị khiến cho thuật toán chạy chậm ngang với thuật toán Bellman-Ford gốc.

## 13.2 Thuật toán Dijkstra

**Thuật toán Dijkstra**<sup>2</sup> tìm đường đi ngắn nhất từ đỉnh xuất phát đến toàn bộ các đỉnh khác trên đồ thị, giống như thuật toán Bellman-Ford. Lợi ích của thuật toán Dijkstra là nó chạy với tốc độ cao hơn và có thể được sử dụng để xử lý các đồ thị lớn. Tuy nhiên, thuật toán yêu cầu rằng không tồn tại cạnh với trọng số âm trong đồ thị.

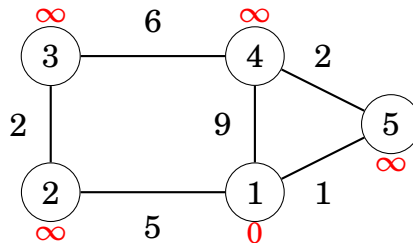
Giống như thuật toán Bellman-Ford, thuật toán Dijkstra lưu khoảng cách đến các đỉnh và giảm chúng trong lúc tìm kiếm. Thuật toán Dijkstra

<sup>2</sup>E. W. Dijkstra đã công bố thuật toán vào năm 1959 [14]; tuy nhiên, trong bài báo gốc của ông đã không nhắc đến cách để cài đặt thuật toán một cách hiệu quả.

chạy nhanh, là bởi vì nó chỉ xử lý mỗi cạnh trên đồ thị một lần duy nhất, tận dụng tính chất đồ thị không có cạnh trọng số âm.

## Ví dụ

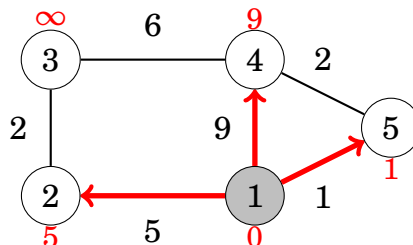
Hãy xem thuật toán Dijkstra hoạt động như thế nào trên đồ thị sau đây khi đỉnh xuất phát là đỉnh 1.



Giống như thuật toán Bellman-Ford, đầu tiên khoảng cách đến đỉnh bắt đầu là 0 và khoảng cách đến tất cả các đỉnh còn lại là vô cùng.

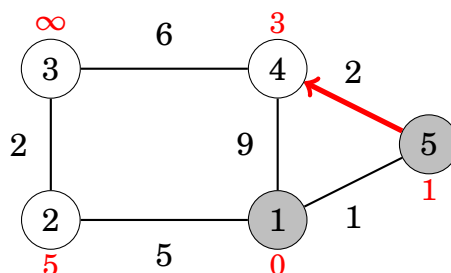
Ở mỗi bước, thuật toán Dijkstra chọn một đỉnh chưa được xử lý sao cho khoảng cách đến đỉnh đó nhỏ nhất có thể. Đỉnh đầu tiên như vậy sẽ là đỉnh 1 với khoảng cách 0.

Khi một đỉnh được chọn, thuật toán duyệt qua toàn bộ các cạnh bắt đầu từ đỉnh đó mà có thể làm giảm các khoảng cách của các đỉnh khác.

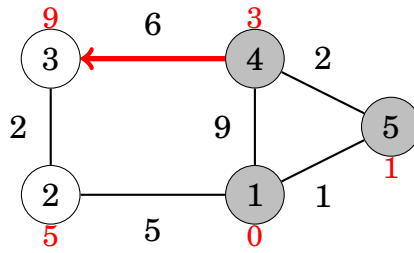


Trong trường hợp này, các cạnh từ đỉnh 1 đã làm giảm khoảng cách của các đỉnh 2, 4 và 5, khoảng cách của chúng bây giờ là 5, 9 và 1.

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

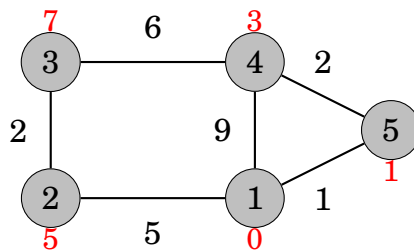


Sau đó, đỉnh tiếp theo là đỉnh 4, giảm khoảng cách đến đỉnh 3 xuống 9:



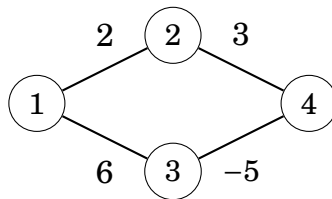
Một tính chất đáng chú ý trong thuật toán Dijkstra chính là khi một đỉnh được chọn, khoảng cách của đỉnh đó là chính xác. Ví dụ, xét đến lúc này thì các khoảng cách 0, 1 và 3 là khoảng cách cuối cùng của các đỉnh 1, 5 và 4.

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



## Cạnh âm

Thuật toán Dijkstra có tính hiệu quả cao dựa vào tính chất: đồ thị không chứa bất kỳ cạnh âm nào. Nếu như có một cạnh âm, thuật toán có thể đưa ra đáp án sai. Ví dụ, xét đồ thị sau:



Đường đi ngắn nhất từ đỉnh 1 đến đỉnh 4 là  $1 \rightarrow 3 \rightarrow 4$  có độ dài là 1. Tuy nhiên, thuật toán Dijkstra tìm đường  $1 \rightarrow 2 \rightarrow 4$  bằng cách theo cạnh với giá trị nhỏ nhất. Thuật toán không tính đến việc trên đường đi khác, trọng số  $-5$  đã bù trừ vào trọng số lớn 6 ở phía trước.

## Cài đặt

Phần cài đặt sau đây của thuật toán Dijkstra tính đường đi ngắn nhất từ đỉnh  $x$  đến các đỉnh khác trên đồ thị. Đồ thị được lưu dưới dạng danh sách kề sao cho  $\text{adj}[a]$  chứa một cặp  $(b, w)$  nếu tồn tại một cạnh nối từ đỉnh  $a$  đến đỉnh  $b$  với trọng số  $w$ .

Một cách cài đặt hiệu quả của thuật toán Dijkstra yêu cầu rằng nó phải có thể tìm nhanh đỉnh có khoảng cách nhỏ nhất mà chưa được xử lý. Một cấu trúc dữ liệu thích hợp cho điều này là một hàng đợi ưu tiên chứa các đỉnh xếp theo thứ tự là khoảng cách của chúng. Sử dụng một hàng đợi ưu tiên, đỉnh tiếp theo cần được xử lý có thể lấy ra được trong thời gian logarit.

Trong đoạn mã sau đây, hàng đợi ưu tiên  $q$  chứa các cặp với dạng  $(-d, x)$ , có nghĩa rằng khoảng cách hiện tại đến đỉnh  $x$  là  $d$ . Mảng `distance` chứa khoảng cách đến từng đỉnh, và mảng `processed` cho biết một đỉnh đã được xử lý hay chưa. Ban đầu khoảng cách là 0 cho  $x$  và  $\infty$  cho tất cả các đỉnh còn lại.

```
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 khoảng cách âm đến các đỉnh. Lý do cho việc này là bởi vì hàng đợi ưu tiên mặc định của C++ tìm phần tử lớn nhất, trong khi đó chúng ta cần tìm phần tử nhỏ nhất. Bằng cách sử dụng khoảng cách âm, chúng ta có thể trực tiếp sử dụng hàng đợi ưu tiên mặc định<sup>3</sup>. Cũng lưu ý rằng có thể sẽ có một đỉnh xuất hiện nhiều lần trong hàng đợi ưu tiên; tuy nhiên, chỉ duy nhất lần xuất hiện với khoảng cách nhỏ nhất sẽ được xử lý.

Độ phức tạp của phần cài đặt trên là  $O(n + m \log m)$ , bởi vì thuật toán duyệt qua toàn bộ các đỉnh của đồ thị và với mỗi cạnh, nó thêm vào tối đa một phần tử vào hàng đợi ưu tiên.

### 13.3 Thuật toán Floyd–Warshall

**Thuật toán Floyd–Warshall**<sup>4</sup> là một cách khác để tiếp cận bài toán tìm đường đi ngắn nhất. Không giống như các thuật toán khác trong chương

<sup>3</sup>Tất nhiên, chúng ta cũng có thể khai báo hàng đợi ưu tiên như Chương 4.5 và sử dụng khoảng cách dương, tuy nhiên phần cài đặt sẽ dài hơn một chút.

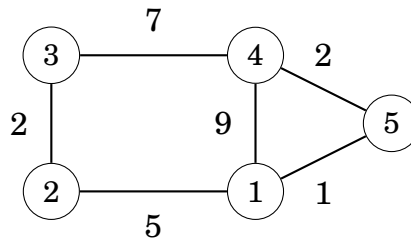
<sup>4</sup>Thuật toán được đặt theo tên R. W. Floyd và S. Warshall, họ đã đăng bài báo độc lập vào năm 1962 [23, 70].

này, nó tìm đường đi ngắn nhất giữa mọi cặp đỉnh trong một lần chạy.

Thuật toán duy trì một mảng hai chiều chứa khoảng cách giữa các đỉnh. Đầu tiên, khoảng cách được tính chỉ dựa vào các cạnh nối trực tiếp, và sau đó, thuật toán giảm các khoảng cách bằng cách sử dụng các đỉnh trung gian trên các đường đi.

## Ví dụ

Hãy xem cách hoạt động của thuật toán Floyd-Warshall trong đồ thị sau:



Đầu tiên, khoảng cách từ mỗi đỉnh đến chính nó là 0, và khoảng cách giữa đỉnh  $a$  và  $b$  là  $x$  nếu như có một cạnh nối đỉnh  $a$  và  $b$  với trọng số  $x$ . Tất cả các khoảng cách còn lại là vô cùng.

Trong đồ thị này, mảng ban đầu sẽ như sau:

	1	2	3	4	5
1	0	5	$\infty$	9	1
2	5	0	2	$\infty$	$\infty$
3	$\infty$	2	0	7	$\infty$
4	9	$\infty$	7	0	2
5	1	$\infty$	$\infty$	2	0

Thuật toán bao gồm nhiều lượt. Tại mỗi lượt, thuật toán chọn một đỉnh mới có thể được chọn là một đỉnh trung gian trên các đường đi, và các khoảng cách được giảm thông qua đỉnh này.

Trong lần lặp thứ nhất, đỉnh 1 là đỉnh trung gian mới. Có một đường đi mới giữa 2 và 4 với độ dài 14, bởi vì đỉnh 1 nối chúng lại. Còn một đường khác giữa đỉnh 2 và 5 với độ dài 6.

	1	2	3	4	5
1	0	5	$\infty$	9	1
2	5	0	2	<b>14</b>	<b>6</b>
3	$\infty$	2	0	7	$\infty$
4	9	<b>14</b>	7	0	2
5	1	<b>6</b>	$\infty$	2	0

Trong lần lặp thứ hai, đỉnh 2 là đỉnh trung gian mới. Nó tạo ra các đường đi mới giữa đỉnh 1 và 3 và giữa đỉnh 3 và 5:

	1	2	3	4	5
1	0	5	<b>7</b>	9	1
2	5	0	2	14	6
3	<b>7</b>	2	0	7	<b>8</b>
4	9	14	7	0	2
5	1	6	<b>8</b>	2	0

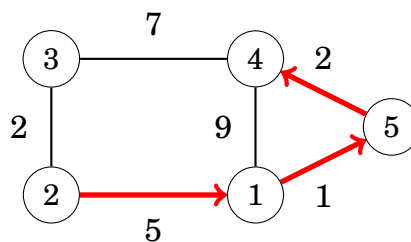
Trong lần lặp thứ ba, đỉnh 3 là đỉnh trung gian mới. Có một đường đi mới giữa đỉnh 2 và 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	<b>9</b>	6
3	7	2	0	7	8
4	9	<b>9</b>	7	0	2
5	1	6	8	2	0

Thuật toán tiếp tục như vậy, cho đến khi tất cả các đỉnh đều đã được chọn là đỉnh trung gian. Sau khi thuật toán kết thúc, mảng chứa khoảng cách nhỏ nhất giữa mỗi cặp hai đỉnh bất kỳ.

	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 trên cho ta biết rằng khoảng cách nhỏ nhất giữa đỉnh 2 và 4 là 8, tương ứng với đường đi sau:



## Cài đặt

Lợi thế của thuật toán Floyd-Warshall chính ở việc dễ cài đặt. Đoạn mã sau đây xây dựng một ma trận khoảng cách,  $\text{distance}[a][b]$  là khoảng cách nhỏ nhất giữa hai đỉnh  $a$  và  $b$ . Đầu tiên, thuật toán khởi tạo  $\text{distance}$  sử dụng ma trận kề  $\text{adj}$  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 nhỏ nhất sẽ được tính 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 của thuật toán này là  $O(n^3)$ , bởi vì nó gồm ba vòng lặp lồng nhau duyệt qua các đỉnh của đồ thị.

Bởi vì phần cài đặt của Floyd-Warshall đơn giản, thuật toán có thể là một lựa chọn tốt kể cả khi ta chỉ cần tìm một đường đi ngắn nhất trên đồ thị. Tuy nhiên, thuật toán chỉ có thể được sử dụng khi đồ thị đủ nhỏ để độ phức tạp lập phương đủ nhanh.

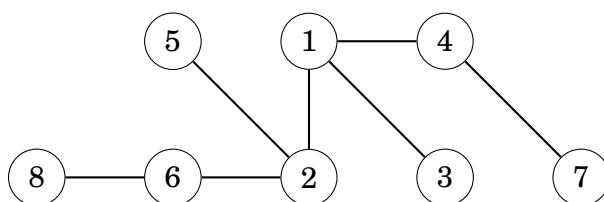


# Chương 14

## Các thuật toán trên cây

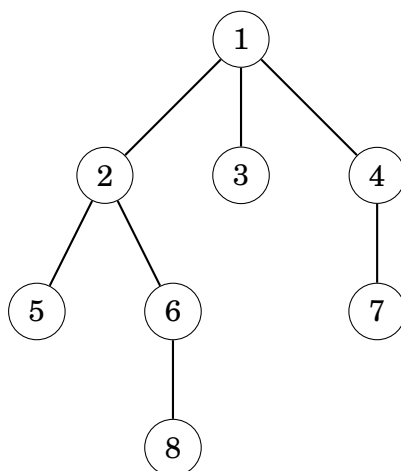
**Cây** là một đồ thị liên thông, không có chu trình gồm  $n$  đỉnh và  $n - 1$  cạnh. Xóa bỏ một cạnh bất kỳ trên cây sẽ chia cây thành hai phần, và thêm một cạnh bất kỳ vào cây sẽ tạo ra một chu trình. Hơn nữa, luôn luôn tồn tại một đường đi duy nhất giữa hai đỉnh bất kỳ trên cây.

Ví dụ, cây sau đây gồm 8 đỉnh và 7 cạnh:



**Lá** của một cây là các đỉnh có bậc bằng 1, tức là chỉ có một đỉnh kề. Ví dụ, lá của cây trên là các đỉnh 3, 5, 7 và 8.

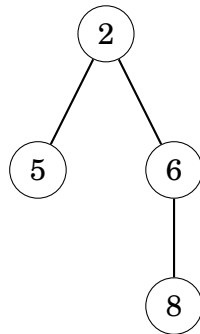
Trong một cây **có gốc**, một đỉnh được chỉ định là **gốc** của cây, và tất cả các đỉnh khác được đặt dưới gốc. Ví dụ, trong cây sau đây, đỉnh 1 là gốc của cây.



Trong một cây có gốc, **con** của một đỉnh là các đỉnh kề thấp hơn, và **cha** của một đỉnh chính là đỉnh kề cao hơn nó. Mỗi đỉnh có đúng một cha, ngoại

trừ nút gốc. Ví dụ, trong cây trên, con của đỉnh 2 là các đỉnh 5 và 6, và cha nó là đỉnh 1.

Cấu trúc của một cây có gốc có tính *đệ quy*: mỗi đỉnh của cây đóng vai trò là gốc của một **cây con** chứa bản thân đỉnh đó và tất cả các đỉnh thuộc vào cây con của các con của nó. Ví dụ, trong cây trên, cây con của đỉnh 2 bao gồm các đỉnh 2, 5, 6 và 8:



## 14.1 Duyệt cây

Các thuật toán duyệt đồ thị tổng quát có thể được sử dụng để duyệt qua các đỉnh của một cây. Tuy nhiên, duyệt cây dễ cài đặt hơn so với đồ thị tổng quát, bởi vì không có chu trình trong cây và không thể đi đến một đỉnh từ nhiều hướng khác nhau.

Cách thông thường để duyệt cây là bắt đầu tìm kiếm theo chiều sâu tại một đỉnh bất kỳ. Có thể sử dụng hàm đệ quy sau:

```
void dfs(int s, int e) {  
    // process node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Hàm này có hai tham số:  $s$  là đỉnh hiện tại và  $e$  là đỉnh trước đó. Mục đích của tham số  $e$  là để đảm bảo rằng thuật tìm kiếm chỉ di chuyển đến các đỉnh chưa được thăm.

Lời gọi hàm sau sẽ bắt đầu việc tìm kiếm từ đỉnh  $x$ :

```
dfs(x, 0);
```

Trong lần gọi đầu tiên,  $e = 0$  vì không có đỉnh nào trước đó, và ta được phép đi đến bất kỳ hướng nào trong cây.

### Quy hoạch động

Có thể sử dụng quy hoạch động để tính toán một số thông tin trong quá trình duyệt cây. Ví dụ, ta có thể tính toán các thông tin sau trong thời gian

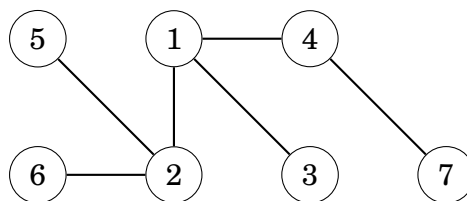
$O(n)$  cho toàn bộ đỉnh của một cây có gốc: số lượng đỉnh nằm trong mỗi cây con, hoặc độ dài lớn nhất của đường đi từ một đỉnh tới lá.

Để thí dụ, hãy tính cho mỗi đỉnh  $s$  một giá trị  $\text{count}[s]$ : số lượng đỉnh trong cây con của nó. Cây con này chứa bản thân đỉnh  $s$  và tất cả các đỉnh thuộc vào cây con của các con của nó, vì vậy ta có thể tính toán số lượng đỉnh một cách đệ quy bằng đoạn mã sau:

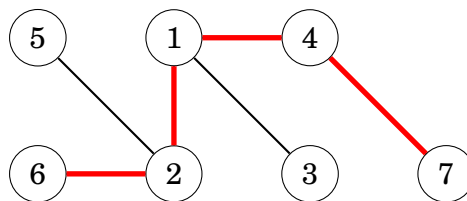
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

## 14.2 Đường kính

**Đường kính** của cây là độ dài lớn nhất của đường đi giữa hai đỉnh bất kỳ. Ví dụ, xét cây sau:



Đường kính của cây này là 4, tương ứng với đường đi sau:



Lưu ý rằng có thể có nhiều đường đi có độ dài lớn nhất. Trong đường đi trên, ta có thể thay thế đỉnh 6 bằng đỉnh 5 để thu được một đường đi khác có độ dài 4.

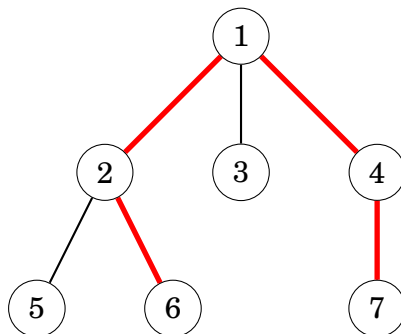
Tiếp theo, ta sẽ thảo luận hai thuật toán có độ phức tạp  $O(n)$  dùng để tính đường kính của cây. Thuật toán đầu tiên dựa trên quy hoạch động, và thuật toán thứ hai sử dụng hai lần duyệt theo chiều sâu.

### Thuật toán 1

Một hướng tổng quát để tiếp cận nhiều bài toán về cây là trước tiên, chọn một đỉnh bất kỳ làm gốc. Sau đó, ta có thể cố giải bài toán đó độc lập cho từng cây con. Thuật toán đầu tiên để tính đường kính dựa trên ý tưởng này.

Có một nhận xét quan trọng là mỗi đường đi trong cây có gốc có một *điểm cao nhất*: đỉnh cao nhất thuộc đường đi. Do đó, với mỗi đỉnh, ta có thể tính độ dài của đường đi dài nhất nhận đỉnh đấy làm điểm cao nhất. Một trong những đường đi đó chính là đường kính của cây.

Ví dụ, với cây sau, đỉnh 1 là điểm cao nhất trên đường đi, mà đường đó tương ứng với đường kính:



Với mỗi đỉnh  $x$ , ta tính hai giá trị:

- $\text{toLeaf}(x)$ : độ dài đường đi dài nhất từ  $x$  đến lá bất kỳ
- $\text{maxLength}(x)$ : độ dài đường đi dài nhất mà nhận  $x$  làm điểm cao nhất

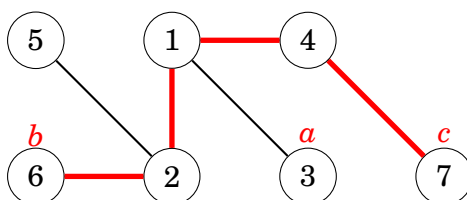
Ví dụ, với cây trên,  $\text{toLeaf}(1) = 2$ , ứng với đường đi  $1 \rightarrow 2 \rightarrow 6$ , và  $\text{maxLength}(1) = 4$ , ứng với  $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ . Trong trường hợp này,  $\text{maxLength}(1)$  chính là đường kính.

Quy hoạch động có thể được sử dụng để tính các giá trị trên cho tất cả các đỉnh trong  $O(n)$ . Đầu tiên, để tính  $\text{toLeaf}(x)$ , ta duyệt qua các con của  $x$ , chọn một con  $c$  có giá trị  $\text{toLeaf}(c)$  lớn nhất và cộng thêm một vào giá trị này. Sau đó, để tính  $\text{maxLength}(x)$ , ta chọn hai con  $a$  và  $b$  khác nhau sao cho tổng  $\text{toLeaf}(a) + \text{toLeaf}(b)$  lớn nhất và cộng thêm hai vào tổng này.

## Thuật toán 2

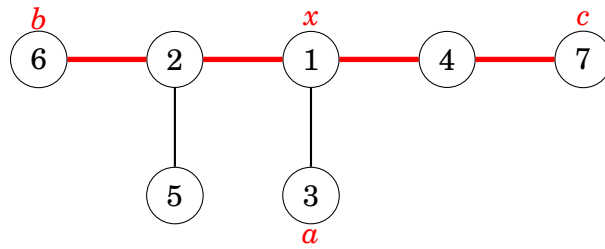
Một cách hiệu quả khác để tính đường kính của cây dựa trên hai lần duyệt theo chiều sâu. Đầu tiên, ta chọn một đỉnh  $a$  tùy ý trong cây và tìm đỉnh  $b$  xa  $a$  nhất. Sau đó, ta tìm đỉnh  $c$  xa  $b$  nhất. Đường kính của cây là khoảng cách giữa  $b$  và  $c$ .

Trong đồ thị sau,  $a$ ,  $b$  và  $c$  có thể là:



Đây là một cách làm tinh tế, nhưng vì sao nó đúng?

Sẽ dễ hình dung hơn nếu ta vẽ cây theo cách khác sao cho đường đi (tương ứng với đường kính) nằm ngang, và mọi đỉnh khác "treo" vào nó.

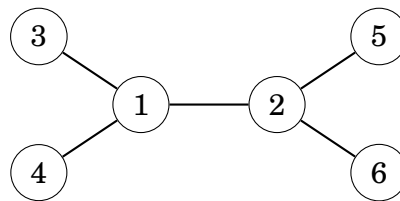


Đỉnh  $x$  báo hiệu nơi mà đường đi từ đỉnh  $a$  nhập vào đường đi tương ứng với đường kính. Đỉnh xa  $a$  nhất là  $b$ ,  $c$  hoặc một đỉnh khác chỉ ít cũng phải cách  $x$  một khoảng như vậy. Do đó, đỉnh  $b$  này luôn là một lựa chọn hợp lý để làm đầu cuối của một đường đi tương ứng với đường kính.

### 14.3 Mọi đường đi dài nhất

Vấn đề tiếp theo của chúng ta là tính độ dài lớn nhất của đường đi bắt đầu từ một đỉnh bất kỳ trong cây, cho mọi đỉnh. Đây có thể được xem như là phiên bản tổng quát của bài toán tìm đường kính của cây, vì độ dài lớn nhất trong số những độ dài đó bằng chính đường kính của cây. Bài toán này cũng có thể giải được trong  $O(n)$ .

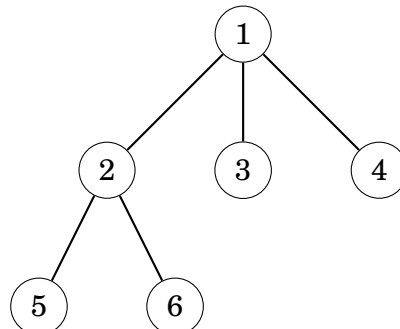
Ví dụ, xét cây sau:



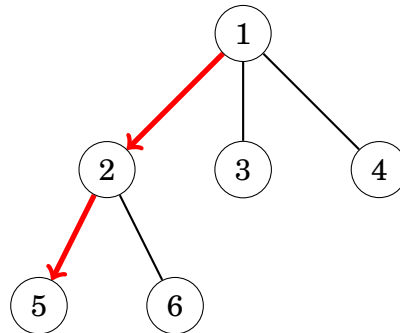
Đặt  $\text{maxLength}(x)$  là độ dài lớn nhất của một đường đi bắt đầu từ đỉnh  $x$ . Ví dụ, trong cây trên,  $\text{maxLength}(4) = 3$ , vì có một đường đi  $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ . Dưới đây là bảng chứa đầy đủ các giá trị:

đỉnh $x$	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Trong bài này, việc đầu tiên cần làm **cũng** là chọn một đỉnh bất kỳ làm gốc của cây.

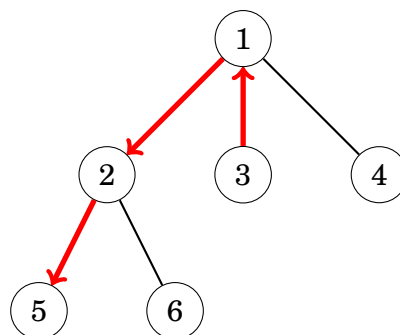


Phần đầu tiên của bài toán là với mỗi đỉnh  $x$ , tính độ dài lớn nhất của một đường đi xuống con của  $x$ . Ví dụ, độ dài lớn nhất của đường đi từ đỉnh 1 xuống con 2 của nó là:

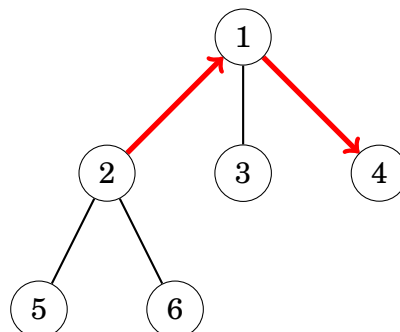


Phần này dễ giải được trong  $O(n)$ , vì ta có thể sử dụng quy hoạch động như đã làm trước đó.

Tiếp đến, phần thứ hai của bài toán là với mỗi đỉnh  $x$  tính độ dài lớn nhất của một đường đi qua cha  $p$  của nó. Ví dụ, đường đi dài nhất từ đỉnh 3 đi qua cha 1 của nó là:



Thoạt nhìn, trông có vẻ như ta nên chọn đường đi dài nhất từ  $p$ . Tuy nhiên, điều này *không phải* lúc nào cũng đúng, vì đường đi dài nhất từ  $p$  có thể đi qua  $x$ . Dưới đây là một ví dụ cho tình huống này:



Dù vậy, ta vẫn có thể giải phần thứ hai này trong  $O(n)$  bằng cách lưu trữ *hai* độ dài lớn nhất với mỗi đỉnh  $x$ :

- $\text{maxLength}_1(x)$ : độ dài lớn nhất của một đường đi từ  $x$

- $\text{maxLength}_2(x)$ : độ dài lớn nhất của một đường đi từ  $x$  khác hướng với đường đi đầu tiên

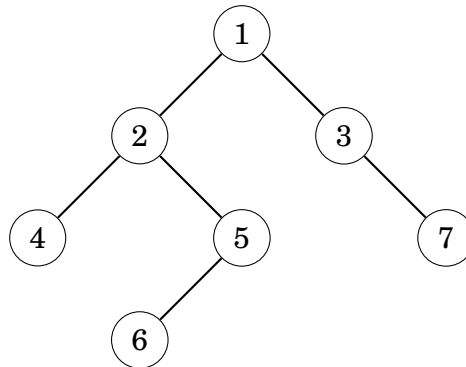
Ví dụ, trong đồ thị trên,  $\text{maxLength}_1(1) = 2$  với đường đi  $1 \rightarrow 2 \rightarrow 5$ , và  $\text{maxLength}_2(1) = 1$  ứng với đường đi  $1 \rightarrow 3$ .

Cuối cùng, nếu đường đi tương ứng với  $\text{maxLength}_1(p)$  đi qua  $x$ , ta kết luận rằng độ dài lớn nhất là  $\text{maxLength}_2(p) + 1$ , ngược lại độ dài lớn nhất là  $\text{maxLength}_1(p) + 1$ .

## 14.4 Cây nhị phân

Cây nhị phân là một cây có gốc mà trong đó mỗi nút có một cây con trái và một cây con phải. Cây con có thể rỗng. Do đó, mỗi nút trong cây nhị phân có thể có không, một hoặc hai con.

Ví dụ, cây sau đây là một cây nhị phân:



Các nút của cây nhị phân có ba thứ tự tự nhiên tương ứng với các cách khác nhau để duyệt cây bằng đệ quy:

- **pre-order**: xử lý nút gốc trước, sau đó duyệt cây con trái, sau đó duyệt cây con phải
- **in-order**: duyệt cây con trái trước, sau đó xử lý nút gốc, sau đó duyệt cây con phải
- **post-order**: duyệt cây con trái trước, sau đó duyệt cây con phải, sau đó xử lý nút gốc

Với cây trên, các nút theo thứ tự pre-order là  $[1, 2, 4, 5, 6, 3, 7]$ , theo thứ tự in-order là  $[4, 2, 6, 5, 1, 3, 7]$  và theo thứ tự post-order là  $[4, 6, 5, 2, 7, 3, 1]$ .

Nếu ta biết thứ tự pre-order và in-order của một cây, ta có thể dựng lại chính xác cấu trúc của cây đó. Ví dụ, cây trên là cây duy nhất có thể có thứ tự pre-order là  $[1, 2, 4, 5, 6, 3, 7]$  và thứ tự in-order là  $[4, 2, 6, 5, 1, 3, 7]$ . Tương tự, thứ tự post-order và in-order cũng xác định cấu trúc của cây.

Tuy nhiên, nếu ta chỉ biết thứ tự pre-order và post-order thì khác. Trong trường hợp này, có thể có nhiều hơn một cây có các thứ tự khớp nhau. Ví dụ, trong cả hai cây sau đây



thứ tự pre-order là  $[1, 2]$  và thứ tự post-order là  $[2, 1]$ , nhưng cấu trúc của các cây là khác nhau.

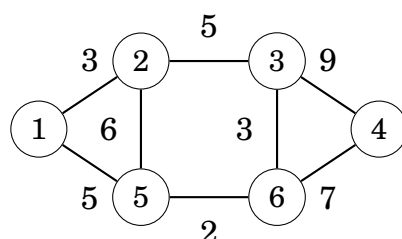


# Chương 15

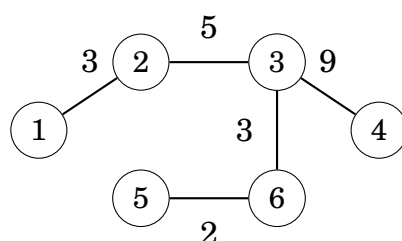
## Cây khung

**Cây khung** của một đồ thị bao gồm tất cả các đỉnh và một số cạnh của đồ thị đó sao cho có một đường đi giữa hai đỉnh bất kỳ. Giống như cây nói chung, cây khung có các đỉnh liên thông và không chứa chu trình. Thông thường có nhiều cách để xây dựng một cây khung.

Ví dụ, xét đồ thị sau:

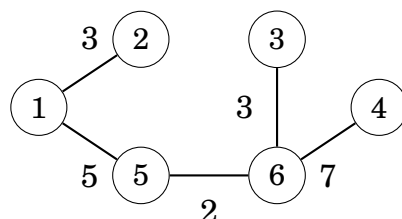


Một cây khung của đồ thị như sau:

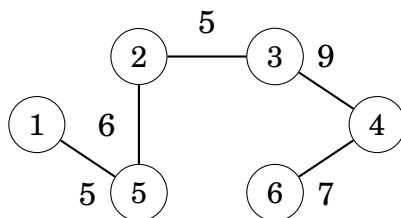


Trọng số của một cây khung là tổng trọng số các cạnh của nó. Ví dụ, trọng số của cây khung trên là  $3 + 5 + 9 + 3 + 2 = 22$ .

Một **cây khung nhỏ nhất** là một cây khung có trọng số nhỏ nhất có thể. Trọng số của cây khung nhỏ nhất cho đồ thị trong ví dụ là 20, và một cây như vậy có thể được xây dựng như sau:



Hiểu theo cách tương tự, một **cây khung lớn nhất** là một cây khung có trọng số lớn nhất có thể. Trọng số của cây khung lớn nhất cho đồ thị trên là 32:



Lưu ý rằng một đồ thị có thể có nhiều cây khung nhỏ nhất và lớn nhất, vì vậy các cây khung không phải là duy nhất.

Người ta tìm ra rằng có vài phương pháp tham lam có thể được sử dụng để xây dựng cây khung nhỏ nhất và lớn nhất. Trong chương này, ta sẽ thảo luận về hai thuật toán mà xử lý các cạnh của đồ thị theo thứ tự tăng dần về trọng số của chúng. Ta sẽ tập trung vào việc tìm cây khung nhỏ nhất. Các thuật toán tương tự cũng có thể tìm được cây khung lớn nhất bằng cách xử lý các cạnh theo thứ tự ngược lại.

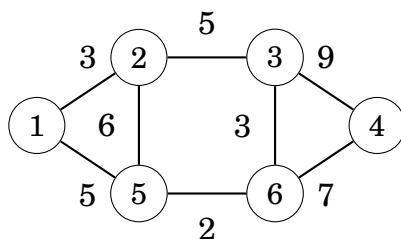
## 15.1 Thuật toán Kruskal

Trong **thuật toán Kruskal**<sup>1</sup>, cây khung ban đầu chỉ chứa các đỉnh của đồ thị và không chứa bất kỳ cạnh nào. Sau đó, thuật toán sẽ duyệt qua các cạnh theo thứ tự tăng dần về trọng số của chúng, và luôn luôn thêm một cạnh vào cây khung nếu nó không tạo thành chu trình.

Thuật toán duy trì các thành phần của cây. Ban đầu, mỗi đỉnh của đồ thị thuộc vào một thành phần riêng biệt. Khi một cạnh được thêm vào cây, hai thành phần sẽ được gộp lại. Cuối cùng, tất cả các đỉnh thuộc về cùng một thành phần và như vậy ta đã tìm thấy được một cây khung nhỏ nhất.

### Ví dụ

Ta cùng xem thuật toán Kruskal xử lý như thế nào với đồ thị sau:



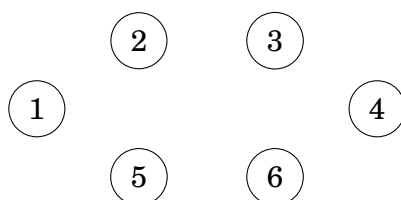
Bước đầu tiên của thuật toán là sắp xếp các cạnh theo thứ tự tăng dần về trọng số của chúng. Ta có được danh sách sau:

<sup>1</sup>Thuật toán được công bố vào năm 1956 bởi J. B. Kruskal [48].

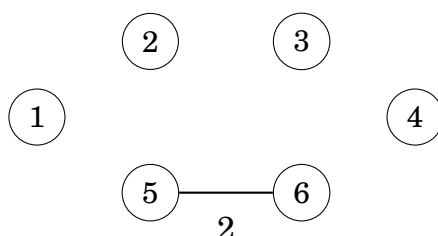
cạnh	trọng số
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Sau đó, thuật toán duyệt qua danh sách này và thêm mỗi cạnh vào cây nếu nó nối hai thành phần riêng biệt.

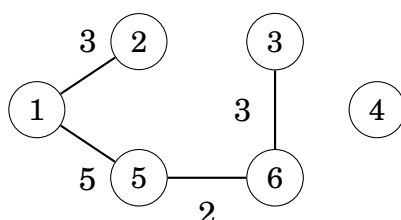
Ban đầu, mỗi đỉnh thuộc về một thành phần:



Cạnh đầu tiên được thêm vào cây là cạnh 5-6, tạo thành thành phần {5,6} bằng cách gộp hai thành phần {5} và {6}:



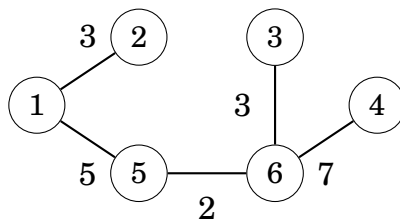
Sau đó, các cạnh 1-2, 3-6 và 1-5 được thêm vào tương tự như trên:



Sau các bước này, hầu hết các thành phần đã được gộp và còn 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 thêm vào cây, vì đỉnh 2 và 3 đã ở trong cùng một thành phần. Vì lý do tương tự, cạnh 2-5 cũng sẽ không được thêm vào cây.

Cuối cùng, cạnh 4–6 sẽ được thêm vào cây:

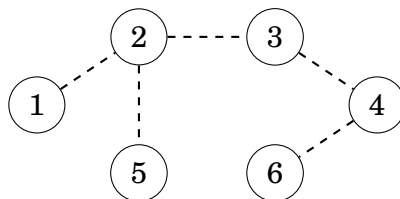


Sau đó, thuật toán sẽ không thêm bất kỳ cạnh mới nào nữa vì đồ thị đã liên thông và có một đường đi giữa hai đỉnh bất kỳ. Đồ thị thu được là một cây khung nhỏ nhất với trọng số  $2 + 3 + 3 + 5 + 7 = 20$ .

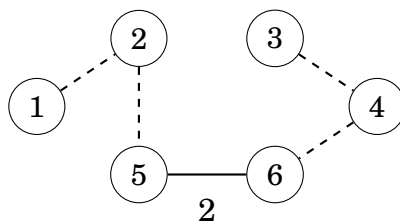
### Tại sao thuật toán này đúng?

Đó là một câu hỏi hay. Tại sao thuật toán Kruskal hoạt động? Tại sao phương pháp tham lam đảm bảo rằng ta sẽ tìm thấy một cây khung nhỏ nhất?

Hãy xem điều gì sẽ xảy ra nếu cạnh có trọng số nhỏ nhất của đồ thị *không* được bao gồm trong cây khung. Ví dụ, giả sử rằng có một cây khung trong đồ thị đã cho mà không chứa cạnh có trọng số nhỏ nhất 5–6. Ta không biết được cấu trúc chính xác của một cây khung như vậy, nhưng dù gì nó cũng phải chứa một số cạnh nào đó. Giả sử rằng cây có cấu trúc như sau:



Tuy nhiên, cây này không thể là một cây khung nhỏ nhất của đồ thị. Lý do cho điều này là ta có thể loại bỏ một cạnh trong cây và thay thế nó bằng cạnh có trọng số nhỏ nhất 5–6. Điều này sẽ tạo ra một cây khung có trọng số *nhỏ hơn*:



Vì lý do này, việc bao gồm cạnh có trọng số nhỏ nhất vào để tạo ra cây khung nhỏ nhất luôn là phương án tối ưu. Sử dụng lập luận tương tự, ta có thể chứng minh rằng việc thêm cạnh tiếp theo theo thứ tự tăng dần về trọng số vào cây cũng tối ưu, và cứ tiếp tục như vậy. Do đó, thuật toán Kruskal chính xác và luôn tạo ra một cây khung nhỏ nhất.

## Cài đặt

Khi cài đặt thuật toán Kruskal, ta sử dụng danh sách cạnh để biểu diễn đồ thị nhằm thuận tiện hơn. Giai đoạn đầu tiên của thuật toán là sắp xếp các cạnh trong danh sách trong  $O(m \log m)$ . Sau đó, giai đoạn thứ hai của thuật toán là 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 xử lý từng cạnh  $a-b$  (với  $a$  và  $b$  là hai đỉnh). Cần hai hàm: hàm `same` kiểm tra xem  $a$  và  $b$  có nằm trong cùng một thành phần không, và hàm `unite` gộp thành phần chứa  $a$  và thành phần chứa  $b$ .

Vấn đề là làm thế nào để cài đặt hiệu quả hai hàm `same` và `unite`. Một cách là cài đặt hàm `same` bằng cách duyệt đồ thị và kiểm tra xem ta có thể đi từ đỉnh  $a$  đến đỉnh  $b$ . Tuy nhiên, độ phức tạp của hàm này sẽ là  $O(n+m)$  và thuật toán sẽ chạy chậm vì hàm `same` sẽ được gọi với mỗi cạnh trong đồ thị.

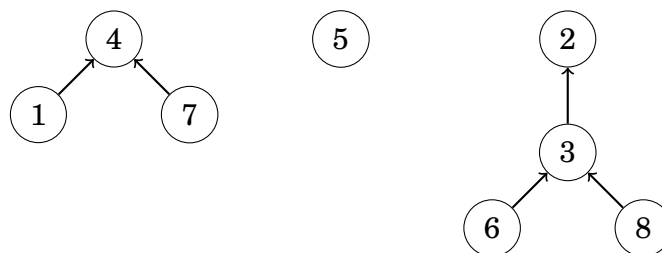
Ta sẽ giải bài toán bằng cách sử dụng một cấu trúc dữ liệu các tập không giao nhau mà thực hiện cả hai hàm trong  $O(\log n)$ . Do đó, độ phức tạp của thuật toán Kruskal sẽ là  $O(m \log n)$  sau khi sắp xếp danh sách cạnh.

## 15.2 Cấu trúc dữ liệu các tập không giao nhau

Một **cấu trúc dữ liệu các tập không giao nhau** duy trì một danh sách các tập hợp. Các tập này không giao nhau, do đó không có phần tử nào thuộc nhiều hơn một tập. Hai thao tác có độ phức tạp  $O(\log n)$  được hỗ trợ là: thao tác `unite` gộp hai tập, và thao tác `find` tìm đại diện của tập chứa một phần tử cho trước<sup>2</sup>.

### Cấu trúc

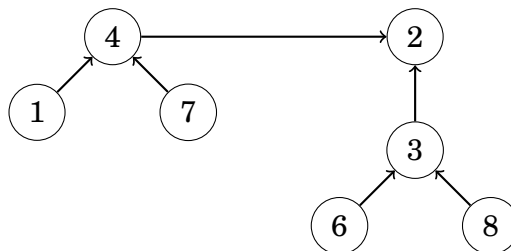
Trong một cấu trúc các tập không giao nhau, mỗi tập có một phần tử được chọn làm đại diện và luôn có một đường đi từ mọi phần tử khác của tập đến phần tử đại diện đó. Ví dụ, giả sử các tập là  $\{1, 4, 7\}$ ,  $\{5\}$  và  $\{2, 3, 6, 8\}$ :



<sup>2</sup>Cấu 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 nghiên cứu một biến thể phức tạp hơn của cấu trúc [64] được đề cập trong nhiều giáo trình thuật toán ngày nay.

Trong trường hợp này, các phần tử đại diện của các tập là 4, 5 và 2. Ta có thể tìm đại diện của của bất kỳ phần tử nào bằng cách đi theo đường đi bắt đầu từ phần tử đó. Ví dụ, phần tử 2 là đại diện của phần tử 6 vì ta có thể đi  $6 \rightarrow 3 \rightarrow 2$ . Hai phần tử thuộc cùng một tập khi và chỉ khi chúng có chung đại diện.

Hai tập có thể được gộp với nhau bằng cách nối đại diện của tập này vào đại diện của tập kia. Ví dụ, hai tập  $\{1, 4, 7\}$  và  $\{2, 3, 6, 8\}$  có thể được gộp như sau:



Tập thu được chứa các phần tử  $\{1, 2, 3, 4, 6, 7, 8\}$ . Từ đây, phần tử 2 là phần tử đại diện của cả tập và phần tử đại diện cũ là 4 sẽ trở tới phần tử 2.

Độ hiệu quả của cấu trúc dữ liệu các tập không giao nhau phụ thuộc vào cách gộp các tập. Hoá ra ta có thể áp dụng một chiến lược đơn giản như sau: luôn nối phần tử đại diện của tập *nhỏ hơn* với phần tử đại diện của tập *lớn hơn* (hoặc nếu hai tập có cùng kích thước, ta có thể chọn một cách tùy ý). Sử dụng chiến lược này, độ dài của mọi đường đi cũng là  $O(\log n)$ , do đó ta có thể tìm đại diện của bất kỳ phần tử nào một cách hiệu quả bằng cách lần theo đường đi tương ứng.

## Cài đặt

Cấu trúc dữ liệu các tập không giao nhau có thể được cài đặt bằng mảng. Trong cách cài đặt dưới đây, mảng `link` chứa phần tử tiếp theo trong đường đi hoặc chính phần tử đó nếu nó là phần tử đại diện, và mảng `size` chứa kích thước của tập tương ứng với phần tử đại diện.

Ban đầu, mỗi phần tử thuộc một tập 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 phần tử đại diện của phần tử  $x$ . Phần tử đại diện có thể tìm được bằng cách đi theo đường đi bắt đầu từ  $x$ .

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

Hàm `same` kiểm tra xem hai phần tử  $a$  và  $b$  có thuộc cùng một tập hay không. Điều này có thể làm được dễ dàng bằng cách sử dụng hàm `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

Hàm unite gộp hai tập chứa hai phần tử  $a$  và  $b$  (hai phần tử phải thuộc hai tập khác nhau). Đầu tiên hàm này tìm phần tử đại diện của hai tập và sau đó nối tập nhỏ hơn vào tập lớn hơn.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

Độ phức tạp của hàm find là  $O(\log n)$  nếu độ dài của mỗi đường đi là  $O(\log n)$ . Trong trường hợp này, hàm same và unite cũng hoạt động trong  $O(\log n)$ . Hàm unite đảm bảo rằng độ dài của mỗi đường đi là  $O(\log n)$  bằng cách nối tập nhỏ hơn vào tập lớn hơn.

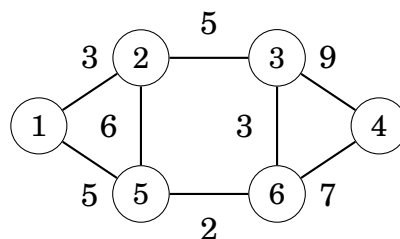
## 15.3 Thuật toán Prim

**Thuật toán Prim**<sup>3</sup> là một phương pháp khác để tìm cây khung nhỏ nhất. Đầu tiên thuật toán thêm một đỉnh tùy ý vào cây. Sau đó, thuật toán luôn chọn một cạnh có trọng số nhỏ nhất mà có thể thêm một nút mới vào cây. Cuối cùng, tất cả các nút đã được thêm vào cây và ta đã tìm thấy một cây khung nhỏ nhất.

Thuật toán Prim giống với thuật toán Dijkstra. Sự khác biệt duy nhất là thuật toán Dijkstra luôn chọn một cạnh có khoảng cách từ đỉnh bắt đầu là nhỏ nhất, nhưng thuật toán Prim đơn giản là chọn cạnh có trọng số nhỏ nhất mà có thể thêm một đỉnh mới vào cây.

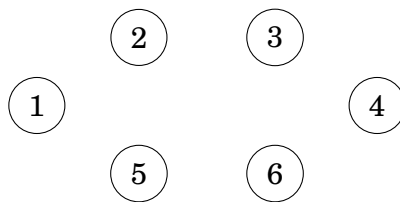
### Ví dụ

Hãy xem thuật toán Prim hoạt động như thế nào trong đồ thị sau:

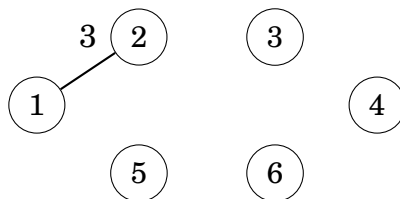


<sup>3</sup>Thuật toán được đặt theo tên R. C. Prim, người đã công bố thuật toán này vào năm 1957 [54]. Tuy nhiên, thuật toán này đã được tìm ra từ năm 1930 bởi V. Jarník.

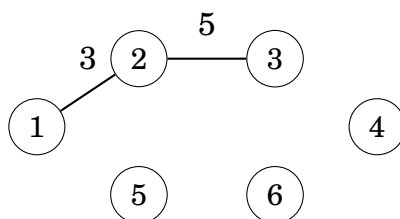
Ban đầu, không có cạnh nào giữa các đỉnh:



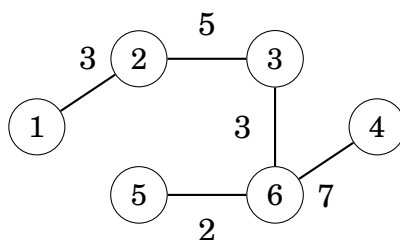
Một đỉnh tùy ý có thể được chọn là đỉnh bắt đầu, vì vậy hãy chọn đỉnh 1. Đầu tiên, chúng ta thêm đỉnh 2 được nối với đỉnh 1 bởi một cạnh có trọng số là 3:



Sau đó, có hai cạnh có trọng số là 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 được bao gồm trong cây:



## Cài đặt

Giống như thuật toán Dijkstra, thuật toán Prim có thể được cài đặt một cách hiệu quả bằng cách sử dụng hàng đợi ưu tiên. Hàng đợi ưu tiên chứa tất cả các đỉnh có thể được nối vào thành phần hiện tại thông qua một cạnh, xếp theo thứ tự tăng dần về trọng số của cạnh nối tương ứng.

Độ phức tạp của thuật toán Prim là  $O(n + m \log m)$  bằng với độ phức tạp của thuật toán Dijkstra. Trong thực tế, thuật toán Prim và Kruskal đều hiệu quả, và việc lựa chọn thuật toán nào là tùy vào sở thích. Tuy nhiên, hầu hết mọi người đều dùng thuật toán Kruskal.



# Chương 16

## Đồ thị có hướng

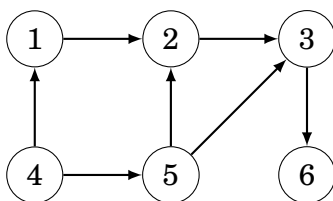
Chương này sẽ tập trung vào hai lớp đồ thị có hướng:

- **Đồ thị không có chu trình (Acyclic graph):** Không tồn tại chu trình trong đồ thị, vì vậy không tồn tại đường đi từ một đỉnh đến chính nó<sup>1</sup>.
- **Đồ thị mặt trời (Successor graph):** Bậc ra của mỗi đỉnh là 1, vì vậy mỗi đỉnh đều có đúng một đỉnh kế tiếp duy nhất.

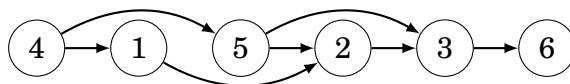
Trong cả hai loại đồ thị này, ta có thể thiết kế ra những thuật toán hiệu quả dựa trên các tính chất đặc biệt của chúng.

### 16.1 Sắp xếp Tô-pô

**Sắp xếp Tô-pô** (topological sort) là một cách sắp xếp các đỉnh của một đồ thị có hướng sao cho nếu tồn tại một đường đi từ đỉnh  $a$  đến đỉnh  $b$ , thì đỉnh  $a$  phải xuất hiện trước đỉnh  $b$  trong thứ tự sắp xếp. Ví dụ, cho đồ thị



một thứ tự sắp xếp Tô-pô là [4, 1, 5, 2, 3, 6]:



Một đồ thị có hướng phi chu trình luôn có ít nhất một thứ tự sắp xếp Tô-pô. Nhưng nếu đồ thị có chứa chu trình, không thể tạo được một thứ tự Tô-pô, vì không có đỉnh nào trong chu trình có thể xuất hiện trước những

<sup>1</sup>Đồ thị có hướng phi chu trình đôi khi được gọi là DAG

đỉnh khác cùng nằm trong chu trình đó trong thứ tự sắp xếp. Ta có thể dùng thuật toán duyệt theo chiều sâu để vừa kiểm tra một đồ thị có hướng có chứa chu trình hay không, vừa xây dựng một thứ tự sắp xếp Tô-pô nếu đồ thị đó không có chu trình.

## Thuật toán

Ý tưởng là xét qua từng đỉnh trong đồ thị và bắt đầu duyệt theo chiều sâu từ một đỉnh nào đó nếu nó chưa được duyệt đến. Trong quá trình duyệt đồ thị, các đỉnh có một trong ba trạng thái sau:

- Trạng thái 0: đỉnh chưa được duyệt đến (màu trắng)
- Trạng thái 1: đỉnh đang trong quá trình duyệt (màu xám nhạt)
- Trạng thái 2: đỉnh đã được duyệt (màu xám đậm)

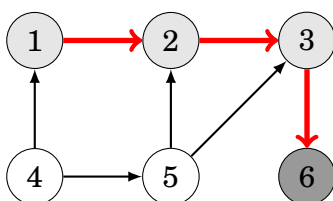
Ban đầu, trạng thái mỗi đỉnh là 0. Khi một đỉnh được duyệt đến lần đầu tiên, trạng thái của đỉnh đó trở thành 1. Cuối cùng, sau khi tất cả các đỉnh kế tiếp nó đã được duyệt xong, trạng thái của đỉnh đó trở thành 2.

Nếu như đồ thị có chứa chu trình, ta sẽ phát hiện được trong quá trình duyệt, bởi vì sẽ có một thời điểm một đỉnh có trạng thái 1 được xét đến. Trong trường hợp này không thể nào xây dựng được một thứ tự sắp xếp Tô-pô.

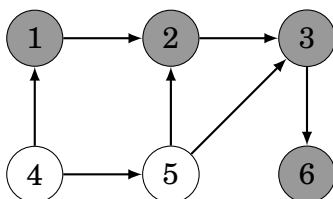
Nếu đồ thị không chứa chu trình, thứ tự sắp xếp Tô-pô có thể được xây dựng bằng cách thêm từng đỉnh vào danh sách ngay khi trạng thái đỉnh đó trở thành 2. Đảo ngược danh sách, ta được một thứ tự sắp xếp Tô-pô.

## Ví dụ 1

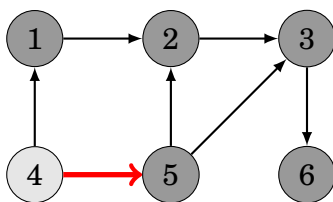
Trong đồ thị ví dụ, thuật toán duyệt bắt đầu từ đỉnh 1 đến đỉnh 6:



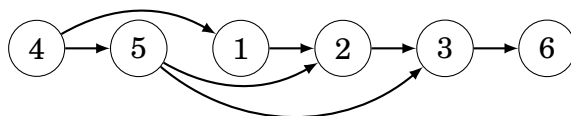
Bây giờ đỉnh 6 đã được duyệt, nên nó được thêm vào danh sách. Sau đó, đỉnh 3, 2 và 1 lần lượt được thêm vào danh sách:



Tại thời điểm này, danh sách là [6,3,2,1]. Lần duyệt tiếp theo bắt đầu tại đỉnh 4:



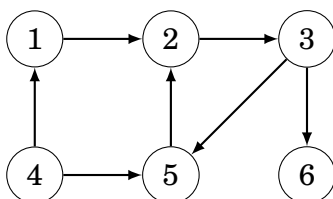
Như vậy, danh sách sau cùng là [6,3,2,1,5,4]. Tất cả các đỉnh đã được duyệt, nên ta đã dựng được một thứ tự sắp xếp Tô-pô. Thứ tự sắp xếp Tô-pô là danh sách sau khi đảo ngược [4,5,1,2,3,6]:



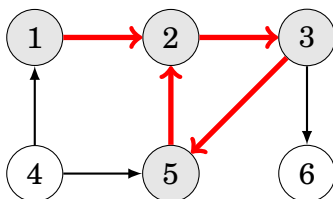
Lưu ý rằng thứ tự sắp xếp Tô-pô không phải là duy nhất, mà có thể có nhiều thứ tự sắp xếp Tô-pô khác nhau cho cùng một đồ thị.

## Ví dụ 2

Bây giờ, xét một đồ thị mà ta không thể xây dựng được một thứ tự Tô-pô nào cho nó, bởi vì đồ thị đó có chứa chu trình:



Quá trình duyệt diễn ra như sau:



Quá trình duyệt xét đến đỉnh 2 có trạng thái 1, điều này có nghĩa là đồ thị có chứa chu trình. Ở ví dụ này, có một chu trình là  $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ .

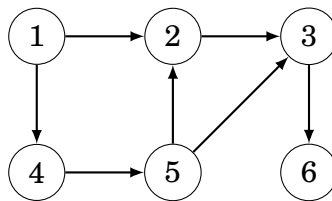
## 16.2 Quy hoạch động

Nếu một đồ thị có hướng không có chu trình, ta có thể áp dụng quy hoạch động cho đồ thị này. Ví dụ, ta có thể giải quyết một cách hiệu quả những bài toán xét các đường đi từ một đỉnh xuất phát đến một đỉnh kết thúc:

- có bao nhiêu đường đi khác nhau như thế?
- đường đi ngắn/dài nhất?
- số lượng cạnh ít/nhiều nhất trên một đường đi là bao nhiêu?
- những đỉnh nào chắc chắn xuất hiện trong mọi đường đi?

### Đếm số lượng đường đi

Ví dụ, đếm số lượng đường đi bắt đầu từ đỉnh 1 đến đỉnh 6 trong đồ thị sau đây:



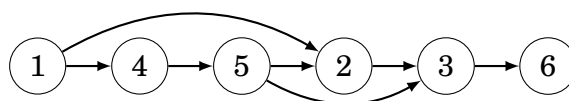
Có tổng cộng ba đường đi thoả mãn:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

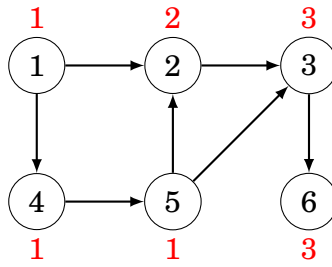
Gọi  $\text{paths}(x)$  là số lượng đường đi từ đỉnh 1 đến đỉnh  $x$ . Trường hợp gốc của quy hoạch động là  $\text{paths}(1) = 1$ . Sau đó, để tính các giá trị khác của  $\text{paths}(x)$ , ta có thể sử dụng hệ thức truy hồi

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

với  $a_1, a_2, \dots, a_k$  là các đỉnh có cạnh nối đến  $x$ . Bởi vì đồ thị không có chu trình, các giá trị của  $\text{paths}(x)$  có thể được tính theo một thứ tự Tô-pô. Có một thứ tự Tô-pô cho đồ thị trên, như sau:



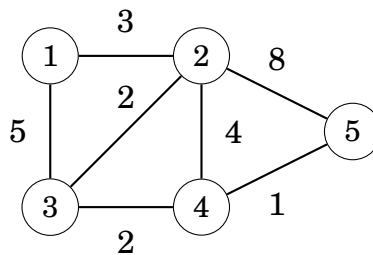
Vì vậy, số lượng đường đi tới từng đỉnh là:



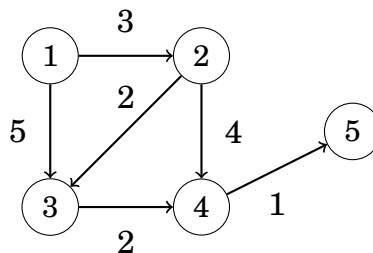
Ví dụ, để tính được giá trị của  $\text{paths}(3)$ , có thể sử dụng công thức  $\text{paths}(2) + \text{paths}(5)$ , bởi vì có các cạnh nối từ đỉnh 2 và 5 đến đỉnh 3. Vì  $\text{paths}(2) = 2$  và  $\text{paths}(5) = 1$ , ta kết luận  $\text{paths}(3) = 3$ .

## Thuật toán Dijkstra mở rộng

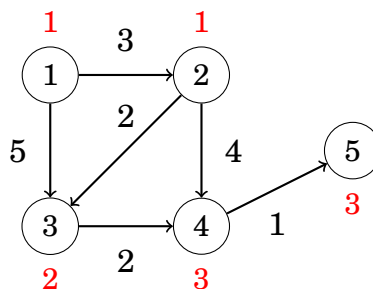
Một sản phẩm phụ của thuật toán Dijkstra là một đồ thị có hướng, phi chu trình có thể dùng để chỉ ra các đường đi ngắn nhất đến mỗi đỉnh của đồ thị ban đầu từ một đỉnh bắt đầu. Quy hoạch động có thể được áp dụng vào đồ thị này. Ví dụ, trong đồ thị



những đường đi ngắn nhất từ đỉnh 1 có thể đi qua các cạnh sau đây:



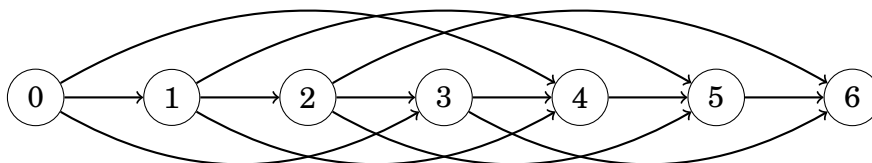
Ví dụ, bây giờ ta có thể tính số lượng đường đi ngắn nhất từ đỉnh 1 đến đỉnh 5 bằng cách sử dụng quy hoạch động:



## Biểu diễn bài toán thành đồ thị

Thực ra, bất kỳ bài toán quy hoạch động nào cũng có thể được biểu diễn thành một đồ thị có hướng, không chu trình. Trong đồ thị như vậy, mỗi đỉnh tương ứng với một trạng thái quy hoạch động và các cạnh cho biết các trạng thái phụ thuộc lẫn nhau như thế nào.

Ví dụ, xét bài toán tạo ra tổng tiền  $n$  từ các đồng xu  $\{c_1, c_2, \dots, c_k\}$ . Trong bài toán này, ta có thể xây dựng một đồ thị trong đó mỗi đỉnh tương ứng với một lượng tiền, và các cạnh cho biết cách chọn các đồng xu. Ví dụ, với các đồng xu  $\{1, 3, 4\}$  và  $n = 6$ , đồ thị trông như sau:



Sử dụng cách biểu diễn này, đường đi ngắn nhất từ đỉnh 0 đến đỉnh  $n$  tương ứng với một cách chọn với số lượng đồng xu nhỏ nhất, và tổng số đường đi từ đỉnh 0 đến đỉnh  $n$  tương ứng với tổng số cách tạo.

## 16.3 Đồ thị mặt trời

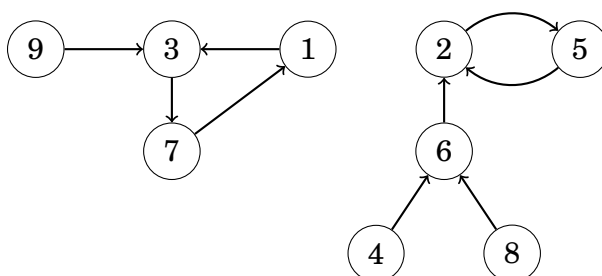
Trong phần còn lại của chương này, ta sẽ tập trung vào *đồ thị mặt trời*. Trong đồ thị này, bậc ra của mỗi đỉnh là 1, tức là, có chính xác một cạnh bắt đầu tại mỗi đỉnh. Một đồ thị mặt trời bao gồm một hoặc nhiều thành phần liên thông, mỗi thành phần chứa một chu trình và một vài đường đi dẫn đến nó.

Đồ thị mặt trời đôi khi được gọi là *đồ thị hàm*. Lý do cho điều này là bất kỳ đồ thị mặt trời nào cũng tương ứng với một hàm định nghĩa cạnh của đồ thị. Tham số cho hàm là một đỉnh của đồ thị, và hàm trả về đỉnh kế tiếp của đỉnh đó.

Ví dụ, hàm

$x$	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

định nghĩa một đồ thị mặt trời như sau:



Bởi vì mỗi đỉnh của một đồ thị mặt trời có duy nhất một đỉnh kế tiếp, có thể định nghĩa một hàm  $\text{succ}(x, k)$  cho biết đỉnh ta sẽ đến được nếu như bắt đầu từ đỉnh  $x$  và đi  $k$  bước theo cạnh đồ thị. Ví dụ, trong đồ thị trên,  $\text{succ}(4, 6) = 2$ , vì ta sẽ đến đỉnh 2 sau 6 bước đi từ đỉnh 4:



Một cách đơn giản để tính giá trị của  $\text{succ}(x, k)$  là mô phỏng lại việc đi  $k$  bước bắt đầu từ đỉnh  $x$  theo đúng định nghĩa, trong  $O(k)$ . Tuy nhiên, sử dụng tiền xử lý, bất kỳ giá trị nào của  $\text{succ}(x, k)$  cũng có thể tính được chỉ trong độ phức tạp  $O(\log k)$ .

Ý tưởng là sử dụng tiền xử lý để tính toán tất cả các giá trị  $\text{succ}(x, k)$  mà tại đó  $k$  là một lũy thừa của hai và không vượt quá  $u$ , với  $u$  là số bước đi tối đa. Việc tính toán này có thể được thực hiện một cách hiệu quả bằng hệ thức truy hồi sau:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Tiền xử lý những giá trị này tốn độ phức tạp  $O(n \log u)$ , vì có  $O(\log u)$  giá trị được tính cho mỗi đỉnh. Trong đồ thị trên, ta có một vài 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, có thể tính được giá trị  $\text{succ}(x, k)$  bất kỳ bằng cách biểu diễn số bước  $k$  dưới dạng tổng các lũy thừa của hai. Ví dụ, nếu muốn tính giá trị của  $\text{succ}(x, 11)$ , trước tiên biểu diễn  $11 = 8 + 2 + 1$ . Từ đó ta có,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Ví dụ, trong đồ thị trên

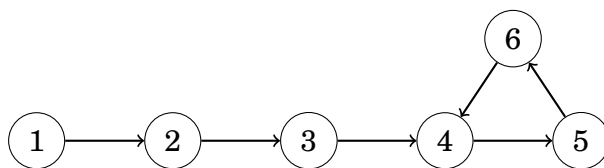
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Một biểu diễn như thế luôn bao gồm  $O(\log k)$  số hạng, vì vậy việc tính giá trị  $\text{succ}(x, k)$  có độ phức tạp là  $O(\log k)$ .

## 16.4 Phát hiện chu trình

Xét một đồ thị mặt trời chỉ chứa một đường đi kết thúc trong một chu trình. Ta có thể hỏi những câu như sau: nếu bắt đầu đi từ đỉnh xuất phát, thì gặp đỉnh đầu tiên thuộc chu trình là đỉnh nào, và chu trình có bao nhiêu đỉnh?

Ví dụ, trong đồ thị



bắt đầu đi từ đỉnh 1, đỉnh đầu tiên trong chu trình là đỉnh 4, và chu trình có 3 đỉnh (4, 5 và 6).

Một cách đơn giản để phát hiện chu trình là đi dọc theo đồ thị và theo dõi tất cả các đỉnh đã được thăm. Ngay khi phát hiện một đỉnh được thăm lần thứ hai, ta có thể kết luận rằng đỉnh đó là đỉnh đầu tiên trong chu trình. Phương pháp này có độ phức tạp thời gian và bộ nhớ đều là  $O(n)$ .

Tuy nhiên, có các thuật toán tốt hơn để phát hiện chu trình. Độ phức tạp thời gian của các thuật toán này vẫn là  $O(n)$ , nhưng chúng chỉ sử dụng  $O(1)$  bộ nhớ. Đây là một cải tiến quan trọng nếu  $n$  lớn. Phần tiếp theo sẽ giới thiệu thuật toán Floyd thỏa mãn những tính chất này.

## Thuật toán Floyd

Thuật toán Floyd<sup>2</sup> đi lần theo đồ thị bằng hai con trỏ  $a$  và  $b$ . Cả hai con trỏ bắt đầu từ một đỉnh  $x$  là đỉnh xuất phát của đồ thị. Sau đó, ở mỗi lượt, con trỏ  $a$  đi một bước và con trỏ  $b$  đi hai bước. Quá trình lặp lại cho đến khi hai con trỏ gặp nhau:

```

a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}

```

Ở thời điểm này, con trỏ  $a$  đã đi  $k$  bước và con trỏ  $b$  đã đi  $2k$  bước, vì vậy độ dài của chu trình phải chia hết cho  $k$ . Do đó, đỉnh đầu tiên trong chu trình có thể được tìm thấy bằng cách dời con trỏ  $a$  đến đỉnh  $x$  và tịnh tiến cả hai con trỏ từng bước một cho đến khi chúng gặp nhau.

```

a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;

```

Sau đó, độ dài của chu trình có thể được tính như sau:

<sup>2</sup>Ý tưởng của thuật toán được đề cập trong [46] và được ghi nhận là của R. W. Floyd; tuy nhiên, không biết Floyd có thực sự phát hiện ra thuật toán này hay không.



```
b = succ(a);  
length = 1;  
while (a != b) {  
    b = succ(b);  
    length++;  
}
```

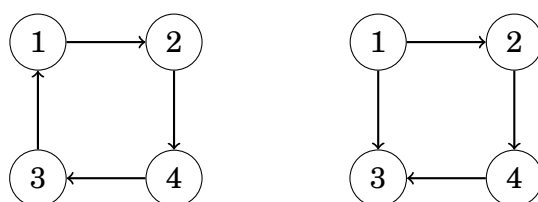


# Chương 17

## Tính liên thông mạnh

Trong một đồ thị có hướng, các cạnh chỉ có thể được duyệt theo một hướng duy nhất, cho nên ngay cả khi đồ thị là liên thông, điều này không đảm bảo rằng sẽ tồn tại đường đi từ một đỉnh này đến một đỉnh khác. Vì vậy, ta cần định nghĩa một khái niệm mới "mạnh" hơn tính liên thông.

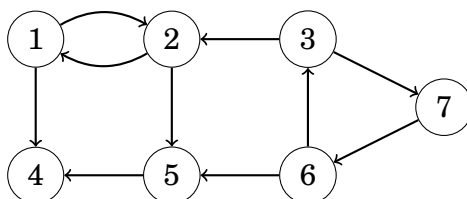
Một đồ thị được gọi là **liên thông mạnh** nếu tồn tại đường đi từ một đỉnh bất kì đến tất cả các đỉnh còn lại trong đồ thị. Ví dụ, trong hình sau, đồ thị bên trái liên thông mạnh, còn đồ thị bên phải thì không.



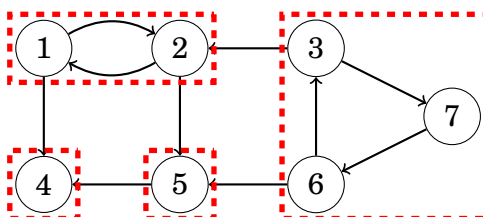
Đồ thị bên phải không liên thông mạnh, bởi vì, ví dụ, từ đỉnh 2 không có đường đi đến đỉnh 1.

Các **thành phần liên thông mạnh** của một đồ thị chia nó thành các phần liên thông mạnh sao cho mỗi phần gồm nhiều đỉnh nhất có thể. Các thành phần liên thông mạnh tạo thành một **đồ thị thành phần** phi chu trình, biểu diễn cấu trúc *ẩn* của đồ thị gốc.

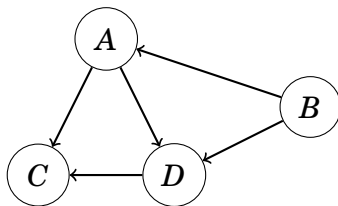
Ví dụ, với đồ thị



các thành phần liên thông mạnh của đồ thị là:



Đồ thị thành phần liên thông tương ứng như sau:



Các thành phần liên thông là  $A = \{1, 2\}$ ,  $B = \{3, 6, 7\}$ ,  $C = \{4\}$  và  $D = \{5\}$ .

Đồ thị thành phần liên thông là một đồ thị có hướng, phi chu trình, do đó khiến việc xử lý dễ dàng hơn đồ thị gốc. Do đồ thị không chứa chu trình, ta luôn có thể xác định một thứ tự sắp xếp topo, và ứng dụng kĩ thuật quy hoạch động như đã mô tả ở Chương 16.

## 17.1 Thuật toán Kosaraju

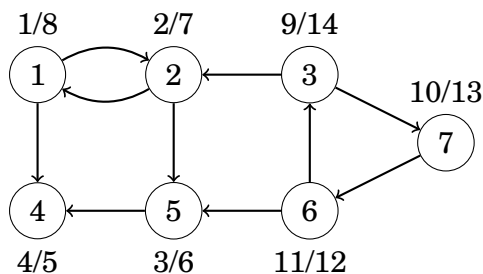
**Thuật toán Kosaraju**<sup>1</sup> là một phương pháp hiệu quả để xác định các thành phần liên thông mạnh của một đồ thị có hướng.

Thuật toán thực hiện hai lần duyệt theo chiều sâu (DFS): lần duyệt đầu tiên lập một danh sách các đỉnh dựa trên cấu trúc của đồ thị, và lần duyệt thứ hai xác định các thành phần liên thông mạnh.

### Lần duyệt thứ 1

Pha đầu tiên trong thuật toán Kosaraju là lập một danh sách các đỉnh theo thứ tự mà thuật tìm kiếm theo chiều sâu DFS duyệt chúng. Thuật toán duyệt qua các đỉnh và thực hiện DFS từ các đỉnh chưa được xử lý. Mỗi đỉnh sẽ được thêm vào danh sách sau khi được nó đã được xử lý xong.

Trong ví dụ dưới đây, các đỉnh của đồ thị được xử lý theo thứ tự sau:



Kí hiệu  $x/y$  có nghĩa là đỉnh bắt đầu được xử lý tại thời điểm  $x$  và hoàn tất tại thời điểm  $y$ . Từ đó, lập được danh sách tương ứng như sau:

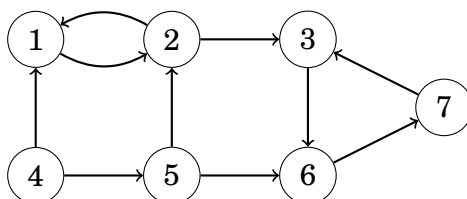
<sup>1</sup>Theo [1], S. R. Kosaraju đã phát minh ra thuật toán này vào năm 1978 nhưng không công bố. Đến năm 1981, chính thuật toán đó đã được tìm ra và công bố bởi M. Sharir [57].

đỉnh	thời gian xử lí
4	5
5	6
2	7
1	8
6	12
7	13
3	14

## Lần duyệt thứ 2

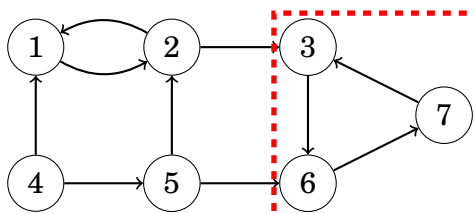
Pha thứ hai của thuật toán sẽ xác định các thành phần liên thông mạnh của đồ thị. Đầu tiên, thuật toán đảo tất cả các cạnh của đồ thị. Điều này đảm bảo trong lượt duyệt thứ hai, ta luôn xác định các thành phần liên thông mạnh mà không bị dư đỉnh.

Sau khi đảo các cạnh, đồ thị ví dụ trở thành:



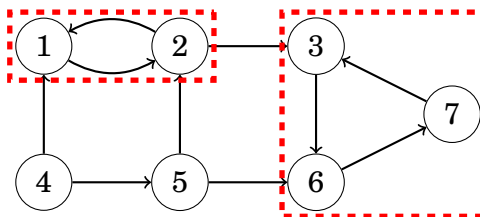
Sau đó, thuật toán duyệt qua danh sách các đỉnh vừa được tạo ra từ lượt đầu tiên, theo thứ tự *ngược lại*. Nếu một đỉnh chưa thuộc thành phần liên thông nào, thuật toán sẽ tạo một thành phần mới và bắt đầu DFS để thêm tất cả các đỉnh tìm được trong quá trình duyệt vào thành phần mới này.

Trong đồ thị ví dụ, thành phần đầu tiên bắt đầu tại đỉnh 3:

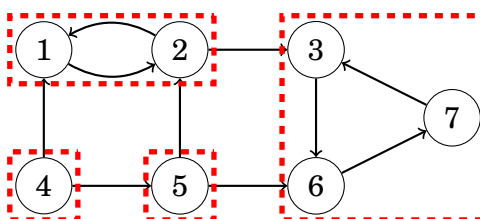


Lưu ý rằng do các cạnh đều được đảo, thành phần liên thông được xác định sẽ không lấy "dư" đỉnh từ các thành phần khác trong đồ thị.

Các đỉnh tiếp theo trong danh sách là đỉnh 7 và 6, tuy nhiên chúng đã được xác định vào một thành phần liên thông mạnh, do đó thành phần mới sẽ được duyệt từ đỉnh 1:



Cuối cùng, thuật toán xử lý đỉnh 5 và 4 để xác định các thành phần liên thông còn lại.



Độ phức tạp thời gian của thuật toán là  $O(n + m)$ , vì thuật toán thực hiện hai lần duyệt theo chiều sâu.

## 17.2 Bài toán 2SAT

Tính liên thông mạnh cũng được ứng dụng trong **bài toán 2SAT**<sup>2</sup>

Trong bài toán này, ta được cho một biểu thức logic

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

trong đó,  $a_i$  và  $b_i$  hoặc là 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 kí hiệu " $\wedge$ " và " $\vee$ " thể hiện các toán tử logic "and" và "or". Nhiệm vụ của ta là gán mỗi biến một giá trị sao cho biểu thức đúng hoặc cho biết điều này bất khả thi.

Ví dụ, biểu thức

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

biểu thức đúng khi các biến được gán như sau:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

<sup>2</sup>Thuật toán được trình bày ở đây đã được giới thiệu tại [4]. Ngoài ra, còn có một thuật toán nổi tiếng có độ phức tạp tuyến tính khác [19] dựa trên quay lui.

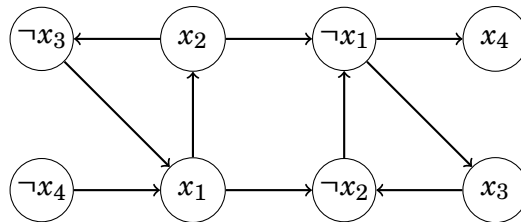
Tuy nhiên, biểu thức

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

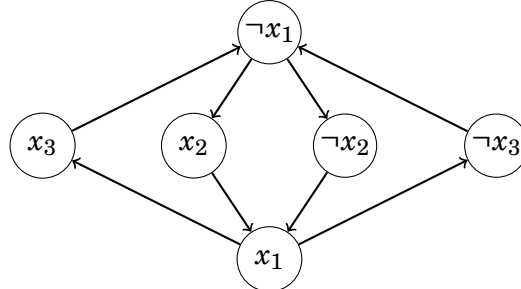
luôn sai, cho dù ta có gán như thế nào đi nữa. Lí do là ta không thể chọn một giá trị cho  $x_1$  mà không xuất hiện mâu thuẫn. Nếu  $x_1$  là sai, cả  $x_2$  và  $\neg x_2$  đều phải đúng (bất khả thi). Nếu  $x_1$  là đúng, cả  $x_3$  lẫn  $\neg x_3$  đều phải đúng (bất khả thi).

Bài toán 2SAT có thể được biểu diễn bằng một đồ thị có các đỉnh tương ứng với các biến  $x_i$  và các phủ định  $\neg x_i$ , và các cạnh thể hiện mối quan hệ giữa các biến. Mỗi cặp  $(a_i \vee b_i)$  tạo thành hai cạnh:  $\neg a_i \rightarrow b_i$  và  $\neg b_i \rightarrow a_i$ . Điều này cho biết nếu  $a_i$  không thỏa mãn, thì  $b_i$  bắt buộc thỏa, và ngược lại.

Đồ thị tương ứng với biểu thức  $L_1$  là:



Và đồ thị cho biểu thức  $L_2$  là:



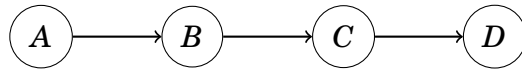
Cấu trúc của đồ thị cho ta biết ta có thể gán các giá trị cho các biến để thỏa mãn biểu thức hay không. Thật ra, ta luôn có thể xác định được cách gán thỏa mãn khi không tồn tại cặp đỉnh  $x_i$  và  $\neg x_i$  nào thuộc cùng một thành phần liên thông mạnh. Nếu hai đỉnh như vậy tồn tại, đồ thị sẽ chứa một đường đi từ  $x_i$  đến  $\neg x_i$  và một đường đi từ  $\neg x_i$  đến  $x_i$ , vì vậy cả  $x_i$  lẫn  $\neg x_i$  đều phải đúng, điều này là bất khả thi.

Trong đồ thị của biểu thức  $L_1$ , không có cặp đỉnh  $x_i$  và  $\neg x_i$  nào thuộc cùng một thành phần liên thông mạnh, do đó tồn tại lời giải cho biểu thức. Trong đồ thị của biểu thức  $L_2$ , tất cả các nút đều nằm trong cùng một thành phần liên thông mạnh, do đó không tồn tại lời giải cho biểu thức.

Nếu một lời giải tồn tại, ta có thể tìm giá trị cho các biến bằng cách duyệt các đỉnh của đồ thị thành phần theo thứ tự topo ngược. Tại mỗi bước, ta xử lí một thành phần liên thông mà không có cung đến một thành phần chưa xét. Nếu các biến trong thành phần liên thông chưa được gán giá trị, chúng sẽ được xác định dựa theo các giá trị trong thành phần liên thông đó, và

nếu các biến đã được gán trước đó, giá trị của chúng sẽ được giữ nguyên. Quá trình xử lý tiếp tục đến khi các biến đều được gán giá trị.

Đồ thị thành phần cho biểu thức  $L_1$  như sau:



Các thành phần liên thông 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 dựng lời giải, đầu tiên ta xử lý thành phần liên thông  $D$  và gán  $x_4$  thành đúng (true). Sau đó, ta xử lý các thành phần liên thông  $C$ , gán  $x_1$  và  $x_2$  thành sai (false), và  $x_3$  thành đúng.

Toàn bộ các biến đều đã được gán giá trị, do đó các thành phần còn lại là  $A$  và  $B$  sẽ không thay đổi giá trị các biến.

Lưu ý rằng, phương pháp này đúng đắn, bởi vì đồ thị có một cấu trúc đặc biệt: nếu tồn tại đường đi từ đỉnh  $x_i$  đến đỉnh  $x_j$  và từ đỉnh  $x_j$  đến đỉnh  $\neg x_j$ , thì đỉnh  $x_i$  sẽ không bao giờ đúng. Lý do là nếu vậy, thì đồ thị cũng sẽ tồn tại đường đi từ đỉnh  $\neg x_j$  đến đỉnh  $\neg x_i$ , và cả  $x_i$  và  $x_j$  sẽ được gán thành sai.

Một phiên bản khó hơn của bài toán là **bài toán 3SAT**, trong đó các điều kiện trong biểu thức có dạng  $(a_i \vee b_i \vee c_i)$ . Đây là một bài toán NP-khó, vì thế chưa có thuật toán hiệu quả nào được tìm ra.



# Chương 18

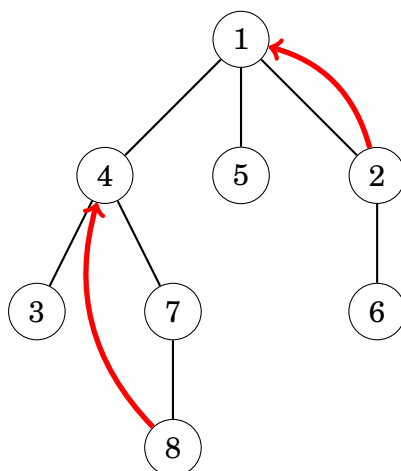
## Truy vấn trên cây

Chương này sẽ giới thiệu kỹ thuật xử lý các truy vấn trên cây con và đường đi của một cây có gốc. Ví dụ như:

- Tổ tiên thứ  $k$  của một nút là nút nào?
- Tổng giá trị thuộc một cây con bằng bao nhiêu ?
- Tổng giá trị trên đường đi giữa 2 nút bằng bao nhiêu?
- Tổ tiên chung gần nhất của hai nút là nút nào?

### 18.1 Tìm nút tổ tiên

**Tổ tiên** thứ  $k$  của nút  $x$  trong một cây có gốc là nút mà ta sẽ tới được nếu nhảy lên  $k$  bước từ nút  $x$ . Gọi  $\text{ancestor}(x, k)$  là tổ tiên thứ  $k$  của nút  $x$  (hoặc 0 nếu không tồn tại tổ tiên như vậy). Lấy ví dụ như cây sau:  $\text{ancestor}(2, 1) = 1$  và  $\text{ancestor}(8, 2) = 4$ .



Một cách dễ dàng để tính bất kì giá trị  $\text{ancestor}(x, k)$  nào là nhảy lần lượt một dãy  $k$  bước trên cây. Tuy nhiên, độ phức tạp của phương pháp này

là  $O(k)$ , có thể sẽ khá chậm bởi vì một cây  $n$  nút có thể suy biến thành một chuỗi  $n$  nút liên tiếp.

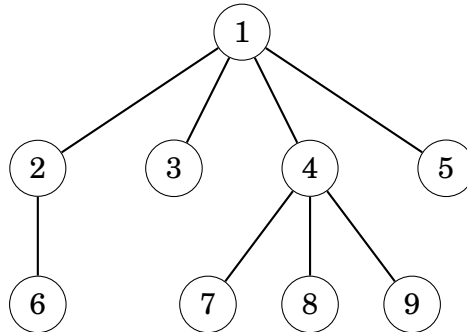
May mắn thay, sử dụng một cách tương tự như đã làm ở Chương 16.3, bất kỳ giá trị  $\text{ancestor}(x, k)$  nào cũng tính được một cách hiệu quả trong thời gian  $O(\log k)$  sau khi tiền xử lý. Ý tưởng là tính trước mọi giá trị  $\text{ancestor}(x, k)$  với  $k \leq n$  là lũy thừa của hai. Ví dụ, các giá trị tính được 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
...								

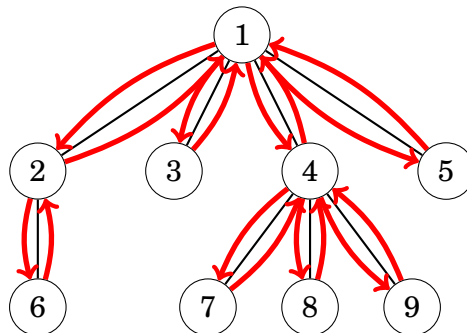
Quá trình tiền xử lý mất độ phức tạp là  $O(n \log n)$  vì có  $O(\log n)$  giá trị cần được tính với mỗi nút. Sau đó, bất kỳ giá trị  $\text{ancestor}(x, k)$  nào cũng có thể tính được trong độ phức tạp thời gian  $O(\log k)$  bằng cách biểu diễn  $k$  thành một tổng các lũy thừa của 2.

## 18.2 Cây con và đường đi trên cây

**Mảng duyệt cây** chứa các nút của một cây có gốc theo thứ tự mà chúng được thăm khi duyệt theo chiều sâu từ nút gốc. Ví dụ trong cây:



Quá trình duyệt theo chiều sâu diễn ra như sau:



Do đó, có mảng duyệt cây tương ứng như sau:

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

## Truy vấn cây con

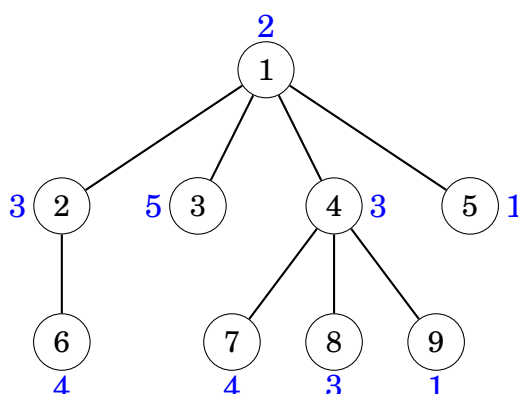
Mỗi cây con tương ứng với một đoạn con liên tiếp trong mảng duyệt cây, mà phần tử đầu tiên của đoạn đó chính là nút gốc của cây con này. Ví dụ, đoạn con liên tiếp sau chứa các nút con của cây con gốc 4:

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

Tận dụng điều này, ta có thể xử lý truy vấn liên quan tới cây con một cách hiệu quả. Ví dụ, xét bài toán mà tại đó mỗi nút được gán một giá trị và nhiệm vụ của ta là xử lý các truy vấn sau:

- Cập nhật giá trị của một nút.
- Tính tổng giá trị thuộc cây con tại một nút.

Xét cây sau với những số màu xanh là giá trị của các nút tương ứng. Ví dụ, tổng của cây con gốc 4 là  $3 + 4 + 3 + 1 = 11$ .



Ý tưởng là tạo ra một mảng duyệt cây chứa ba giá trị với mỗi nút: Định danh của nút, kích thước cây con (lấy nút đó làm gốc), và giá trị của nút. Ví dụ, mảng cho cây trên trông như sau:

định danh nút	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
giá trị của nút	2	3	4	5	3	4	3	1	1

Sử dụng mảng này, ta có thể tính tổng của bất kỳ cây con nào bằng cách trước tiên xác định kích thước của nó, sau đó xác định giá trị của các nút tương ứng. Ví dụ, giá trị của cây con gốc 4 có thể được tính như sau:

định danh nút	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
giá trị của nút	2	3	4	5	3	4	3	1	1

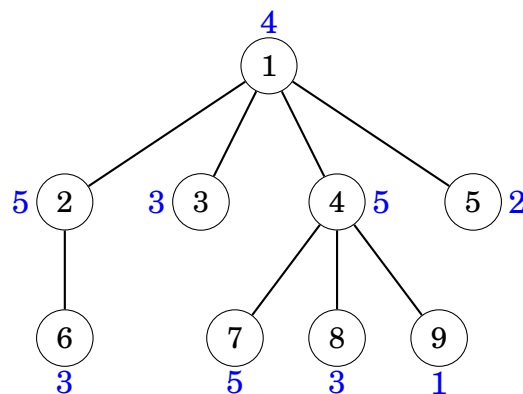
Để trả lời các truy vấn này một cách hiệu quả, chỉ cần lưu ba giá trị của mỗi nút trên một cây chỉ số nhị phân hoặc cây phân đoạn là đủ. Sau đó, ta có thể thực hiện cả hai thao tác cập nhật giá trị và tính tổng trong độ phức tạp thời gian là  $O(\log n)$ ,

## Truy vấn đường đi

Sử dụng mảng duyệt cây, ta có thể tính tổng các giá trị trên đường đi từ nút gốc tới bất kỳ nút nào trong cây một cách hiệu quả. Xét bài toán mà trong đó, ta cần thực hiện các truy vấn sau:

- Thay đổi giá trị của một nút.
- Tính tổng giá trị trên đường đi từ nút gốc tới một nút bất kỳ.

Ví dụ, với cây sau đây thì tổng giá trị đường đi từ nút gốc tới nút 7 là  $4 + 5 + 5 = 14$ :



Ta có thể giải bài này như trước, nhưng bây giờ mỗi giá trị trong hàng cuối cùng của mảng mang ý nghĩa là tổng giá trị của các nút trên đường đi từ nút gốc tới nút đó. Ví dụ, mảng sau tương ứng với cây ở trên:

định danh của nút	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	4	9	12	7	9	14	12	10	6

Khi giá trị của một nút tăng lên  $x$ , tổng đường đi của mọi nút trong cây con của nó tăng lên  $x$ . Ví dụ, nếu như giá trị của nút 4 tăng lên 1, thì mảng sẽ thay đổi như sau:

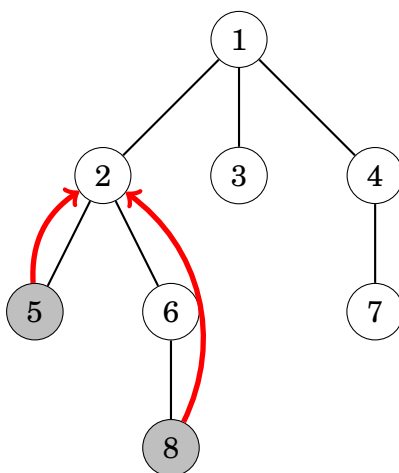
định danh của nút	1	2	6	3	4	7	8	9	5
kích thước cây con	9	2	1	1	4	1	1	1	1
tổng đường đi	4	9	12	7	10	15	13	11	6

Vì vậy, để hỗ trợ cả hai thao tác, ta phải làm được hai việc: tăng giá trị cho một đoạn liên tiếp và truy xuất giá trị tại 1 vị trí bất kỳ. Việc này có thể hoàn thành trong độ phức tạp thời gian là  $O(\log n)$  sử dụng cây chỉ số nhị phân hoặc cây phân đoạn (xem ở Chương 9.4).

## 18.3 Tổ tiên chung gần nhất

**Tổ tiên chung gần nhất** của hai nút trong một cây có gốc là nút thấp nhất mà cây con của nó chứa cả hai nút đang xét. Một bài toán điển hình là trả lời nhanh chóng một loạt các truy vấn tìm tổ tiên chung gần nhất của 2 nút bất kỳ

Ví dụ, trong cây sau, tổ tiên chung gần nhất của nút 5 và 8 là nút 2:



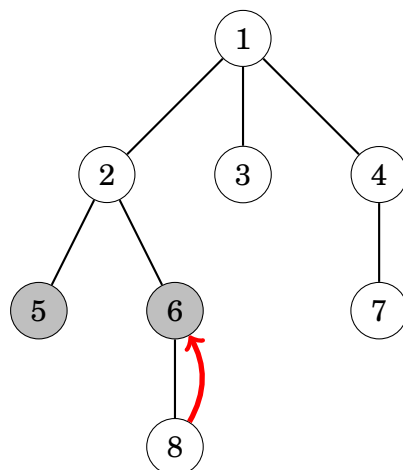
Tiếp theo ta sẽ thảo luận hai kĩ thuật hiệu quả để tìm tổ tiên chung gần nhất của hai nút.

### Cách 1

Có một cách để giải quyết vấn đề, cách này dựa trên thực tế rằng ta có thể tìm nhanh tổ tiên thứ  $k$  của nút bất kì trong cây. Tận dụng điều đó, ta có thể chia bài toán thành hai phần.

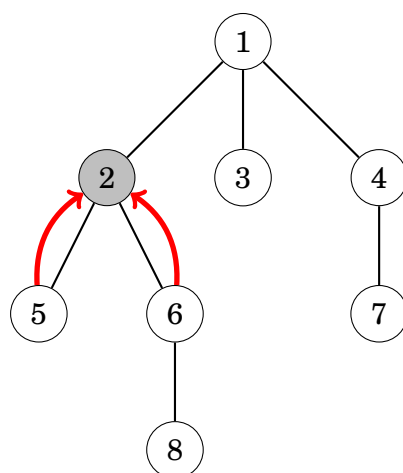
Ta dùng hai con trỏ, ban đầu chúng sẽ trỏ vào hai nút mà ta đang quan tâm - cần tìm tổ tiên chung gần nhất của chúng. Đầu tiên, chúng ta di chuyển một trong hai con trỏ đi lên sao cho hai con trỏ đều chỉ vào hai nút ở cùng tầng.

Trong trường hợp ví dụ, chúng ta di chuyển con trỏ thứ hai lên một tầng để nó chỉ vào nút 6 cùng tầng với nút 5:



Sau đó, ta cần định rõ số bước nhỏ nhất cần để hai con trỏ nhảy lên để chúng cùng trở vào một nút. Nút mà cả hai con trỏ đều chỉ vào sau khi nhảy lên chính là tổ tiên chung gần nhất.

Trong trường hợp ví dụ, chỉ cần di chuyển cả hai con trỏ một bước là đủ lên tới nút 2, đó là tổ tiên chung gần nhất:

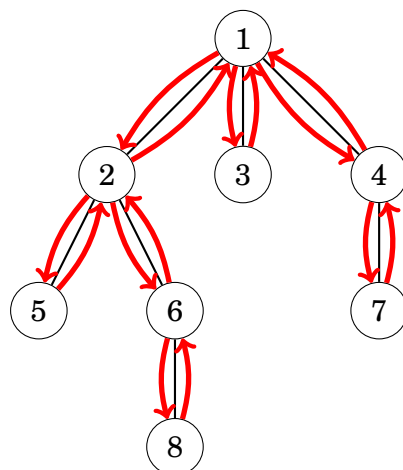


Vì cả hai phần của thuật toán có thể thực hiện trong độ phức tạp thời gian  $O(\log n)$  dựa vào những thông tin đã xử lý trước, chúng ta có thể tìm tổ tiên chung gần nhất của bất kỳ cặp nút nào trong thời gian  $O(\log n)$

## Cách 2

Một cách khác để giải quyết vấn đề này là dựa vào mảng duyệt cây.<sup>1</sup> Một lần nữa, ý tưởng là duyệt qua các nút bằng phép duyệt chiều sâu:

<sup>1</sup>Thuật toán tổ tiên chung thấp nhất này đã được trình bày trong [7]. Đôi khi được gọi là **Đường đi Euler trên cây** [66].



Tuy nhiên, chúng ta sử dụng một mảng duyệt cây khác với lúc trước: ta thêm các nút vào mảng *mỗi* khi phép duyệt theo chiều sâu thăm qua nút đó, không chỉ ở lần đầu tiên thăm. Do đó, một nút có  $k$  con sẽ xuất hiện  $k + 1$  lần trong mảng và có tổng cộng là  $2n - 1$  nút trong mảng.

Chúng ta lưu hai giá trị trong mảng: Định danh của nút và độ sâu của nút trong cây. Mảng sau sẽ tương ứng với cây trên: >>>>> main

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
định danh nút	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
độ sâu	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Bây giờ chúng ta có thể tìm tổ tiên chung gần nhất của nút  $a$  và  $b$  bằng cách tìm nút với độ sâu *nhỏ nhất* trong mảng nằm giữa  $a$  và  $b$ . Ví dụ như, tổ tiên chung gần nhất của nút 5 và 8 có thể được tìm như sau:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
định danh nút	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

↑

Nút 5 ở vị trí 2, nút 8 ở vị trí 5, và nút với độ sâu nhỏ nhất trong phạm vi 2...5 là nút 2 ở vị trí 3 với độ sâu là 2. Vì vậy, tổ tiên chung gần nhất của nút 5 và 8 là nút 2.

Vì thế, để tìm tổ tiên chung gần nhất của hai nút thì cần phải tìm cực tiểu trên một đoạn.

Vì mảng là mảng tĩnh, nên ta có thể trả lời các truy vấn như vậy trong độ phức tạp thời gian  $O(1)$  sau khi tiền xử lý với độ phức tạp thời gian  $O(n \log n)$ .

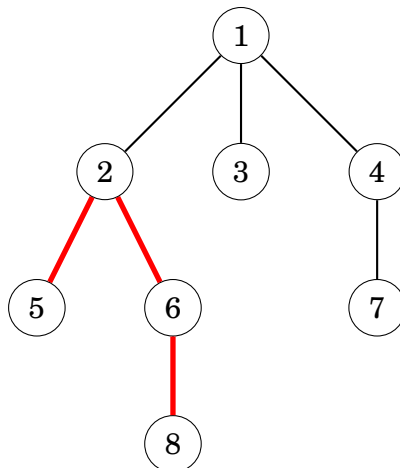
## Khoảng cách giữa các nút

Khoảng cách giữa nút  $a$  và nút  $b$  bằng chính độ dài đường đi từ  $a$  tới  $b$ . Thực chất, bài toán tính khoảng cách giữa các nút có thể quy về tìm tổ tiên chung gần nhất của chúng.

Đầu tiên, chúng ta chọn gốc cây tùy ý. Sau đó, khoảng cách từ nút  $a$  tới  $b$  có thể tính bằng công thức

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c),$$

với  $c$  là tổ tiên chung gần nhất của  $a$  và  $b$  và  $\text{depth}(s)$  cho biết độ sâu của nút  $s$ . Ví dụ, xét khoảng cách của hai nút 5 và 8 sau :



Tổ tiên chung gần nhất của nút 5 và 8 là nút 2. Độ sâu của các nút là  $\text{depth}(5) = 3$ ,  $\text{depth}(8) = 4$  and  $\text{depth}(2) = 2$ , vậy nên khoảng cách giữa hai nút 5 và 8 là  $3 + 4 - 2 \cdot 2 = 3$ .

## 18.4 Xử lý Ngoại tuyến

Cho đến giờ, chúng ta đã thảo luận về các thuật toán *trực tuyến* dùng cho truy vấn cây. Các thuật toán này có thể xử lý lần lượt từng truy vấn nên mỗi truy vấn đều có thể trả lời được trước khi nhận truy vấn tiếp theo.

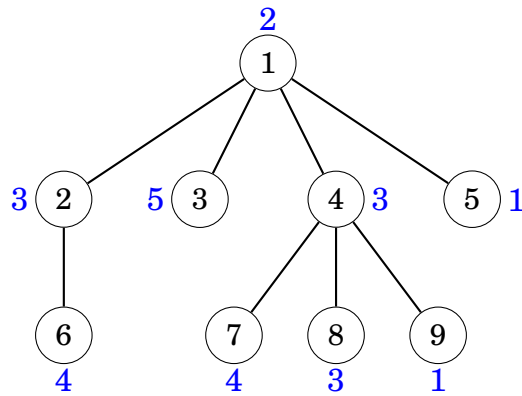
Tuy nhiên, trong nhiều bài tập, ta không cần tính chất trực tuyến này. Trong phần này, chúng ta tập trung vào các thuật toán xử lý *ngoại tuyến*. Các thuật toán này có đầu vào là một tập các truy vấn có thể trả lời theo bất kỳ thứ tự nào. Xử lý ngoại tuyến thường sẽ dễ cài đặt hơn so với xử lý trực tuyến.

### Cấu trúc kết hợp

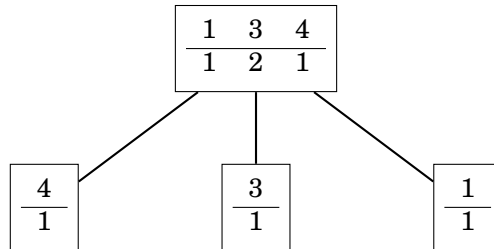
Một cách xây dựng nên một thuật toán ngoại tuyến là thực hiện duyệt theo chiều sâu trên cây và duy trì các cấu trúc dữ liệu trên các nút. Với mỗi nút  $s$ , ta tạo một cấu trúc dữ liệu  $d[s]$  dựa trên các cấu trúc dữ liệu của các con của  $s$ . Sau đó, dùng cấu trúc dữ liệu này, các truy vấn liên quan tới  $s$  sẽ được xử lý.

Ví dụ, xét bài toán sau: Ta được cho một cây, mỗi nút trên cây được gán một giá trị. Việc của ta là phải xử lý truy vấn dưới dạng: "Đếm số nút mang giá trị  $x$  trong cây con gốc  $s$ ". Ví dụ, trong cây sau, cây con gốc 4 chứa hai nút có giá trị là 3.





Trong bài này, ta có thể dùng kiểu dữ liệu ánh xạ để trả lời truy vấn. Ví dụ, ánh xạ cho nút 4 và các con của nó trông như sau:



Nếu chúng ta tạo một cấu trúc như vậy cho mỗi nút, ta có thể dễ dàng trả lời tất cả các truy vấn, bởi vì ta có thể xử lý được các truy vấn liên quan đến một nút ngay sau khi tạo cấu trúc dữ liệu của nó. Ví dụ, cấu trúc ánh xạ trên của nút 4 cho ta biết cây con của nó chứa 2 nút có giá trị 3.

Tuy nhiên, việc tạo mỗi cấu trúc dữ liệu từ đầu là quá chậm. Thay vào đó, với mỗi nút  $s$  ta khởi tạo một cấu trúc  $d[s]$  ban đầu chỉ chứa giá trị nút  $s$ . Sau đó, ta duyệt qua các nút con  $u$  của  $s$  và *kết hợp*  $d[s]$  với tất cả các cấu trúc  $d[u]$ .

Ví dụ với cây trên, cấu trúc ánh xạ cho nút 4 được tạo bằng cách kết hợp các cấu trúc ánh xạ sau:



Ở đây ánh xạ đầu tiên là cấu trúc dữ liệu khởi tạo của nút 4, và ba ánh xạ còn lại tương ứng với các nút 7, 8 và 9.

Việc gộp nút tại  $s$  có thể được xử lý như sau: Ta duyệt qua các con của  $s$  và tại mỗi nút con  $u$  ta kết hợp  $d[s] \gg \gg \gg$  main với  $d[u]$ . Chúng ta luôn sao chép nội dung từ  $d[u]$  sang  $d[s]$ . Tuy nhiên, trước đó, ta *đổi* giá trị của  $d[s]$  và  $d[u]$  cho nhau nếu  $d[s]$  nhỏ hơn  $d[u]$ . Bằng cách này, mỗi giá trị chỉ bị sao chép  $O(\log n)$  lần suốt quá trình duyệt cây, đảm bảo sự hiệu quả của thuật toán.

Để hoán đổi nội dung của hai cấu trúc ánh xạ  $a$  và  $b$  hiệu quả, ta có thể dùng đoạn mã sau:

```
swap(a,b);
```

Chắc chắn rằng đoạn mã trên có độ phức tạp hằng số khi  $a$  và  $b$  là các cấu trúc dữ liệu thuộc thư viện chuẩn C++

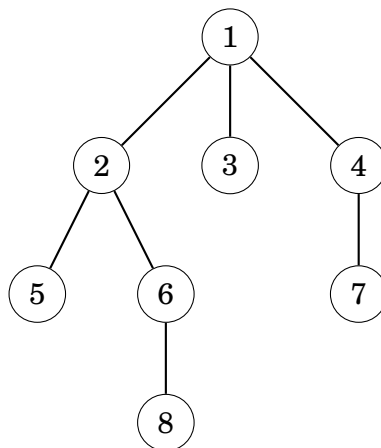
## Tổ tiên chung gần nhất

Có một thuật toán ngoại tuyến có thể xử lý một tập các truy vấn về tổ tiên chung gần nhất<sup>2</sup>. Thuật toán dựa trên cấu trúc dữ liệu các tập không giao nhau (xem ở Chương 15.2), và lợi ích của thuật này là nó dễ cài đặt hơn các thuật toán đã bàn ở chương này.

Thuật toán có đầu vào là một tập các cặp nút, và với mỗi cặp như vậy nó tìm được tổ tiên chung gần nhất của hai nút đó. Thuật toán thực hiện duyệt cây theo chiều sâu và duy trì các tập nút không giao nhau. Ban đầu, mỗi nút chỉ thuộc về một tập riêng biệt. Với mỗi tập, ta cũng lưu trữ nút cao nhất trong cây thuộc vào tập hợp đó.

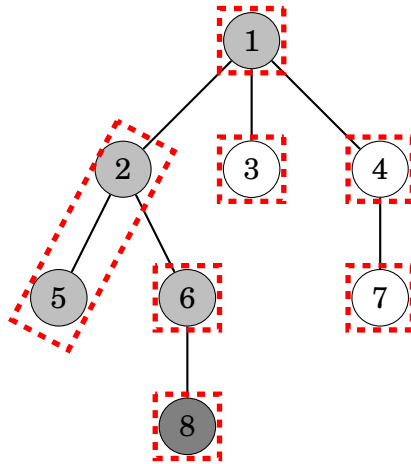
Khi thuật toán thăm nút  $x$ , nó duyệt qua mọi nút  $y$  mà ta cần tìm tổ tiên chung nhỏ nhất của  $x$  và  $y$ . Nếu  $y$  đã được thăm, thuật toán sẽ kết luận tổ tiên chung gần nhất của  $x$  và  $y$  là nút có độ cao lớn nhất trong tập chứa  $y$ . Tiếp đến, sau khi xử lý nút  $x$  thuật toán sẽ hợp hai tập, tập chứa  $x$  và tập chứa cha của nó.

Ví dụ: giả sử chúng ta muốn tìm tổ tiên chung gần nhất của các cặp nút (5, 8) và (2, 7) trong cây sau:

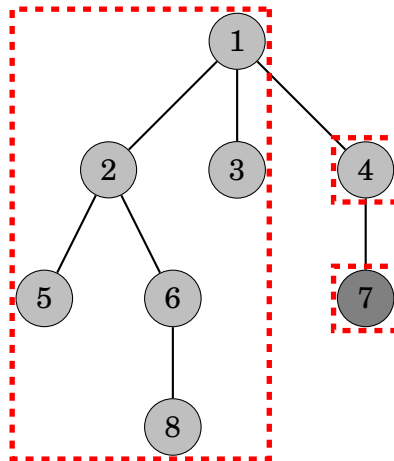


Trong cây sau, màu xám thể hiện các nút đã thăm và nhóm các nút cùng được bọc trong một đường nét đứt thuộc về cùng một tập hợp. Khi thuật toán thăm nút 8, nó biết rằng nút 5 đã được thăm và nút cao nhất trong tập hợp của nó là 2. Vì vậy, tổ tiên con chung nhỏ nhất của nút 5 và 8 là 2:

<sup>2</sup>Thuật toán này được công bố bởi R. E. Tarjan năm 1979 [65].



Sau đó, khi ta thăm nút 7, thuật toán kết luận rằng tổ tiên chung nhỏ nhất của nút 2 và 7 là 1:





# Chương 19

## Đường đi và chu trình

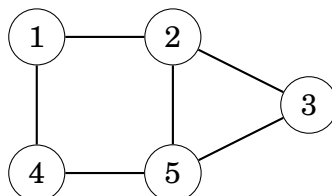
Chương này tập trung vào hai loại đường đi trong đồ thị:

- **đường đi Euler** là đường đi qua tất cả các cạnh, mỗi cạnh đúng một lần.
- **đường đi Hamilton** là đường đi qua tất cả các đỉnh, mỗi đỉnh đúng một lần.

Thoạt nhìn thì đường đi Euler và Hamilton có khái niệm tương tự nhau, tuy nhiên các bài toán liên quan đến chúng lại rất khác nhau. Trên thực tế, có một quy tắc đơn giản để xác định xem một đồ thị có tồn tại đường đi Euler hay không. Và cũng có một thuật toán để tìm ra một đường đi Euler nếu nó tồn tại. Ngược lại, việc kiểm tra sự tồn tại của một đường đi Hamilton là một bài toán NP-khó, và chưa có thuật toán nào có thể giải quyết được nó một cách hiệu quả.

### 19.1 Đường đi Euler

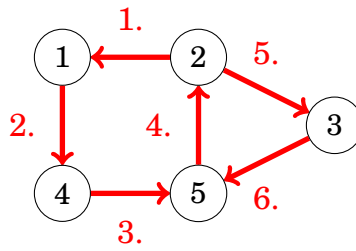
Một **Đường đi Euler**<sup>1</sup> là một đường đi qua mọi cạnh trên đồ thị, mỗi cạnh đúng một lần. Ví dụ, đồ thị



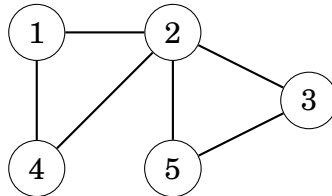
có một đường đi Euler từ đỉnh 2 đến đỉnh 5:

---

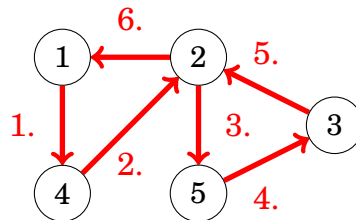
<sup>1</sup>L. Euler từng nghiên cứu những đường đi như vậy vào năm 1736, khi ông giải bài toán 7 cây cầu ở Königsberg nổi tiếng. Đây cũng là sự ra đời của lý thuyết đồ thị.



một **chu trình Euler** là một đường đi Euler bắt đầu và kết thúc tại cùng một đỉnh. Ví dụ, đồ thị



có một chu trình Euler bắt đầu và kết thúc tại đỉnh 1:



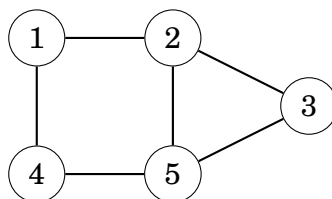
## Sự tồn tại

Sự tồn tại của đường đi và chu trình Euler phụ thuộc vào bậc của các đỉnh trong đồ thị. Đầu tiên, một đồ thị vô hướng tồn tại đường đi Euler khi tất cả các cạnh của đồ thị đều thuộc cùng một thành phần liên thông và

- bậc của mọi đỉnh đều chẵn *hoặc*
- bậc của đúng hai đỉnh là lẻ, và bậc của tất cả các đỉnh khác đều chẵn.

Trong trường hợp đầu tiên, mỗi đường đi Euler cũng là một chu trình Euler. Trong trường hợp thứ hai, hai đỉnh có bậc lẻ chính là hai đỉnh bắt đầu và kết thúc của các đường đi Euler, đường đi này không phải là chu trình như trường hợp 1.

Ví dụ, trong đồ thị



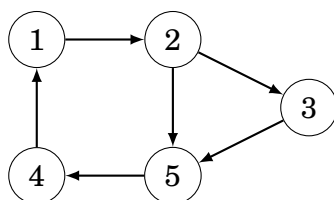
các đỉnh 1, 3 và 4 có bậc là 2, đỉnh 2 và 5 có bậc là 3. Có chính xác hai đỉnh có bậc lẻ, vì vậy có một đường đi Euler giữa hai đỉnh 2 và 5, nhưng đồ thị trên không chứa chu trình Euler nào.

Trong đồ thị có hướng, ta quan tâm đến các bán bậc ra và bán bậc vào của các đỉnh. Một đồ thị có hướng chứa đường đi Euler khi tất cả các cạnh đều thuộc cùng một thành phần liên thông và

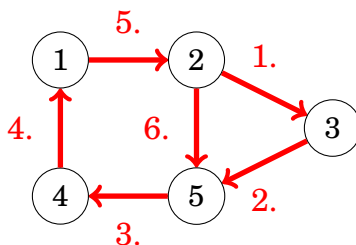
- với mọi đỉnh, bán bậc ra bằng bán bậc vào *hoặc*
- có một đỉnh có bán bậc vào lớn hơn bán bậc ra một đơn vị, và một đỉnh khác có bán bậc vào nhỏ hơn bán bậc ra một đơn vị, trong mọi đỉnh còn lại thì bán bậc ra đúng bằng bán bậc vào

Trong trường hợp đầu tiên, mỗi đường đi Euler cũng là một chu trình Euler. Trong trường hợp thứ hai, đồ thị tồn tại đường đi Euler bắt đầu từ đỉnh có bán bậc ra lớn hơn và kết thúc tại đỉnh có bán bậc vào lớn hơn.

Ví dụ, trong đồ thị



các đỉnh 1, 3 và 4 đều có bán bậc vào là 1 và bán bậc ra là 1, đỉnh 2 có bán bậc vào là 1 và bán bậc ra là 2, và đỉnh 5 có bán bậc vào là 2 và bán bậc ra là 1. Vì vậy, đồ thị trên tồn tại một đường đi Euler từ đỉnh 2 đến đỉnh 5:



## Thuật toán Hierholzer

**Thuật toán Hierholzer**<sup>2</sup> là một phương pháp hiệu quả để xây dựng một chu trình Euler. Thuật toán gồm nhiều bước, mỗi bước sẽ thêm một số cạnh mới vào chu trình. Giả thiết rằng đồ thị có chứa chu trình Euler, nếu không thuật toán Hierholzer không thể tìm được nó.

Trước tiên, thuật toán sẽ xây dựng một chu trình chứa một số (không nhất thiết là tất cả) cạnh của đồ thị. Sau đó, thuật toán mở rộng chu trình đã tạo từng bước một bằng cách thêm những chu trình phụ vào nó. Quá trình này được lặp lại cho đến khi tất cả các cạnh đã được thêm vào chu trình.

Thuật toán Hierholzer mở rộng chu trình bằng cách tìm một đỉnh  $x$  thuộc chu trình nhưng có một cạnh hướng ra chưa được thêm vào chu trình. Thuật toán sẽ xây dựng một đường đi mới chỉ gồm những cạnh chưa thuộc

<sup>2</sup>Thuật toán đã được công bố vào năm 1873, sau khi Hierholzer mất.[35].

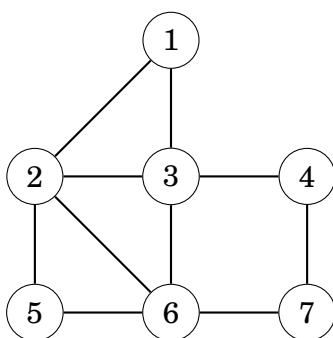
chu trình mà xuất phát từ đỉnh  $x$ . Sớm muộn gì, đường đi cũng sẽ tự trở về đỉnh  $x$  và sẽ tạo ra một chu trình phụ.

Nếu đồ thị chỉ tồn tại đường đi Euler, ta vẫn có thể sử dụng thuật toán Hierholzer để tìm nó bằng cách thêm một cạnh vào đồ thị và loại bỏ cạnh đó sau khi tạo được chu trình. Ví dụ, trong đồ thị vô hướng, ta thêm vào cạnh nối giữa hai đỉnh có bậc lẻ của đồ thị.

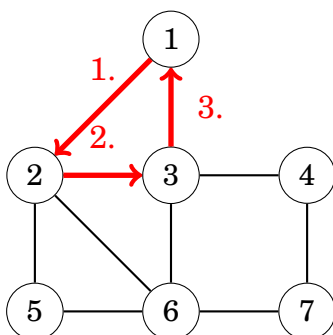
Tiếp theo chúng ta sẽ cùng xem quá trình dựng nên một chu trình Euler trong đồ thị vô hướng của thuật toán Hierholzer.

## Ví dụ

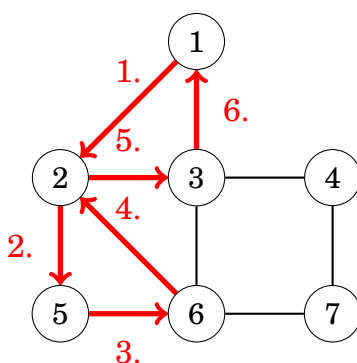
Xét đồ thị sau:



Giả sử thuật toán ban đầu tạo ra một chu trình xuất phát từ đỉnh 1. Để thấy  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$  là một chu trình như vậy:

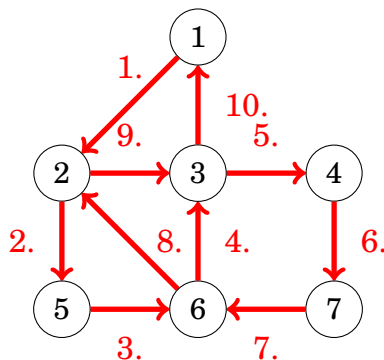


Sau đó, thuật toán sẽ thêm chu trình phụ  $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$  vào:





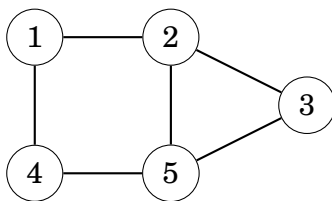
Cuối cùng, thuật toán sẽ thêm chu trình phụ  $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$  vào:



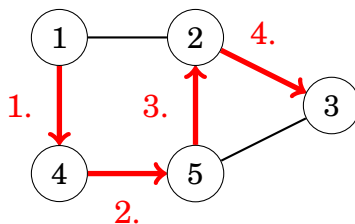
Đến đây thì chu trình được tạo đã bao gồm tất cả các cạnh của đồ thị, vì vậy ta đã thành công tạo ra một chu trình Euler.

## 19.2 Đường đi Hamilton

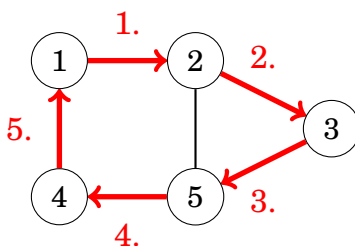
**Đường đi Hamilton** là một đường đi qua tất cả các đỉnh, mỗi đỉnh được thăm đúng một lần. Ví dụ, đồ thị



chứa một đường đi Hamilton từ đỉnh 1 đến đỉnh 3:



Nếu một đường đi Hamilton xuất phát và kết thúc tại một đỉnh thì nó được gọi là một **chu trình Hamilton**. Đồ thị dưới đây cũng có một chu trình Hamilton bắt đầu và kết thúc tại đỉnh 1:



## Sự tồn tại

Ta chưa biết đến phương pháp nào hiệu quả để kiểm tra một đồ thị có chứa đường đi Hamilton hay không, và đây là một bài toán NP-khó. Tuy nhiên, trong một số trường hợp đặc biệt, ta có thể chắc chắn rằng một đồ thị chứa đường đi Hamilton.

Có một nhận xét đơn giản rằng nếu một đồ thị là đầy đủ, tức có một cạnh giữa mọi cặp đỉnh, thì nó chứa một đường đi Hamilton. Hơn nữa, người ta cũng đã chứng minh được một số định lý "mạnh" hơn:

- **Định lý Dirac:** Nếu mỗi đỉnh có bậc tối thiểu là  $n/2$ , đồ thị chứa một đường đi Hamilton.
- **Định lý Ore:** Nếu tổng bậc của mỗi cặp đỉnh không kề nhau của đồ thị không bé hơn  $n$ , đồ thị chứa một đường đi Hamilton.

Một đặc điểm chung của hai định lý trên và những kết luận khác là chúng bảo đảm sự tồn tại của đường đi Hamilton nếu đồ thị có *một lượng lớn* cạnh. Điều này là có lý, vì đồ thị chứa càng nhiều cạnh thì khả năng để tạo ra một đường đi Hamilton là càng lớn.

## Cách xây dựng

Vì không có cách nào hiệu quả để kiểm tra sự tồn tại của đường đi Hamilton, rõ ràng không có phương pháp nào có thể xây dựng được đường đi Hamilton một cách hiệu quả, bởi nếu có thì ta chỉ việc thử xây dựng một đường đi Hamilton và thấy được nó có tồn tại hay không (nếu không dựng được đường đi nào thì chứng tỏ là không tồn tại).

Một cách đơn giản để tìm một đường đi Hamilton là sử dụng thuật toán quay lui xét hết mọi cách có thể để xây dựng Một đường đi. Độ phức tạp về thời gian của một thuật toán như vậy ít nhất là  $O(n!)$ , Vì có tất cả  $n!$  cách khác nhau để chọn thứ tự của  $n$  đỉnh.

Một giải pháp tối ưu hơn là dựa vào quy hoạch động (xem Chương 10.5). Ý tưởng là tính giá trị của hàm  $\text{possible}(S, x)$ , Với  $S$  là một tập đỉnh và  $x$  là một đỉnh thuộc  $S$ . Hàm này cho biết liệu có tồn tại một đường đi Hamilton đi qua các đỉnh thuộc tập  $S$  và kết thúc tại đỉnh  $x$  không. Dễ thấy rằng ta có thể thực hiện cách này trong độ phức tạp  $O(2^n n^2)$ .

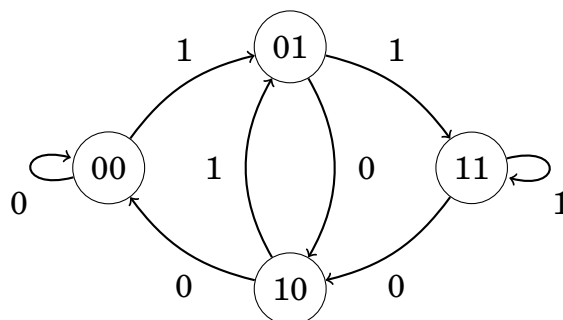
## 19.3 Dãy De Bruijn

**Dãy De Bruijn** là một dãy ký tự bao gồm mọi xâu độ dài  $n$ , mỗi xâu xuất hiện đúng một lần dưới dạng một xâu con, với một bảng chữ cái cố định gồm  $k$  ký tự. Độ dài của một xâu như vậy là  $k^n + n - 1$  ký tự. Ví dụ, với  $n = 3$  và  $k = 2$ , một dãy De Bruijn tương ứng là

0001011100.

Các xâu con của dãy này là mọi tổ hợp của ba phần tử: 000, 001, 010, 011, 100, 101, 110 and 111.

Thực tế là mỗi dãy De Bruijn tương ứng với một đường đi Euler trong đồ thị. Ý tưởng là ta phải tạo ra một đồ thị mà mỗi đỉnh chứa một xâu gồm  $n - 1$  ký tự và mỗi cạnh sẽ thêm một ký tự vào xâu. Đồ thị dưới đây tương ứng với trường hợp được nêu trên:



Mỗi đường đi Euler trên đồ thị này tương ứng với một dãy De Bruijn, dãy bao gồm các ký tự của đỉnh xuất phát và ký tự của các cạnh. Có  $n - 1$  ký tự trên đỉnh xuất phát và có  $k^n$  ký tự trên các cạnh, vì vậy xâu tìm được có tất cả  $k^n + n - 1$  ký tự.

## 19.4 Mã đi tuần

**Mã đi tuần** là một chuỗi các nước đi của quân mã trên một bàn cờ  $n \times n$  theo luật cờ vua, sao cho mỗi ô được thăm đúng một lần. Có thể hiểu đây là "Hành trình của quân mã". Hành trình này được gọi là *đóng* nếu quân mã kết thúc tại chính ô mà nó khởi đầu. Nếu không thì nó được gọi là một hành trình *mở*.

Ví dụ dưới đây là một hành trình mở của quân mã trên bàn cờ  $5 \times 5$ :

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 đồ thị mà mỗi đỉnh đại diện cho một ô trên bảng và hai đỉnh được nối với nhau bằng một cạnh nếu quân mã có thể di chuyển được giữa hai ô theo luật cờ vua.

Một cách tự nhiên để xây dựng một hành trình cho quân mã là sử dụng quay lui. Việc tìm kiếm có thể được thực hiện hiệu quả hơn bằng cách sử dụng *heuristics* thử hướng quân mã đi sao cho tìm thấy được một hành trình hoàn chỉnh nhanh nhất.

## Giải thuật Warnsdorf

**Giải thuật Warnsdorf** là một giải thuật heuristic đơn giản và hiệu quả trong việc tìm kiếm hành trình của quân mã <sup>3</sup>. Nếu tuân theo quy tắc, ta có thể dễ dàng tạo ra một hành trình cho quân mã ngay cả trên một bàn cờ lớn. Ý tưởng là ta luôn di chuyển quân mã sao cho nó đi vào ô có số lượng nước đi hợp lệ vào lượt tiếp theo nhỏ nhất có thể.

Ví dụ, trong trường hợp dưới đây, có năm ô mà quân mã có thể di chuyển đến (các ô  $a \dots e$ ):

1				$a$
		2		
$b$				$e$
	$c$		$d$	

Trong trường hợp này, giải thuật Warnsdorf di chuyển quân mã đến ô  $a$ , vì sau lượt di chuyển này, chỉ có một nước đi duy nhất có thể. Những lựa chọn khác sẽ đưa quân mã đến những ô vuông mà có ba cách di chuyển hợp lệ.

---

<sup>3</sup>Giải thuật heuristic này đã được đề xuất trong cuốn sách của Warnsdorf [69] vào năm 1823. Ngoài ra có một số thuật toán đa thức khác để tìm hành trình của quân mã [52], nhưng chúng phức tạp hơn.

# Chương 20

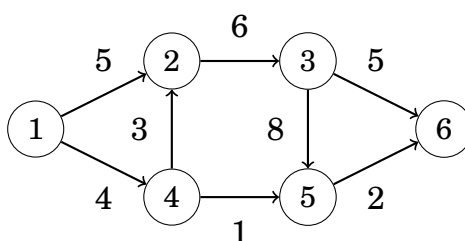
## Luồng và lát cắt

Ở chương này, ta tập trung vào hai bài toán sau:

- **Tìm luồng cực đại:** Lượng luồng cực đại mà ta có thể truyền từ đỉnh này tới đỉnh khác là bao nhiêu?
- **Tìm lát cắt cực tiểu:** Tổng trọng số cạnh nhỏ nhất của một lát cắt - một tập các cạnh phân chia hai đỉnh cho trước trong đồ thị, tức nếu ta cắt bỏ những cạnh thuộc tập này thì hai đỉnh này không còn liên thông nữa.

Cả hai bài toán đều có đầu vào là một đồ thị có hướng, có trọng số và chứa hai đỉnh đặc biệt: *Nguồn* (đỉnh phát) là đỉnh không có cạnh đi vào, và *đích* (đỉnh thu) là đỉnh không có cạnh đi ra.

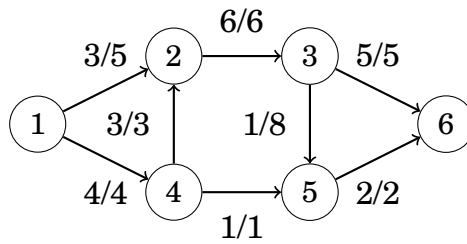
Ví dụ, xét đồ thị dưới đây có nút 1 là đỉnh phát và nút 6 là đỉnh thu:



### Luồng cực đại

Trong bài toán **Luồng cực đại**, nhiệm vụ của ta là gửi càng nhiều luồng càng tốt từ đỉnh phát tới đỉnh thu. Trọng số của mỗi cạnh chính là thông lượng, giới hạn lượng luồng được phép thông qua cạnh đó. Với mỗi nút trung gian, tổng luồng vào phải bằng với tổng luồng ra.

Ví dụ, luồng cực đại của đồ thị trên là 7. Hình ảnh sau minh họa một cách định hướng, phân bổ luồng trong mạng:

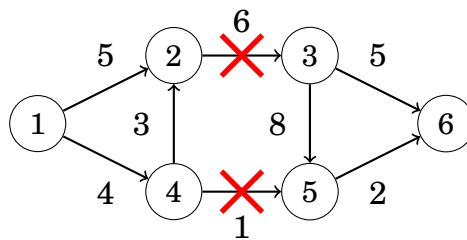


Ký hiệu  $v/k$  biểu diễn một lượng  $v$  đơn vị luồng được truyền qua một cạnh có giới hạn thông qua là  $k$  đơn vị. Tổng lượng luồng là 7, bởi vì đỉnh phát gửi đi  $3 + 4$  đơn vị luồng và đỉnh thu nhận về  $5 + 2$  đơn vị luồng. Dễ thấy rằng lượng luồng này là cực đại, bởi vì tổng giới hạn thông qua của các cạnh dẫn đến đỉnh thu là 7.

## Lát cắt cực tiểu

Trong bài toán **lát cắt hẹp nhất** (tên gọi khác của lát cắt cực tiểu), ta cần phải loại một tập các cạnh ra khỏi đồ thị sao cho không còn đường đi từ đỉnh phát đến đỉnh thu và tổng trọng số của các cạnh bị xóa là nhỏ nhất.

Lát cắt hẹp nhất trong đồ thị ví dụ là 7. Chỉ cần xóa các cạnh  $2 \rightarrow 3$  và  $4 \rightarrow 5$  là đủ:



Sau khi xóa các cạnh, sẽ không còn đường đi nào đi từ đỉnh phát tới đỉnh thu. Kích thước của lát cắt là 7, bởi vì trọng số của các cạnh bị xóa là 6 và 1. Lát cắt này là cực tiểu, bởi vì không tồn tại cách nào để xóa các cạnh khỏi đồ thị sao cho tổng trọng số của chúng nhỏ hơn 7.

Không phải trùng hợp mà trong ví dụ trên, hai giá trị luồng cực đại và lát cắt hẹp nhất bằng nhau. Thực ra, hai giá trị này *luôn luôn* bằng nhau, nên có thể nói các khái niệm này là hai mặt của một đồng xu.

Tiếp đến ta sẽ thảo luận thuật toán Ford-Fulkerson có thể dùng để tìm luồng cực đại và lát cắt hẹp nhất của một đồ thị. Thuật toán sẽ cho ta thấy được *tại sao* chúng bằng nhau.

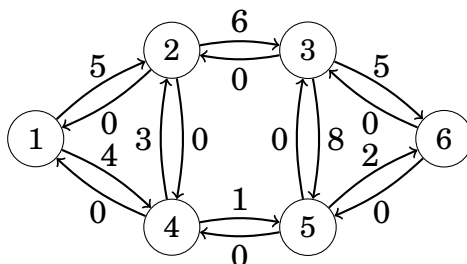
## 20.1 Thuật toán Ford-Fulkerson

**Thuật toán Ford-Fulkerson** [25] tìm luồng cực đại của một đồ thị. Thuật toán bắt đầu với một luồng rỗng, và ở mỗi bước tìm một đường đi từ đỉnh

phát đến đỉnh thu để tăng thêm luồng. Cuối cùng, khi không thể tăng luồng được nữa, ta kết luận đã tìm thấy luồng cực đại.

Trong thuật toán, ta biểu diễn đồ thị theo cách khác: với mỗi cạnh ban đầu, ta thêm vào một cung ngược. Trọng số của mỗi cạnh cho biết lượng luồng mà ta còn có thể truyền thêm qua nó. Khi bắt đầu thuật toán, trọng số của mỗi cạnh gốc (có trong đồ thị ban đầu) bằng thông lượng của nó, và trọng số của các cạnh ngược bằng 0.

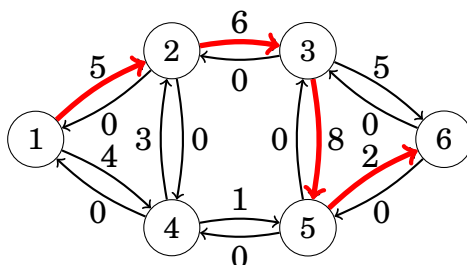
Đồ thị trong ví dụ sẽ được biểu diễn như sau:



## Mô tả thuật toán

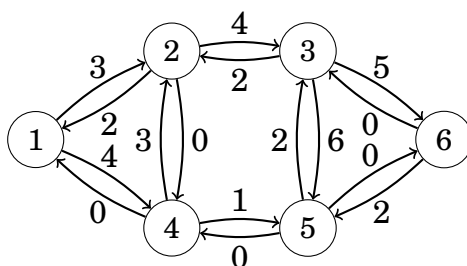
Thuật toán Ford–Fulkerson bao gồm nhiều lượt. Ở mỗi lượt, thuật toán tìm một đường đi từ đỉnh phát tới đỉnh thu sao cho mỗi cạnh trên đường đi đó đều có trọng số dương. Nếu có nhiều hơn một đường đi, ta có thể chọn bất kỳ.

Ví dụ, giả sử ta chọn đường đi sau:



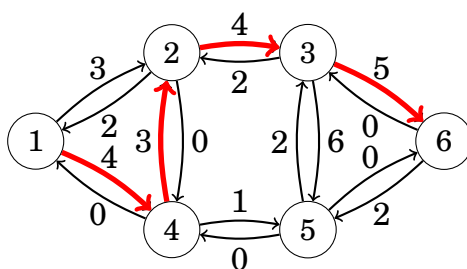
Sau khi chọn, luồng sẽ được tăng lên  $x$  đơn vị với  $x$  là trọng số cạnh nhỏ nhất trên đường đi. Sau đó, trọng số của mỗi cạnh trên đường đi bị giảm đi  $x$  và trọng số của mỗi cạnh ngược được tăng lên  $x$ .

Trong đường đi trên, trọng số của các cạnh lần lượt là 5, 6, 8 và 2. Trọng số nhỏ nhất là 2, nên luồng sẽ tăng lên 2 và đồ thị mới sẽ như sau:



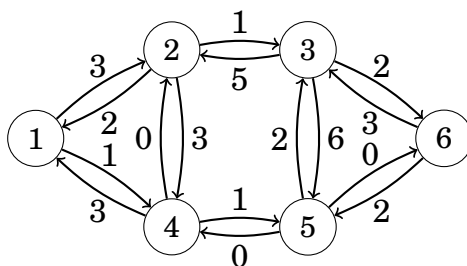
Ý tưởng là việc tăng luồng sẽ giảm đi lượng luồng được phép thông qua những cạnh đó trong tương lai. Mặt khác, ta có thể rút lại bằng cách đi theo các cạnh ngược nếu việc dẫn luồng đi theo tuyến khác sẽ có lợi hơn.

Chừng nào vẫn còn tìm được đường đi từ đỉnh phát tới đỉnh thu thông qua các cạnh trọng số dương, thuật toán sẽ tiếp tục tăng luồng. Trong ví dụ này, đường đi tiếp theo của ta có thể là đường sau:

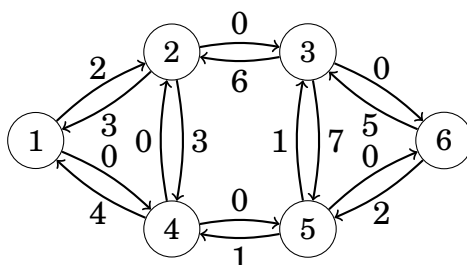


Trọng số nhỏ nhất của cạnh thuộc đường đi là 3, Vì vậy đường đi sẽ tăng luồng lên 3 đơn vị, do đó tổng luồng sẽ là 5.

Đồ thị mới sẽ như sau:



Ta vẫn cần thêm hai lượt nữa để có được luồng cực đại. Ví dụ, ta có thể chọn các đường đi  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$  và  $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ . Cả hai đều tăng luồng thêm 1 đơn vị, ta có được đồ thị cuối cùng như sau:



Không thể tăng luồng thêm được nữa bởi không còn đường đi trọng số dương nào từ đỉnh phát tới đỉnh thu. Do đó, thuật toán kết thúc với luồng cực đại là 7.



## Tìm đường

Thuật toán Ford–Fulkerson không định rõ ta nên chọn đường tăng luồng như thế nào. Dù sao thì, thuật toán sớm muộn gì cũng kết thúc và tìm được chính xác luồng luồng cực đại. Tuy nhiên, độ hiệu quả của thuật toán phụ thuộc vào cách chọn các con đường.

Một cách đơn giản để tìm đường là duyệt theo chiều sâu. Thường thì cách làm này hiệu quả, nhưng trong trường hợp tệ nhất, mỗi đường đi có thể chỉ tăng luồng lên 1 đơn vị khiến thuật toán chạy chậm. May thay, ta có thể tránh điều này bằng cách sử dụng các kĩ thuật sau:

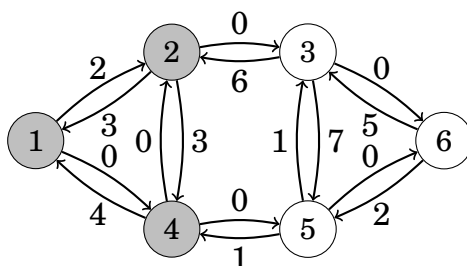
**Thuật toán Edmonds–Karp** [18] chọn những con đường có số cạnh trên đường đi nhỏ nhất có thể. Việc này có thể làm được bằng cách duyệt theo chiều rộng thay vì duyệt theo chiều sâu. Có thể chứng minh rằng thuật toán đảm bảo luồng tăng nhanh. Thuật này có độ phức tạp thời gian là  $O(m^2n)$

**Thuật toán tỉ lệ** [2] sử dụng duyệt theo chiều sâu để tìm những đường đi có trọng số của mỗi cạnh ít nhất bằng với một ngưỡng giá trị nào đó. Ban đầu, chặn dưới này được gán một giá trị đủ lớn, ví dụ như tổng trọng số của tất cả các cạnh trong đồ thị. Mỗi khi không thể tìm thấy đường đi, ta chia đôi ngưỡng giá trị (chặn dưới) này. Độ phức tạp thời gian của thuật toán là  $O(m^2 \log c)$ , với  $c$  là khởi tạo của ngưỡng giá trị.

Trong thực tế, việc thực thi thuật toán tỉ lệ sẽ dễ dàng hơn, vì nó sử dụng duyệt theo chiều sâu để tìm đường đi. Cả hai thuật toán đều đủ hiệu quả cho các dạng bài thường xuất hiện trong các kỳ thi lập trình.

## Lát cắt hẹp nhất

Khi mà thuật toán Ford–Fulkerson tìm xong luồng cực đại, nó cũng xác định được một lát cắt hẹp nhất. Gọi  $A$  là tập hợp các đỉnh có thể đến được từ đỉnh phát bằng các cạnh có trọng số dương. Trong đồ thị ở ví dụ,  $A$  bao gồm các đỉnh 1, 2 và 4:

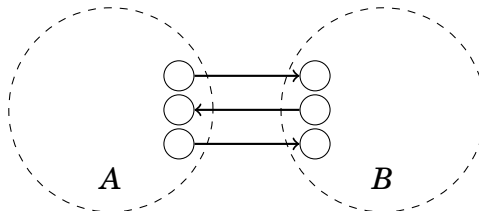


Bây giờ lát cắt hẹp nhất bao gồm các cạnh trong đồ thị ban đầu có điểm đầu là một đỉnh trong  $A$ , điểm cuối là một đỉnh ngoài  $A$ , mà luồng cực đại sử dụng hết thông lượng của cạnh này. Trong đồ thị trên, các cạnh như vậy là  $2 \rightarrow 3$  và  $4 \rightarrow 5$  tương ứng với lát cắt hẹp nhất  $6 + 1 = 7$ .

Vì sao luồng tạo ra bởi thuật toán là cực đại và lát cắt là cực tiểu? Nguyên nhân là do một đồ thị không thể chứa một luồng nào có kích thước lớn hơn

một lát cắt bất kỳ trong đồ thị. Do đó, khi có một luồng và một lát cắt có độ lớn bằng nhau, chúng phải là luồng cực đại và lát cắt cực tiểu.

Xét bất kỳ lát cắt của đồ thị sao cho đỉnh phát thuộc tập  $A$ , đỉnh thu thuộc tập  $B$  và có một số cạnh nối giữa hai tập này:



Kích thước của lát cắt là tổng các cạnh nối từ  $A$  đến  $B$ . Đây là cận trên của luồng trong đồ thị, bởi vì luồng phải truyền từ  $A$  đến  $B$ . Vì vậy, kích thước của luồng cực đại phải nhỏ hơn hoặc bằng kích thước của bất kỳ lát cắt nào trong đồ thị.

Nói cách khác, thuật toán Ford–Fulkerson tạo ra một luồng với kích thước *bằng đúng* với kích thước của một lát cắt trong đồ thị. Vậy luồng này phải là luồng cực đại và lát cắt đó phải là lát cắt hẹp nhất.

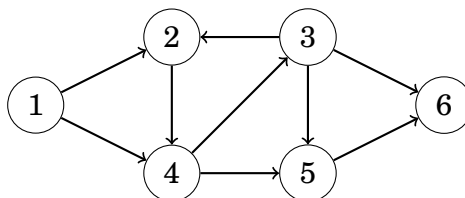
## 20.2 Đường đi phân biệt

Nhiều bài toán đồ thị có thể giải được bằng cách chuyển chúng thành bài toán tìm luồng cực đại. Ví dụ đầu tiên của chúng ta như sau: cho một đồ thị có hướng với đỉnh phát và đỉnh thu, nhiệm vụ của ta là tìm giá trị lớn nhất của số lượng các đường đi phân biệt từ đỉnh phát tới đỉnh thu.

### Đường đi có cạnh phân biệt

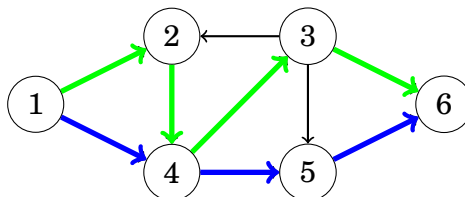
Trước tiên, ta sẽ tập trung vào bài toán đếm số lượng **đường đi có cạnh phân biệt** từ đỉnh phát đến đỉnh thu. Tức là ta sẽ xây dựng một tập các đường đi thỏa mãn mỗi cạnh xuất hiện trong nhiều nhất một đường.

Ví dụ, xét đồ thị sau:



Trong đồ thị này, số lượng đường đi phân biệt nhiều nhất là 2.

Ta có thể chọn các con đường  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$  và  $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$  như sau:

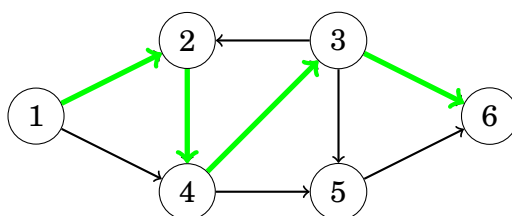


Hóa ra, số đường đi có cạnh phân biệt nhiều nhất lại bằng với luồng cực đại của đồ thị, với giả sử rằng thông lượng của mỗi cạnh là một. Sau khi dựng xong luồng cực đại, có thể tìm được các đường đi phân biệt bằng cách sử dụng chiến thuật tham lam, đi lần theo các con đường từ đỉnh phát đến đỉnh thu.

## Đường đi có đỉnh phân biệt

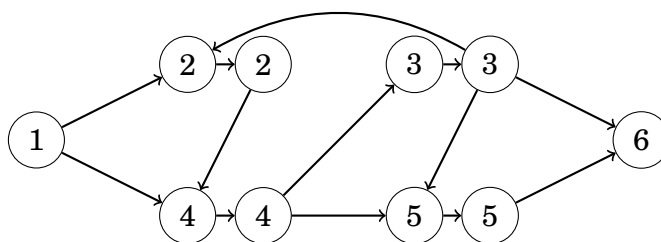
Xét bài toán khác: Tìm số lượng **đường đi có đỉnh phân biệt** nhiều nhất từ đỉnh phát đến đỉnh thu. Trong bài toán này, mỗi đỉnh, ngoại trừ đỉnh phát và đỉnh thu, có thể xuất hiện trong nhiều nhất là một đường đi. Số lượng đường đi có đỉnh phân biệt tối đa có thể sẽ nhỏ hơn số lượng đường đi có cạnh phân biệt.

Ví dụ, với đồ thị trước đó, số lượng đường đi có đỉnh phân biệt nhiều nhất là 1:

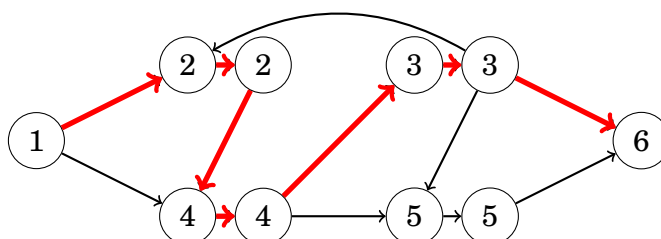


Ta cũng có thể chuyển bài toán này thành bài toán luồng cực đại. Vì các đỉnh chỉ có thể xuất hiện trong nhiều nhất một đường đi, ta phải giới hạn lượng luồng đi qua các đỉnh. Một phương pháp chuẩn để giải quyết vấn đề này là chia mỗi đỉnh thành hai với đỉnh đầu tiên (gọi là đỉnh vào) sẽ lấy mọi cạnh đi vào đỉnh gốc, đỉnh thứ hai (tạm gọi đỉnh ra) sẽ có tất cả cạnh đi ra khỏi đỉnh gốc, và ta sẽ nối một cạnh từ đỉnh vào tới đỉnh ra.

Trong ví dụ của chúng ta, đồ thị sẽ trở thành:



Luồng cực đại của đồ thị:



Vì vậy, số lượng đường đi có đỉnh phân biệt tối đa từ đỉnh phát đến đỉnh thu là 1.

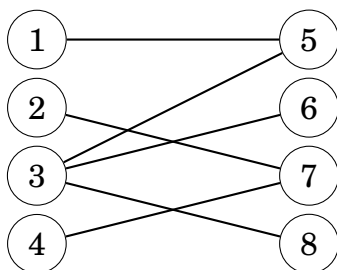
## 20.3 Cặp ghép cực đại

Bài toán **Cặp ghép cực đại** yêu cầu tìm một tập lớn nhất các cặp đỉnh trong đồ thị vô hướng sao cho mỗi cặp đều nối với nhau bởi một cạnh và mỗi đỉnh thuộc vào nhiều nhất một cặp.

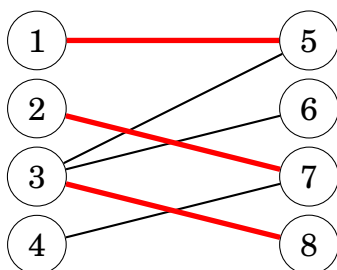
Có thuật toán đa thức tìm được cặp ghép cực đại trong đồ thị tổng quát [17], Nhưng các thuật toán như thế rất phức tạp và hiếm thấy trong các cuộc thi lập trình. Tuy nhiên, trong đồ thị hai phía, bài toán cặp ghép cực đại sẽ dễ dàng để giải hơn, vì ta có thể quy nó thành bài toán luồng cực đại.

### Tìm cặp ghép cực đại

Các đỉnh của một đồ thị hai phía luôn luôn chia được thành hai nhóm sao cho mọi cạnh của đồ thị đều nối từ nhóm bên trái đến nhóm bên phải. Ví dụ, trong đồ thị hai phía sau, có hai nhóm là  $\{1, 2, 3, 4\}$  và  $\{5, 6, 7, 8\}$ .

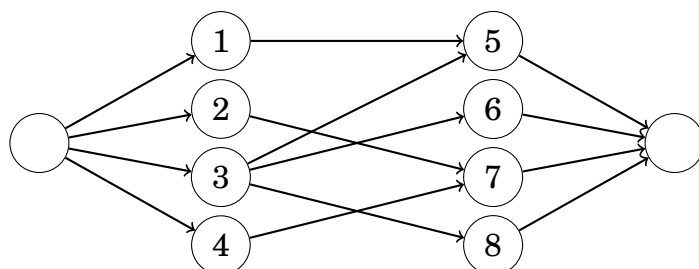


Kích thước cặp ghép cực đại trong đồ thị này là 3:

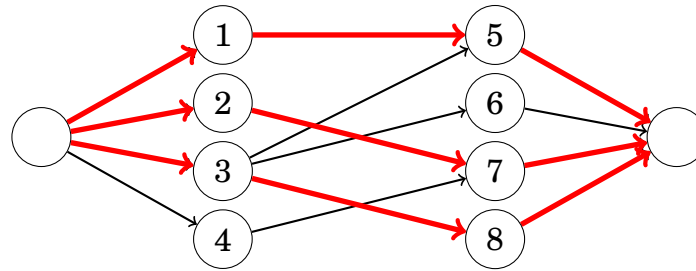


Ta có thể chuyển bài toán cặp ghép cực đại trên đồ thị hai phía thành bài toán luồng cực đại bằng cách thêm hai đỉnh mới vào đồ thị: một đỉnh phát và một đỉnh thu. Ta cũng thêm các cạnh từ đỉnh phát đến mỗi đỉnh bên trái và cạnh từ mỗi đỉnh bên phải đến đỉnh thu. Lúc này, kích thước luồng cực đại của đồ thị sau khi thêm bằng đúng kích thước cặp ghép cực đại trên đồ thị ban đầu.

Ví dụ, ta biến đổi đồ thị ở trên như sau:



Luồng cực đại của đồ thị này như sau:

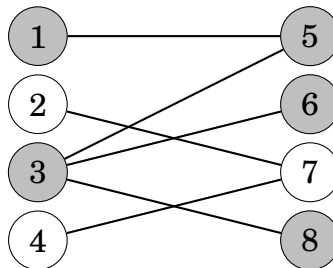


## Định lý Hall

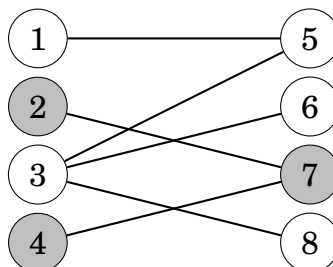
**Định lý Hall** có thể được dùng để xác định xem một đồ thị hai phía có tồn tại một cặp ghép chứa toàn bộ đỉnh bên trái hoặc bên phải hay không. Nếu số lượng đỉnh ở bên trái và bên phải bằng nhau, định lý Hall cho ta biết liệu có dựng được một **cặp ghép hoàn hảo** chứa toàn bộ đỉnh của đồ thị.

Giả sử ta muốn tìm một cặp ghép chứa tất cả các đỉnh trái. Đặt  $X$  là tập hợp các đỉnh bên trái bất kỳ và  $f(X)$  là tập các hàng xóm của chúng. Dựa trên định lý Hall, một cặp ghép chứa tất cả các đỉnh bên trái tồn tại chỉ khi với mỗi  $X$ , điều kiện  $|X| \leq |f(X)|$  thỏa mãn.

Ta hãy nghiên cứu định lý Hall trong đồ thị ví dụ. Đầu tiên, với  $X = \{1, 3\}$  thì  $f(X) = \{5, 6, 8\}$ :



Điều kiện của định lý Hall thỏa mãn, bởi vì  $|X| = 2$  and  $|f(X)| = 3$ . Tiếp theo, với  $X = \{2, 4\}$  ta có  $f(X) = \{7\}$



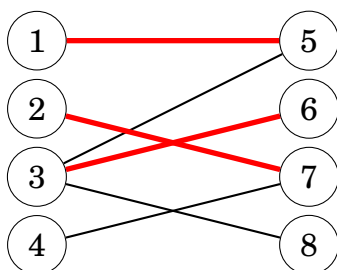
Trong trường hợp này,  $|X| = 2$  và  $|f(X)| = 1$ , nên điều kiện của định lý Hall không được đảm bảo. Tức là không thể tạo một cặp ghép hoàn hảo cho đồ thị. Không ngạc nhiên lắm, vì ta đã biết trước rằng cặp ghép cực đại của đồ thị là 3, không phải 4.

Nếu điều kiện của định lý Hall không được thỏa mãn, tập  $X$  giúp ta giải thích *vì sao* không thể tạo ra một cặp ghép hoàn hảo. Vì  $X$  chứa nhiều đỉnh hơn  $f(X)$ , sẽ không đủ cặp cho toàn bộ đỉnh thuộc  $X$ . Ví dụ, với đồ thị trên, cả đỉnh 2 và 4 đều phải được nối với đỉnh 7 nhưng điều này là không thể.

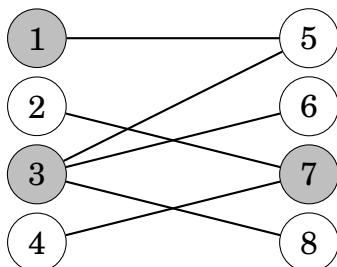
## Định lý Kőnig

Tập phủ đỉnh cực tiểu của một đồ thị là tập các đỉnh bé nhất sao cho mỗi cạnh của đồ thị có ít nhất một đầu mút thuộc vào tập này. Trong đồ thị tổng quát, tìm tập phủ đỉnh cực tiểu là một bài toán NP-khó. Tuy nhiên, nếu đồ thị là đồ thị hai phía, **định lý Kőnig** cho ta biết rằng kích thước của tập phủ đỉnh cực tiểu và kích thước của luồng cực đại luôn bằng nhau. Vì vậy, ta có thể tính được kích thước của tập phủ đỉnh cực tiểu bằng thuật toán luồng cực đại.

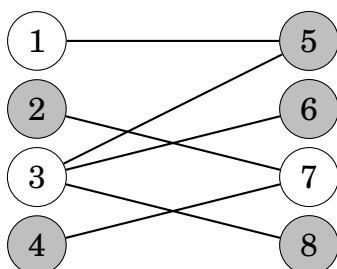
Xét đồ thị sau với cặp ghép cực đại là 3:



Bây giờ định lý Kőnig cho ta biết kích thước của tập phủ đỉnh cực tiểu cũng bằng 3. Một tập như vậy có thể dựng được như sau:



Các đỉnh *không* thuộc về tập phủ đỉnh cực tiểu tạo thành một **tập độc lập cực đại**. Đây là tập lớn nhất các đỉnh sao cho không tồn tại hai đỉnh nào thuộc tập hợp có cạnh nối với nhau. Tìm tập độc lập cực đại trong đồ thị tổng quát lần nữa cũng là một bài toán NP-khó, nhưng trong đồ thị hai phía ta có thể dùng định lý Kőnig để giải bài toán một cách hiệu quả. Trong đồ thị ví dụ, tập độc lập cực đại của đồ thị như sau:

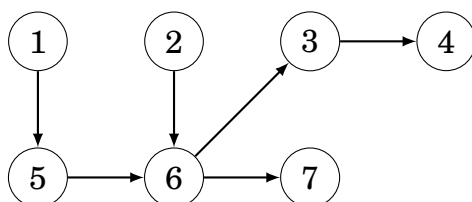


## 20.4 Tập đường bao phủ

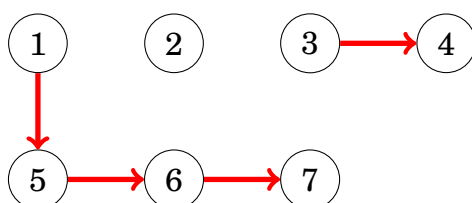
**Tập đường bao phủ** là một tập hợp chứa các đường đi trong một đồ thị sao cho mỗi đỉnh của đồ thị nằm trên đúng một đường đi. Trong đồ thị có hướng, phi chu trình, ta có thể chuyển bài toán tìm tập đường bao phủ cực tiểu thành bài toán luồng cực đại trong một đồ thị khác.

### Tập đường phân biệt theo đỉnh

Trong **tập đường bao phủ với các đỉnh phân biệt**, mỗi đỉnh thuộc chính xác một đường đi. Ví dụ, xét đồ thị sau:



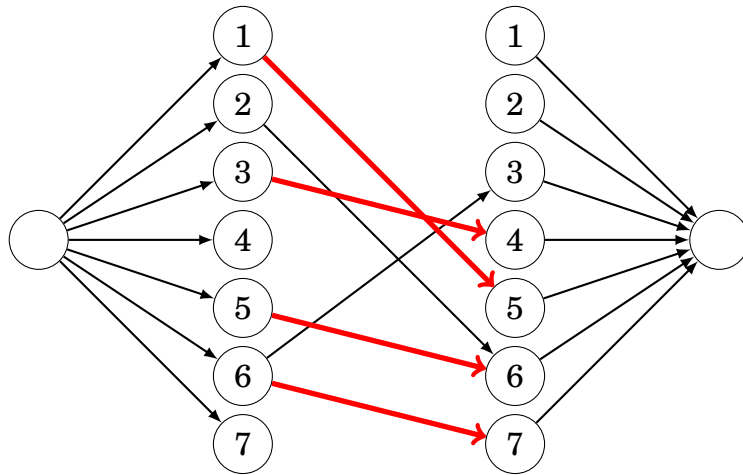
Tập đường bao phủ đỉnh phân biệt của đồ thị này bao gồm ba con đường. Ví dụ, ta có thể chọn các con đường sau:



Chú ý rằng ở trên có một con đường chỉ chứa đỉnh 2, nên trong thực tế có khả năng một đường đi nào đó không có cạnh.

Ta có thể tìm tập đường bao phủ với các đỉnh phân biệt bằng cách xây dựng một *đồ thị ghép nối* trong đó mỗi đỉnh thuộc đồ thị gốc được biểu diễn bởi hai đỉnh thuộc đồ thị này: một đỉnh trái và một đỉnh phải. Có một cạnh nối từ đỉnh  $u$  trái đến đỉnh  $v$  phải nếu có cạnh  $(u, v)$  trong đồ thị gốc. Ngoài ra, đồ thị này còn chứa một đỉnh phát và một đỉnh thu, và có các cạnh từ đỉnh phát đến tất cả các đỉnh trái và cạnh từ các đỉnh phải đến đỉnh thu.

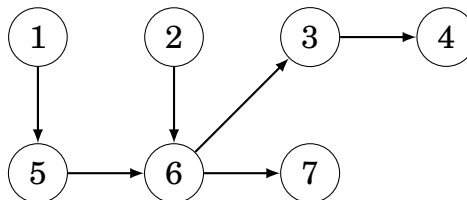
Cặp ghép cực đại của đồ thị tạo thành tương ứng với tập đường bao phủ đỉnh phân biệt trong đồ thị gốc. Ví dụ, từ đồ thị trên, ta tạo được đồ thị ghép nối sau, chứa cặp ghép cực đại kích thước 4:



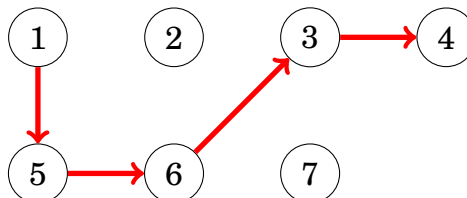
Mỗi cạnh trong cặp ghép cực đại của đồ thị ghép nổi tương ứng với một cạnh trong tập đường bao phủ đỉnh phân biệt cực tiểu của đồ thị gốc. Như vậy, kích thước của tập đường bao phủ với các đỉnh phân biệt cực tiểu là  $n - c$ , với  $n$  là số lượng đỉnh trong đồ thị gốc và  $c$  là kích thước của cặp ghép cực đại.

### Tập đường bao phủ tổng quát

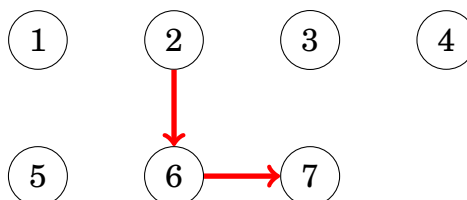
**Tập đường bao phủ tổng quát** là một tập đường bao phủ trong đó mỗi đỉnh có thể thuộc vào nhiều hơn một đường đi. Tập đường bao phủ tổng quát cực tiểu có thể nhỏ hơn tập đường bao phủ đỉnh phân biệt nhỏ nhất, bởi vì một đỉnh có thể được dùng nhiều lần trong các đường đi. Lại xét đồ thị sau:



Tập đường bao phủ cực tiểu của đồ thị bao gồm hai đường đi. Ví dụ, đường đi đầu tiên có thể là:



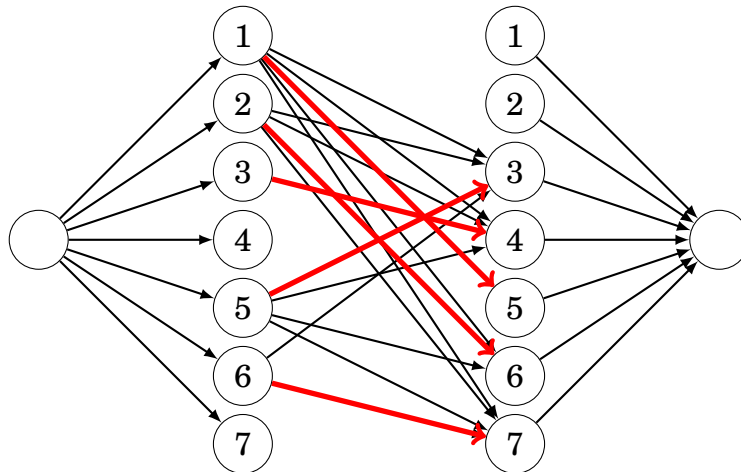
Và đường đi thứ hai có thể là:





Tập đường bao phủ tổng quát có thể tìm được gần giống với tìm tập đường bao phủ đỉnh phân biệt. Ta chỉ việc thêm một vài cạnh mới vào đồ thị ghép nối sao cho luôn có một cạnh  $a \rightarrow b$  nếu có đường đi từ  $a$  đến  $b$  trong đồ thị gốc (có thể thông qua nhiều cạnh).

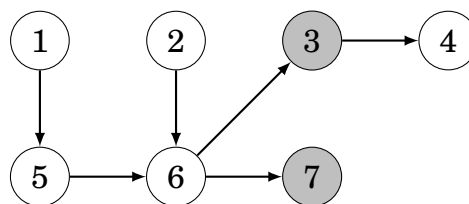
Đồ thị ghép nối cho trường hợp trên như sau:



## Định lý Dilworth

**Tập hợp không chuỗi** (tạm dịch từ antichain) là một tập các đỉnh trong đồ thị sao cho không tồn tại đường đi nào từ một đỉnh tới bất kỳ đỉnh nào khác bằng các cạnh thuộc đồ thị. **Định lý Dilworth** phát biểu rằng trong một đồ thị có hướng phi chu trình, kích thước của tập đường bao phủ tổng quát cực tiểu bằng với kích thước của tập hợp không chuỗi cực đại.

Ví dụ, đỉnh 3 và 7 tạo ra một tập hợp không chuỗi trong đồ thị như sau:



Đây là tập hợp không chuỗi cực đại, bởi ta không thể xây dựng bất kỳ tập nào khác chứa ba đỉnh. Trước đó ta đã biết được tập đường bao phủ tổng quát của đồ thị bao gồm hai đường đi.



# **Phần III**

## **Chủ đề nâng cao**



# Chương 21

## Số học

**Số học** là một nhánh của toán học chuyên nghiên cứu về số nguyên. Số học là một lĩnh vực thú vị, bởi vì có nhiều câu hỏi rất khó liên quan đến số nguyên dù thoát nhìn thì thấy đơn giản.

Ví dụ, xét phương trình sau:

$$x^3 + y^3 + z^3 = 33$$

Dễ dàng tìm được ba số thực  $x$ ,  $y$  và  $z$  thỏa mãn phương trình. Ví dụ, ta có thể chọn

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

Tuy nhiên, đây là một bài toán mở trong Số học - bài toán kiểm tra xem có tồn tại ba số *nguyên*  $x$ ,  $y$  và  $z$  thỏa mãn phương trình hay không [6].

Trong chương này, chúng ta sẽ tập trung vào các khái niệm và thuật toán cơ bản trong Số học. Xuyên suốt chương này, ta sẽ giả sử rằng tất cả các số đều là số nguyên, nếu không nêu gì khác.

### 21.1 Số nguyên tố và Ước số

Số  $a$  được gọi là **ước** của số  $b$  (thuật ngữ: **factor**, **divisor**) nếu  $b$  chia hết cho  $a$ . Nếu  $a$  là ước của  $b$ , ta viết  $a \mid b$ , ngược lại ta viết  $a \nmid b$ . Ví dụ, các ước của 24 là 1, 2, 3, 4, 6, 8, 12 và 24.

Số  $n > 1$  được gọi là **số nguyên tố** nếu nó chỉ có hai ước dương là 1 và  $n$ . Ví dụ, 7, 19 và 41 là các số nguyên tố, nhưng 35 thì không, vì  $5 \cdot 7 = 35$ . Với mọi số  $n > 1$ , có duy nhất một cách **phân tích thừa số nguyên tố**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

trong đó  $p_1, p_2, \dots, p_k$  là các số nguyên tố khác nhau và  $\alpha_1, \alpha_2, \dots, \alpha_k$  là các số nguyên dương. Ví dụ, dạng phân tích thừa số nguyên tố của 84 là

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

**Số lượng ước số của  $n$  bằng**

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

vì với mỗi số nguyên tố  $p_i$ , có  $\alpha_i + 1$  cách chọn số lần xuất hiện của nó trong ước số. Ví dụ, số lượng 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  $n$  bằng**

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

trong đó phần sau của phương trình biến đổi dựa trên công thức của cấp số nhân. Ví dụ, tổng 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  $n$  bằng**

$$\mu(n) = n^{\tau(n)/2}.$$

vì ta có thể tạo nên  $\tau(n)/2$  cặp ước số có tích bằng  $n$ . Ví dụ, các ước số của 84 tạo thành các cặp:  $1 \cdot 84$ ,  $2 \cdot 42$ ,  $3 \cdot 28$ , v.v... và tích các ước là  $\mu(84) = 84^6 = 351298031616$ .

Số  $n$  được gọi là **số hoàn hảo** nếu  $n = \sigma(n) - n$ , tức là,  $n$  bằng tổng các ước số của nó trong đoạn 1 và  $n - 1$ . Ví dụ, 28 là số hoàn hảo, vì  $28 = 1 + 2 + 4 + 7 + 14$ .

## Số lượng số nguyên tố

Dễ dàng chứng minh rằng có vô hạn số nguyên tố. Nếu số lượng số nguyên tố là hữu hạn, ta có thể tạo ra tập  $P = \{p_1, p_2, \dots, p_n\}$  chứa tất cả các số nguyên tố. Ví dụ,  $p_1 = 2$ ,  $p_2 = 3$ ,  $p_3 = 5$ , v.v... Tuy nhiên, từ  $P$ , ta có thể tạo ra số nguyên tố mới

$$p_1 p_2 \cdots p_n + 1,$$

lớn hơn tất cả các phần tử trong  $P$ . Điều này là mâu thuẫn, vì vậy số lượng số nguyên tố phải là vô hạn.

## Mật độ các số nguyên tố

Mật độ của số nguyên tố là mức độ thường xuyên xuất hiện của chúng trong các số nguyên. Đặt  $\pi(n)$  là số lượng số nguyên tố từ 1 tới  $n$ . Ví dụ,  $\pi(10) = 4$ , vì có 4 số nguyên tố từ 1 tới 10: 2, 3, 5 và 7.

Có thể chứng minh rằng

$$\pi(n) \approx \frac{n}{\ln n},$$

tức là các số nguyên tố xuất hiện khá thường xuyên. Ví dụ, số lượng số nguyên tố từ 1 tới  $10^6$  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 cho rằng các giả thuyết là đúng, nhưng chưa ai có thể chứng minh. Ví dụ, các giả thuyết sau khá nổi tiếng:

- **Giả thuyết Goldbach:** Mọi số chẵn  $n > 2$  đều có thể biểu diễn dưới dạng  $n = a + b$ , trong đó  $a$  và  $b$  là các số nguyên tố.
- **Giả thuyết số nguyên tố sinh đôi:** Có vô hạn cặp số có dạng  $\{p, p+2\}$ , trong đó cả  $p$  và  $p+2$  đều là số nguyên tố.
- **Giả thuyết Legendre:** Luôn có một số nguyên tố nằm giữa các số  $n^2$  và  $(n+1)^2$ , trong đó  $n$  là số nguyên dương bất kỳ.

## Các thuật toán cơ bản

Nếu  $n$  không phải là số nguyên tố, nó có thể được biểu diễn dưới dạng tích  $a \cdot b$ , trong đó  $a \leq \sqrt{n}$  hoặc  $b \leq \sqrt{n}$ , nên chắc chắn nó có một ước số nằm trong đoạn từ 2 tới  $\lfloor \sqrt{n} \rfloor$ . Từ sự quan sát này, ta có thể kiểm tra một số có phải là số nguyên tố không và phân tích nó ra thừa số nguyên tố trong độ phức tạp  $O(\sqrt{n})$ .

Hàm prime sau đây kiểm tra xem số  $n$  có phải là số nguyên tố không. Hàm này thử chia  $n$  cho tất cả các số từ 2 tới  $\lfloor \sqrt{n} \rfloor$ , và nếu  $n$  không chia hết cho số nào trong đó, 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 thừa số nguyên tố của  $n$ . Hàm này lấy  $n$  chia cho các ước số nguyên tố của nó, rồi thêm chúng vào vector. Quá trình này kết thúc khi số  $n$  còn lại không có ước số nào nằm trong đoạn từ 2 tới  $\lfloor \sqrt{n} \rfloor$ . Nếu  $n > 1$ , nó là số nguyên tố và là thừa số nguyên tố 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 có số lần xuất hiện bằng với số lần lấy  $n$  chia cho thừa số đó. Ví dụ,  $24 = 2^3 \cdot 3$ , nên kết quả của hàm là  $[2, 2, 2, 3]$ .

## Sàng Eratosthenes

**Sàng Eratosthenes** là một thuật toán tiền xử lý xây dựng một mảng mà từ đó ta có thể kiểm tra nhanh một số bất kỳ trong đoạn từ 2 tới  $n$  có phải là số nguyên tố không, và nếu không phải thì tìm được một thừa số nguyên tố của nó.

Thuật toán này xây dựng một mảng sieve, lưu vào tại các vị trí từ  $2, 3, \dots, n$ . 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 ước số nguyên tố của nó là  $\text{sieve}[k]$ .

Thuật toán lặp qua lần lượt từng số  $2 \dots n$ . Mỗi khi tìm được một số nguyên tố  $x$  mới, thuật toán ghi nhận rằng các bội số của  $x$  ( $2x, 3x, 4x, \dots$ ) không phải là số nguyên tố, bởi vì chúng chia hết cho  $x$ .

Ví dụ, nếu  $n = 20$ , mảng trông như sau:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Đoạn mã sau cài đặt sàng Eratosthenes. Trong đoạn mã này, ta giả định mỗi phần tử của mảng sieve trước đó đều bằng 0.

```

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 nằm trong của thuật toán chạy  $n/x$  lần với mỗi giá trị của  $x$ . Do đó, cận trên cho thời gian chạy của thuật toán là tổng hòa hợp (harmonic sum)

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

Thực tế, thuật toán hiệu quả hơn thế, bởi vì vòng lặp trong chỉ được thực hiện nếu số  $x$  là số nguyên tố. Có thể chứng minh rằng thời gian chạy của thuật toán chỉ là  $O(n \log \log n)$ , một độ phức tạp gần với  $O(n)$ .



## Thuật toán Euclid

Ước chung lớn nhất của hai số  $a$  và  $b$ ,  $\gcd(a, b)$ , là số lớn nhất mà cả  $a$  và  $b$  đều chia hết cho nó, và bội chung nhỏ nhất của  $a$  và  $b$ ,  $\text{lcm}(a, b)$ , là số nhỏ nhất mà chia hết cho cả  $a$  và  $b$ . Ví dụ,  $\gcd(24, 36) = 12$  và  $\text{lcm}(24, 36) = 72$ .

Ước chung lớn nhất và bội chung nhỏ nhất liên hệ với nhau theo công thức sau:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

**Thuật toán Euclid**<sup>1</sup> là một cách làm hiệu quả để tìm ước chung lớn nhất của hai số. Thuật toán dựa trên công thức sau:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

Ví dụ,

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Thuật toán có thể được cài đặt như sau:

```
int gcd(int a, int b) {  
    if (b == 0) return a;  
    return gcd(b, a%b);  
}
```

Có thể chứng minh rằng thuật toán Euclid có độ phức tạp  $O(\log n)$ , với  $n = \min(a, b)$ . Trường hợp xấu nhất của thuật toán là khi  $a$  và  $b$  là hai số Fibonacci liên tiếp. Ví dụ,

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

## Phi hàm Euler

Số  $a$  và  $b$  **nguyên tố cùng nhau** nếu  $\gcd(a, b) = 1$ . **Hàm phi Euler**  $\varphi(n)$  trả về số lượng 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 đều nguyên tố cùng nhau với 12.

Giá trị của  $\varphi(n)$  có thể được tính từ dạng phân tích 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.

---

<sup>1</sup>Euclid là một nhà toán học Hy Lạp sống vào khoảng năm 300 TCN. Đây có lẽ là thuật toán đầu tiên được biết đến trong lịch sử.

## 21.2 Đồng dư thức

Trong **đồng dư thức**, tập hợp các số được giới hạn, 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 bởi số  $x \bmod m$ : phần dư của phép chia  $x$  cho  $m$ . Ví dụ, nếu  $m = 17$ , thì 75 được biểu diễn bởi  $75 \bmod 17 = 7$ .

Thông thường ta có thể chia lấy phần dư trước khi thực hiện các phép tính. Cụ thể, các công thức sau đây đều 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 đồng dư

Thường ta sẽ cần tính nhanh giá trị  $x^n \bmod m$ . Điều này có thể thực hiện trong độ phức tạp  $O(\log n)$  bằng cách sử dụng công thức truy hồi sau:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

Cần đặc biệt lưu tâm trong trường hợp  $n$  chẵn, giá trị  $x^{n/2}$  chỉ được tính một lần. Điều này đảm bảo rằng độ phức tạp của thuật toán là  $O(\log n)$ , vì  $n$  luôn bị chia đôi khi nó là số 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** phát biểu rằng

$$x^{m-1} \bmod m = 1$$

khi  $m$  là số nguyên tố và  $x$  và  $m$  nguyên tố cùng nhau. Từ đây cũng suy ra

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

Tổng quát hơn, **định lý Euler** phát biểu rằng

$$x^{\varphi(m)} \bmod m = 1$$

khi  $x$  và  $m$  nguyên tố cùng nhau. Định lý Fermat được suy ra từ định lý Euler, vì nếu  $m$  là số nguyên tố, thì  $\varphi(m) = m - 1$ .

## Nghịch đảo đồng dư

Nghịch đảo của  $x$  theo modulo  $m$  là một số  $x^{-1}$  sao cho

$$xx^{-1} \bmod m = 1.$$

Ví dụ, nếu  $x = 6$  và  $m = 17$ , thì  $x^{-1} = 3$ , vì  $6 \cdot 3 \bmod 17 = 1$ .

Dùng nghịch đảo đồng dư, ta có thể chia các số theo modulo  $m$ , vì chia cho  $x$  tương đương với nhân cho  $x^{-1}$ . Ví dụ, để tính giá trị của  $36/6 \bmod 17$ , ta có thể dùng công thức  $2 \cdot 3 \bmod 17$ , vì  $36 \bmod 17 = 2$  và  $6^{-1} \bmod 17 = 3$ .

Tuy nhiên, nghịch đảo đồng dư không phải lúc nào cũng tồn tại. Ví dụ, nếu  $x = 2$  và  $m = 4$ , thì phương trình

$$xx^{-1} \bmod m = 1$$

không có nghiệm, vì mọi bội của 2 đều chẵn và phần dư không bao giờ là 1 khi  $m = 4$ . Giá trị  $x^{-1} \bmod m$  luôn tính được khi  $x$  và  $m$  nguyên tố cùng nhau.

Nếu tồn tại nghịch đảo đồng dư, nó có thể được tính bằng công thức

$$x^{-1} = x^{\varphi(m)-1}.$$

Nếu  $m$  là số nguyên tố, thì công thức trở thành

$$x^{-1} = x^{m-2}.$$

Ví dụ,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Công thức này cho phép ta tính nghịch đảo đồng dư một cách hiệu quả bằng thuật toán lũy thừa đồng dư. Công thức có thể được suy ra từ định lý Euler. Trước hết, nghịch đảo đồng dư 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 thế các số  $x^{-1}$  và  $x^{\varphi(m)-1}$  bằng nhau.

## Số học của máy tính

Trong lập trình, số nguyên không dấu được biểu diễn dưới dạng số dư khi chia cho  $2^k$ , trong đó  $k$  là số lượng bit của kiểu dữ liệu. Một hệ quả thường thấy là số sẽ "cuộn vòng" nếu nó trở nên quá lớn.

Ví dụ, trong C++, số kiểu `unsigned int` được biểu diễn theo modulo  $2^{32}$ . Đoạn mã sau khai báo một biến kiểu `unsigned int` có giá trị là 123456789. Sau đó, giá trị của biến sẽ được nhân với chính nó, 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

**Phương trình Diophantine** là 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 số nguyên. Ví dụ, một nghiệm của phương trình  $5x + 2y = 11$  là  $x = 3$  và  $y = -2$ .

Ta có thể giải phương trình Diophantine một cách hiệu quả bằng cách sử dụng thuật toán Euclid. Hóa ra rằng, ta có thể mở rộng thuật toán Euclid để nó tìm được các số  $x$  và  $y$  thỏa mãn phương trình sau:

$$ax + by = \gcd(a, b).$$

Một phương trình Diophantine có thể giải được nếu  $c$  chia hết cho  $\gcd(a, b)$ , ngược lại thì không.

Ví dụ, hãy tìm các số  $x$  và  $y$  thỏa mãn phương trình sau:

$$39x + 15y = 12$$

Phương trình này có nghiệm vì  $\gcd(39, 15) = 3$  và  $3 \mid 12$ . Khi thuật toán Euclid tính ước chung lớn nhất của 39 và 15, nó sẽ tạo ra chuỗi các lời gọi hàm sau:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

Điều này tương ứng với các phương trình sau:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Từ những phương trình này, ta có thể suy ra

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

nhân cả hai vế với 4, ta được

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

vì thế, một nghiệm của phương trình là  $x = 8$  và  $y = -20$ .

Nghiệm của phương trình Diophantine không độc nhất, vì ta có thể tạo ra vô số nghiệm nếu ta biết một nghiệm. Nếu 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 Quốc

**Định lý số dư Trung Quốc** giải một hệ 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  $m_1, m_2, \dots, m_n$  đôi một nguyên tố cùng nhau.

Gọi  $x_m^{-1}$  là nghịch đảo của  $x$  theo modulo  $m$ , và

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Ta sử dụng các ký hiệu này để biểu diễn một nghiệm của hệ phương trình:

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

Trong nghiệm này, với mỗi  $k = 1, 2, \dots, n$ ,

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

vì

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Vì tất cả các số khác trong tổng đều chia hết cho  $m_k$ , nên chúng không ảnh hưởng đến phần dư, và  $x \bmod m_k = a_k$ .

Ví dụ, một nghiệm của hệ

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

là

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Một khi ta đã tìm được nghiệm  $x$ , ta có thể tạo ra vô số nghiệm khác, vì tất cả các số có dạng

$$x + m_1 m_2 \cdots m_n$$

đều là nghiệm.

## 21.4 Kết quả khác

### Định lý Lagrange

**Định lý Lagrange** phát biểu rằng mọi số nguyên dương đều có thể biểu diễn dưới dạng tổng của bốn số bình phương, tức là  $a^2 + b^2 + c^2 + d^2$ . Ví dụ, số 123 có thể biểu diễn dưới dạng tổng  $8^2 + 5^2 + 5^2 + 3^2$ .

### Định lý Zeckendorf

**Định lý Zeckendorf** phát biểu rằng mọi số nguyên dương có một biểu diễn duy nhất dưới dạng tổng các số Fibonacci sao cho không có hai số nào bằng nhau và không có hai số Fibonacci liên tiếp. Ví dụ, số 74 có thể biểu diễn dưới dạng tổng  $55 + 13 + 5 + 1$ .

### Bộ ba Pythagore

**Bộ ba Pythagore** là bộ ba số nguyên dương  $(a, b, c)$  thỏa mãn đẳng thức Pythagore  $a^2 + b^2 = c^2$ , tức là có một tam giác vuông có độ dài các cạnh lần lượt là  $a$ ,  $b$  và  $c$ . Ví dụ,  $(3, 4, 5)$  là một bộ ba Pythagore.

Nếu  $(a, b, c)$  là một bộ ba Pythagore, tất cả các bộ ba có dạng  $(ka, kb, kc)$  cũng là bộ ba Pythagore với  $k > 1$ . Một bộ ba Pythagore được gọi là *nguyên thủy* nếu  $a$ ,  $b$  và  $c$  nguyên tố cùng nhau, và tất cả các bộ ba Pythagore đều có thể tạo ra được từ các bộ ba nguyên thủy bằng cách nhân với một hằng số  $k$ .

**Công thức Euclid** có thể được sử dụng để tạo ra tất cả các bộ ba Pythagore nguyên thủy. Mỗi bộ ba như vậy có dạng

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

trong đó  $0 < m < n$ ,  $n$  và  $m$  nguyên tố cùng nhau và ít nhất một trong hai số  $n$  và  $m$  là số chẵn. Ví dụ, khi  $m = 1$  và  $n = 2$ , công thức tạo ra bộ ba Pythagore nhỏ nhất

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

## Định lý Wilson

**Định lý Wilson** phát biểu rằng một số  $n$  là số nguyên tố khi và chỉ khi

$$(n - 1)! \bmod n = n - 1.$$

Ví dụ, số 11 là số nguyên tố, vì

$$10! \bmod 11 = 10,$$

và số 12 không phải là số nguyên tố, vì

$$11! \bmod 12 = 0 \neq 11.$$

Vì vậy, định lý Wilson có thể dùng để kiểm tra một số có phải là số nguyên tố hay không. Tuy nhiên, trong thực tế, định lý này không thể áp dụng cho các giá trị  $n$  lớn, vì tính toán  $(n - 1)!$  với  $n$  lớn là một việc khó khăn.





# Chương 22

## Tổ hợp

**Tổ hợp** nghiên cứu những phương pháp đếm số cấu hình của các đối tượng. Thông thường, mục tiêu chính là tìm cách đếm số cấu hình một cách hiệu quả mà không phải sinh ra từng cấu hình một.

Ví dụ, xét bài toán đếm số cách biểu diễn số nguyên  $n$  dưới dạng tổng các số nguyên dương. Ví dụ, có 8 cách biểu diễn số 4 như sau:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- $4$

Một bài toán tổ hợp thường có thể được giải bằng một hàm đệ quy. Trong bài toán này, ta có thể định nghĩa hàm  $f(n)$  cho biết số cách biểu diễn số  $n$ . Ví dụ,  $f(4) = 8$  theo ví dụ trên. Các giá trị của hàm có thể được tính một cách đệ quy như sau:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

Trường hợp cơ sở là  $f(0) = 1$  bởi vì tổng rỗng biểu diễn số 0. Sau đó, nếu  $n > 0$ , ta xét tất cả các cách chọn số đầu tiên của tổng. Nếu số đầu tiên là  $k$ , ta có  $f(n-k)$  cách biểu diễn phần còn lại của tổng. Do đó, ta tính tổng của tất cả các giá trị có dạng  $f(n-k)$  với  $k < n$ .

Những 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, ta có thể chứng minh rằng hàm  $f(n)$  có công

thức dạng đóng là:

$$f(n) = 2^{n-1},$$

rút ra từ thực tế rằng có  $n - 1$  vị trí có thể đặt dấu + trong tổng và ta có thể chọn bất kỳ tập con nào của những vị trí này để đặt vào.

## 22.1 Hệ số nhị thức

**Hệ số nhị thức**  $\binom{n}{k}$  là số cách chọn một tập con có  $k$  phần tử từ một tập có  $n$  phần tử. Ví dụ,  $\binom{5}{3} = 10$ , bởi vì tập  $\{1, 2, 3, 4, 5\}$  có 10 tập con có 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

Hệ số nhị thức có thể được tính một cách đệ quy như sau:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Ý tưởng là cố định một phần tử  $x$  trong tập. Nếu  $x$  thuộc tập con, ta phải chọn thêm  $k - 1$  phần tử từ  $n - 1$  phần tử, và nếu  $x$  không nằm trong tập con, ta phải chọn  $k$  phần tử từ  $n - 1$  phần tử còn lại.

Trường hợp cơ sở là

$$\binom{n}{0} = \binom{n}{n} = 1,$$

bởi vì chỉ có đúng một cách để xây dựng một tập con rỗng hoặc một tập con chứa tất cả các phần tử.

### Công thức 2

Có một cách khác để tính hệ số nhị thức như sau:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Có  $n!$  hoán vị của một tập có  $n$  phần tử. Ta duyệt qua tất cả các hoán vị và lấy  $k$  phần tử đầu tiên của hoán vị đưa vào tập con. Vì thứ tự của các phần tử trong tập con và ngoài tập con không quan trọng, kết quả được chia cho  $k!$  và  $(n - k)!$ .

## Tính chất

Trong hệ số nhị thức,

$$\binom{n}{k} = \binom{n}{n-k},$$

vì thực ra, khi ta chọn  $k$  phần tử vào một tập con, điều đó cũng tương đương với việc ta chọn ra  $n-k$  phần tử còn lại để loại ra khỏi tập con đó.

Tổng của các hệ số nhị thức là

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Có thể thấy được lý do đằng sau tên gọi "hệ số nhị thức" khi ta xét lũy thừa bậc  $n$  của  $(a+b)$ :

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

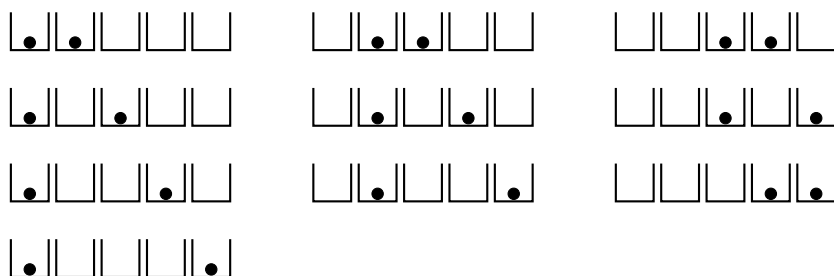
Hệ số nhị thức cũng xuất hiện trong **tam giác Pascal** trong đó mỗi giá trị bằng tổng của hai giá trị ở trên:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

## Hộp và bóng

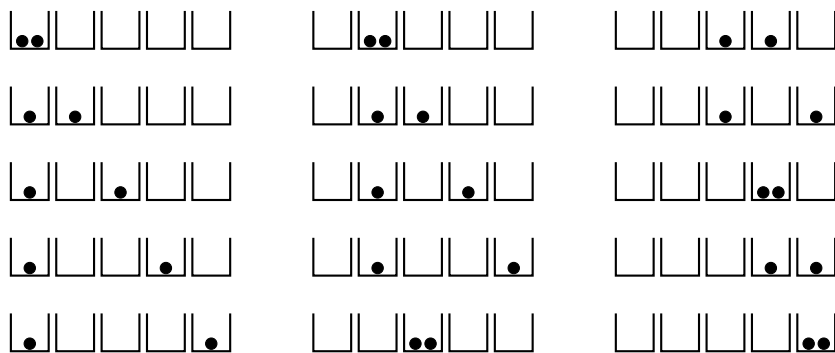
"Hộp và bóng" là một mô hình hữu ích mà trong đó, ta đếm số cách để đặt  $k$  quả bóng vào  $n$  hộp. Ta xét ba trường hợp:

*Trường hợp 1:* Mỗi hộp chỉ chứa tối đa một quả bóng. Ví dụ, với  $n = 5$  và  $k = 2$ , có 10 cách:



Trong trường hợp này, câu trả lời chính là hệ số nhị thức  $\binom{n}{k}$ .

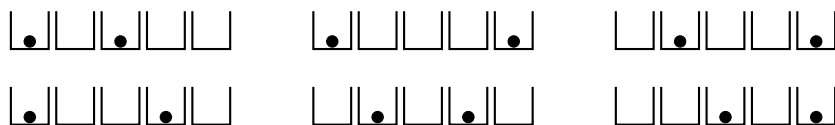
*Trường hợp 2:* Một hộp có thể chứa nhiều quả bóng. Ví dụ, với  $n = 5$  và  $k = 2$ , có 15 cách:



Quá trình đặt các quả bóng vào các hộp có thể được biểu diễn dưới dạng một xâu gồm các ký tự "o" và "→". Ban đầu, giả sử rằng ta đang đứng ở hộp đầu tiên bên trái. Ký tự "o" có nghĩa là ta đặt một quả bóng vào hộp hiện tại và ký tự "→" có nghĩa là ta di chuyển đến hộp tiếp theo bên phải.

Bằng cách sử dụng ký hiệu này, mỗi cách đặt bóng tương ứng với một xâu gồm  $k$  ký tự "o" và  $n - 1$  ký tự "→". Ví dụ, cách đặt bóng ở trên cùng bên phải trong hình ảnh tương ứng với xâu "→ → o → o →". Do đó, số cách đặt bóng là  $\binom{k+n-1}{k}$ .

*Trường hợp 3:* Mỗi hộp chỉ có thể chứa tối đa một quả bóng, và không có hai hộp liên tiếp nào cùng chứa bóng. Ví dụ, với  $n = 5$  và  $k = 2$ , có 6 cách:



Trong trường hợp này, ta có thể giả sử rằng ban đầu có  $k$  quả bóng được đặt vào các hộp và có một hộp trống giữa hai hộp chứa bóng liên tiếp. Công việc còn lại là chọn vị trí cho các hộp trống còn lại. Có  $n - 2k + 1$  hộp trống và  $k + 1$  vị trí cho chúng. Do đó, sử dụng công thức ở trường hợp 2, số cách đặt bóng là  $\binom{n-k+1}{n-2k+1}$ .

## Hệ số đa thức

### Hệ số đa thức

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

là số cách chia  $n$  phần tử thành các tập 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 xem như là một dạng tổng quát của hệ số nhị thức; nếu  $m = 2$ , công thức trên tương đương với công thức của hệ số nhị thức.

## 22.2 Số Catalan

**Số Catalan**  $C_n$  là số cách đặt các dấu ngoặc để tạo thành một dãy đúng gồm  $n$  dấu ngoặc mở và  $n$  dấu ngoặc đóng.

Ví dụ,  $C_3 = 5$  bởi vì ta có thể tạo ra các dãy ngoặc đúng sau đây bằng cách sử dụng ba dấu ngoặc mở và ba dấu ngoặc đóng:

- $()()()$
- $((()))$
- $()(())$
- $((())())$
- $((())())$

## Dãy ngoặc

*Dãy ngoặc đúng* thực chất là gì? *Dãy ngoặc đúng* là một dãy ngoặc được xây dựng dựa trên các quy tắc sau:

- Dãy rỗng là một dãy ngoặc đúng.
- Nếu  $A$  là một dãy ngoặc đúng, thì  $(A)$  cũng là một dãy ngoặc đúng.
- Nếu  $A$  và  $B$  là hai dãy ngoặc đúng, thì dãy  $AB$  cũng là một dãy ngoặc đúng.

Một cách khác để mô tả các dãy ngoặc đúng là nếu ta chọn bất kỳ tiền tố nào của nó, số lượng ngoặc mở phải lớn hơn hoặc bằng số lượng ngoặc đóng. Ngoài ra, dãy ngoặc đúng phải có số lượng ngoặc đóng và số lượng ngoặc mở bằng nhau.

## Công thức 1

Số Catalan có thể được tính bằng cách sử dụng công thức

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

Công thức này tương ứng với việc duyệt qua các cách chia dãy thành hai phần sao cho mỗi phần đều là dãy ngoặc đúng và phần đầu tiên ngắn nhất có thể nhưng không rỗng. Với mỗi  $i$ , phần đầu tiên chứa  $i + 1$  cặp ngoặc và số lượng dãy ngoặc đúng là tích của các giá trị sau:

- $C_i$ : số cách tạo ra một dãy ngoặc đúng bằng cách sử dụng các dấu ngoặc của phần đầu tiên, không tính cặp ngoặc ngoài cùng
- $C_{n-i-1}$ : số cách tạo ra một dãy ngoặc đúng bằng cách sử dụng các dấu ngoặc của phần thứ hai

Trường hợp cơ sở là  $C_0 = 1$ , bởi vì ta có thể tạo ra một dãy ngoặc rỗng bằng cách không sử dụng cặp ngoặc nào.

## Công thức 2

Số Catalan cũng có thể được tính bằng hệ số nhị thức:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Công thức này có thể được giải thích như sau:

Có tổng cộng  $\binom{2n}{n}$  cách để tạo ra một dãy ngoặc (không nhất thiết phải đúng) chứa  $n$  dấu ngoặc mở và  $n$  dấu ngoặc đóng. Ta sẽ tính số lượng các dãy ngoặc không đúng.

Nếu một dãy ngoặc không đúng, ta có thể tìm được một tiền tố chứa nhiều ngoặc đóng nhiều hơn ngoặc mở. Ý tưởng là đảo ngược mỗi dấu ngoặc trong tiền tố này. Ví dụ, dãy  $()()() ($  chứa tiền tố  $()()$ , và sau khi đảo ngược tiền tố này, dãy trở thành  $((() ($ .

Dãy thu được chứa  $n+1$  dấu ngoặc mở và  $n-1$  dấu ngoặc đóng. Số lượng các dãy ngoặc như vậy là  $\binom{2n}{n+1}$ , bằng với số lượng các dãy ngoặc không đúng. Do đó, số lượng các dãy ngoặc đúng có thể được tính bằng công thức

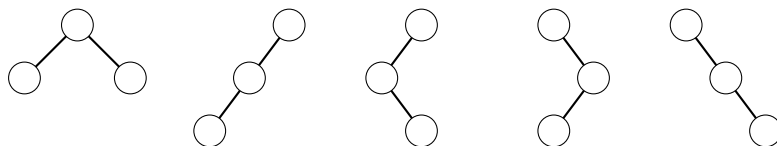
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

## Đếm cây

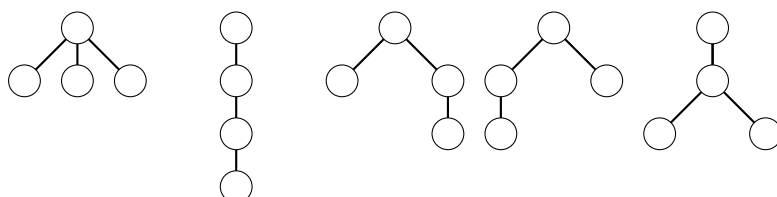
Số Catalan cũng liên quan đến cây:

- Có  $C_n$  cây nhị phân gồm  $n$  đỉnh
- Có  $C_{n-1}$  cây có gốc gồm  $n$  đỉnh

Ví dụ, với  $C_3 = 5$ , các cây nhị phân là



và các cây có gốc là

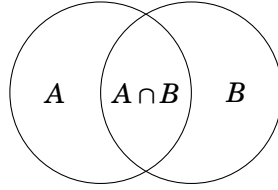


## 22.3 Bao hàm-loại trừ

**Bao hàm-loại trừ** là một kỹ thuật có thể được sử dụng để đếm số phần tử của phần hợp của các tập khi đã biết kích thước của các phần giao, và ngược lại. Một ví dụ đơn giản của kỹ thuật này là công thức

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

trong đó  $A$  và  $B$  là các tập và  $|X|$  được sử dụng để biểu diễn kích thước của tập  $X$ . Công thức này có thể được minh họa như sau:

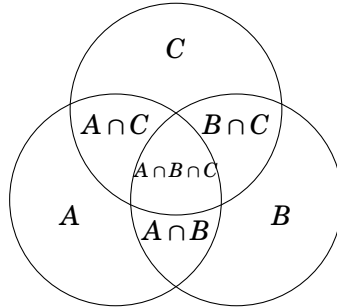


Mục tiêu của ta là tính kích thước của phần hợp  $A \cup B$  tương ứng với diện tích của vùng thuộc về ít nhất một hình tròn. Hình vẽ cho thấy ta có thể tính diện tích của  $A \cup B$  bằng cách trước tiên cộng diện tích của  $A$  và  $B$  lại sau đó trừ đi diện tích của  $A \cap B$ .

Ý tưởng tương tự có thể được áp dụng khi số lượng các tập lớn hơn. Khi có ba tập, 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 vẽ tương ứng là



Trong trường hợp tổng quát, kích thước của phần hợp  $X_1 \cup X_2 \cup \dots \cup X_n$  có thể được tính bằng cách duyệt qua tất cả các phần giao của một vài tập trong  $X_1, X_2, \dots, X_n$ . Nếu phần giao chứa một số lẻ các tập, kích thước của nó được cộng vào kết quả, ngược lại kích thước của nó được trừ khỏi kết quả.

Lưu ý rằng ta có công thức tương tự để tính kích thước của phần giao từ kích thước của phần 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 không bất động

Ta sẽ đi đếm đếm số lượng **hoán vị không bất động** của các phần tử  $\{1, 2, \dots, n\}$ , tức là các hoán vị mà không có phần tử nào ở vị trí ban đầu của nó. Ví dụ, khi  $n = 3$ , có hai hoán vị không bất động:  $(2, 3, 1)$  và  $(3, 1, 2)$ .

Một cách tiếp cận để giải quyết bài toán này là sử dụng bao hàm-loại trừ. Gọi  $X_k$  là tập các hoán vị chứa phần tử  $k$  ở vị trí  $k$ . Ví dụ, khi  $n = 3$ , các tập là 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 này, số lượng hoán vị không bất động bằng

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

vì vậy chỉ cần tính kích thước của phần hợp. Sử dụng bao hàm-loại trừ, bài toán đưa về tính kích thước của các phần giao, điều này có thể được thực hiện một cách hiệu quả.  $|X_1 \cup X_2 \cup X_3|$  bằng

$$\begin{aligned}&|X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\&= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\&= 4,\end{aligned}$$

vậy số lượng hoán vị không bất động là  $3! - 4 = 2$ .

Bài toán này cũng có thể giải được mà không cần sử dụng bao hàm-loại trừ. Gọi  $f(n)$  là số lượng hoán vị không bất động của  $\{1, 2, \dots, n\}$ . Ta có thể sử dụng công thức 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 này có thể được suy ra bằng cách xét mọi khả năng của số 1 trong hoán vị không bất động. Có  $n-1$  cách để chọn một phần tử  $x$  thay thế vào vị trí của số 1. Trong mỗi cách chọn như thế, có hai trường hợp:

*Lựa chọn 1:* Ta cũng thay số 1 vào vị trí ban đầu của phần tử  $x$ . Sau đó, bài toán còn lại là xây dựng một hoán vị không bất động của  $n-2$  phần tử.

*Lựa chọn 2:* Ta thay số 1 vào vị trí khác với vị trí ban đầu của phần tử  $x$ . Bây giờ ta phải xây dựng một hoán vị không bất động cho cả  $n-1$  phần tử, vì không thể thay thế phần tử  $x$  bằng phần tử 1, và tất cả mọi phần tử khác đều phải đổi vị trí.

## 22.4 Bổ đề Burnside

**Bổ đề Burnside** có thể được sử dụng để đếm số lượng cấu hình mà ta chỉ tính một đại diện trong mỗi một nhóm các cấu hình đối xứng. Bổ đề

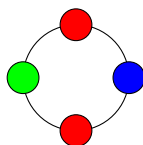


Burnside cho biết số lượng cấu hình là

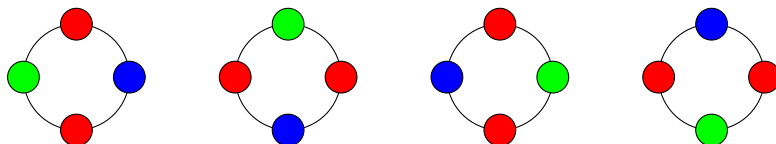
$$\sum_{k=1}^n \frac{c(k)}{n},$$

trong đó có  $n$  cách để thay đổi vị trí của một cấu hình, và có  $c(k)$  cấu hình vẫn giống như cũ khi áp dụng cách thứ  $k$ .

Ví dụ, hãy tính số lượng dây chuyền có  $n$  ngọc trai, mỗi ngọc trai có  $m$  màu khác nhau. Hai dây chuyền được gọi là đối xứng nếu sau khi xoay vòng, chúng giống nhau. Ví dụ, dây chuyền



có các dây chuyền đối xứng sau:



Có  $n$  cách để thay đổi vị trí của một dây chuyền, vì ta có thể xoay nó  $0, 1, \dots, n-1$  bước theo chiều kim đồng hồ. Nếu số bước là 0, tất cả  $m^n$  dây chuyền vẫn giữ nguyên, và nếu số bước là 1, chỉ có  $m$  dây chuyền giống như cũ, đó là những dây mà mọi ngọc trai của nó có cùng màu.

Tổng quát hơn, khi số bước là  $k$ , có tổng cộng

$$m^{\gcd(k,n)}$$

dây chuyền giống ban đầu, trong đó  $\gcd(k,n)$  là ước chung lớn nhất của  $k$  và  $n$ . Lý do cho điều này là các đoạn ngọc trai có kích thước  $\gcd(k,n)$  sẽ thay thế cho nhau. Do đó, theo bổ đề Burnside, số lượng dây chuyền là

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

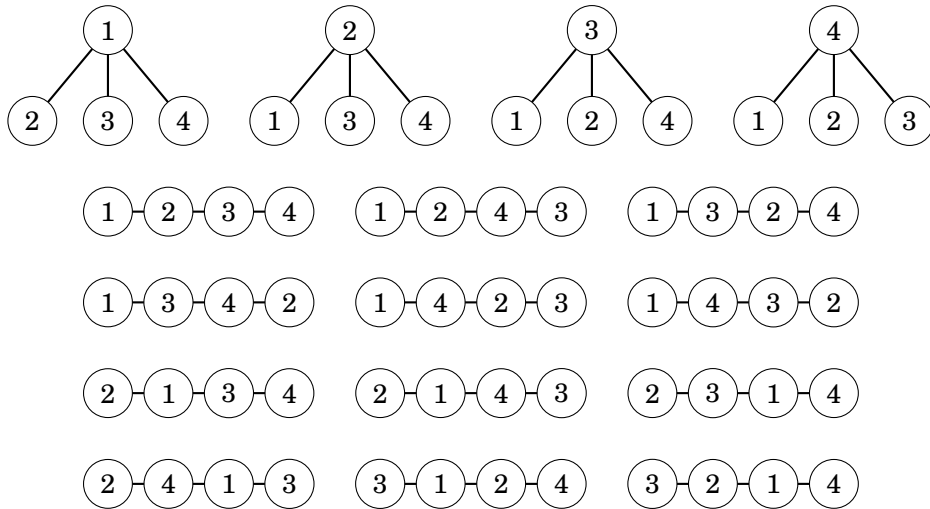
Ví dụ, số lượng dây chuyền có độ dài 4 với 3 màu là

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

## 22.5 Công thức Cayley

**Công thức Cayley** phát biểu rằng có  $n^{n-2}$  cách để gán nhãn cho cây gồm  $n$  đỉnh. Các đỉnh được gán nhãn bằng các số từ 1 đến  $n$ , và hai cây khác nhau nếu cấu trúc hoặc cách gán nhãn của chúng khác nhau.

Ví dụ, khi  $n = 4$ , số lượng cách gán nhãn cho cây là  $4^{4-2} = 16$ :

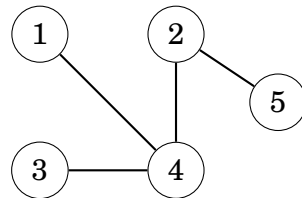


Tiếp theo ta sẽ xem cách rút ra công thức Cayley từ mã Prüfer.

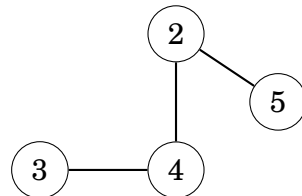
## Mã Prüfer

**Mã Prüfer** là một dãy gồm  $n - 2$  số để mô tả một cây có nhãn. Mã được xây dựng bằng cách lần lượt xoá  $n - 2$  đỉnh lá khỏi cây. Ở mỗi bước, ta xoá đỉnh lá có nhãn nhỏ nhất, và nhãn của đỉnh kề duy nhất với nó được thêm vào mã.

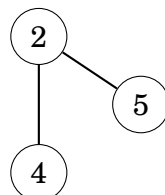
Ví dụ, hãy cùng tính mã Prüfer của đồ thị sau:



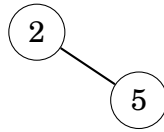
Đầu tiên ta xoá đỉnh 1 và thêm đỉnh 4 vào mã:



Sau đó ta xoá đỉnh 3 và thêm đỉnh 4 vào mã:



Cuối cùng ta xoá đỉnh 4 và thêm đỉnh 2 vào mã:



Do đó, mã Prüfer của đồ thị là  $[4, 4, 2]$ .

Ta có thể xây dựng mã Prüfer cho bất kỳ cây nào, và quan trọng hơn là, cây ban đầu có thể được xây dựng lại từ mã Prüfer tương ứng. Vì thế, số lượng cách đánh số cho cây  $n$  đỉnh là  $n^{n-2}$ , bằng với số lượng mã Prüfer có độ dài  $n$ .



# Chương 23

## Ma trận

Khái niệm **ma trận** trong toán học tương đương 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 có kích thước  $3 \times 4$ , tức là, nó có 3 hàng và 4 cột. Ký hiệu  $[i, j]$  tương ứng với giá trị ở hàng  $i$  và cột  $j$  trong một ma trận. Ví dụ, với ma trận trên,  $A[2, 3] = 8$  và  $A[3, 1] = 9$ .

Một trường hợp đặc biệt của ma trận là **vector**, vector là một ma trận một chiều có kích thước  $n \times 1$ . Ví dụ,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

là một vector chứa 3 phần tử.

**Ma trận chuyển vị**  $A^T$  của ma trận  $A$  là ma trận nhận được khi ta hoán đổi các hàng và cột của  $A$ , tức là  $A^T[i, j] = A[j, i]$ :

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 2 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Một ma trận là **ma trận vuông** nếu nó có cùng số hàng và số cột. Ví dụ, ma trận sau là một ma trận vuông:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

### 23.1 Phép tính

Phép tính cộng  $A + B$  được định nghĩa với hai ma trận  $A$  và  $B$  có cùng kích thước. Kết quả là một ma trận mà trong đó, mỗi phần tử là tổng của hai

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 ma trận  $A$  với một giá trị  $x$  có nghĩa là mỗi phần tử của  $A$  được nhân với  $x$ . Ví dụ,

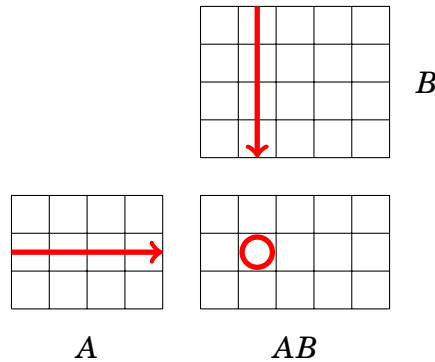
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

## Phép nhân ma trận

Tích  $AB$  của hai 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 (số cột) của  $A$  bằng chiều cao (số hàng) của  $B$ . Kết quả là một ma trận có 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 đó là mỗi phần tử của  $AB$  là tổng của tích các phần tử tương ứng trong  $A$  và  $B$ , như hình sau:



Ví dụ,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Phép nhân ma trận có tính chất kết hợp, nghĩa là  $A(BC) = (AB)C$ , nhưng nó không có tính chất giao hoán, nghĩa là  $AB = BA$  không đúng với mọi  $A, B$ .

**Ma trận đơn vị** là một ma trận vuông mà mọi phần tử trên đường chéo đều bằng 1 và tất cả các phần tử khác đều bằng 0. Ví dụ, ma trận sau đây là một ma trận đơn vị  $3 \times 3$ :

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Phép nhân một ma trận với ma trận đơn vị không làm thay đổi ma trận đó. Ví dụ,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{và} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Bằng một thuật toán đơn giản, có thể tính được tích của hai ma trận  $n \times n$  trong độ phức tạp  $O(n^3)$ . Ngoài ra còn có các thuật toán hiệu quả hơn để nhân ma trận<sup>1</sup>, nhưng chúng thường chỉ có ý nghĩa trên lý thuyết và không cần thiết trong lập trình thi đấu.

## Luỹ thừa của ma trận

Luỹ thừa  $A^k$  của một ma trận  $A$  được định nghĩa nếu  $A$  là một ma trận vuông. Định nghĩa này dựa trên phép nhân ma trận:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ lần}}$$

Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Ngoài ra,  $A^0$  là một ma trận đơn vị. Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Ma trận  $A^k$  có thể được tính toán một cách hiệu quả trong độ phức tạp  $O(n^3 \log k)$  bằng thuật toán ở Chương 21.2. Ví dụ,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

## Định thức

**Định thức**  $\det(A)$  của một ma trận  $A$  được định nghĩa nếu  $A$  là một ma trận vuông. Nếu  $A$  có kích thước  $1 \times 1$ , thì  $\det(A) = A[1, 1]$ . Định thức của một ma trận lớn hơn được tính bằng đệ quy theo công thức

$$\det(A) = \sum_{j=1}^n A[1, j]C[1, j],$$

<sup>1</sup>Thuật toán đầu tiên được công bố là thuật toán của Strassen, được công bố vào năm 1969 [63], độ phức tạp của thuật toán này là  $O(n^{2.80735})$ ; hiện nay thuật toán tốt nhất [27] chạy trong độ phức tạp  $O(n^{2.37286})$ .

trong đó  $C[i, j]$  là **hệ số phụ** của  $A$  tại  $[i, j]$ . Hệ số phụ được tính bằng công thức

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

trong đó  $M[i, j]$  là ma trận thu được bằng cách loại bỏ hàng  $i$  và cột  $j$  của  $A$ . Do hệ số  $(-1)^{i+j}$  có trong công thức, nên mọi định thức khác có giá trị dương hoặc âm. Ví dụ,

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

và

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

Định thức của  $A$  cho biết có tồn tại một **ma trận nghịch đảo**  $A^{-1}$  sao cho  $A \cdot A^{-1} = I$ , trong đó  $I$  là ma trận đơn vị. Như vậy  $A^{-1}$  tồn tại khi và chỉ khi  $\det(A) \neq 0$ , và nó có thể tính được bằng công thức

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Ví dụ,

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

## 23.2 Hàm truy hồi tuyến tính

Một **hàm truy hồi tuyến tính** là một hàm  $f(n)$  có các giá trị ban đầu là  $f(0), f(1), \dots, f(k-1)$  và các giá trị lớn hơn được tính theo công thức truy hồi

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

trong đó các hệ số  $c_1, c_2, \dots, c_k$  là hằng.

Quy hoạch động có thể được sử dụng để tính bất kỳ giá trị  $f(n)$  nào trong độ phức tạp  $O(kn)$  bằng cách tính lần lượt các giá trị  $f(0), f(1), \dots, f(n)$ . Tuy nhiên, nếu  $k$  nhỏ, thì có thể tính  $f(n)$  một cách hiệu quả hơn trong  $O(k^3 \log n)$  bằng cách sử dụng các phép tính ma trận.

### Dãy số Fibonacci

Một ví dụ đơn giản của một hàm 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 các số Fibonacci một cách hiệu quả, ta biểu diễn công thức Fibonacci dưới dạng một ma trận vuông  $X$  kích thước  $2 \times 2$ , trong đó:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Như vậy, các giá trị  $f(i)$  và  $f(i+1)$  được cho dưới dạng "đầu vào" của  $X$ , và  $X$  tính các giá trị  $f(i+1)$  và  $f(i+2)$  từ chúng. Có một ma trận thỏa mãn:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Ví dụ,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Vì thế, ta có thể tính  $f(n)$  bằng công thức

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Giá trị của  $X^n$  có thể được tính trong  $O(\log n)$ , vì vậy giá trị của  $f(n)$  cũng có thể được tính trong độ phức tạp  $O(\log n)$ .

## Trường hợp tổng quát

Bây giờ xét trường hợp tổng quát hơn khi  $f(n)$  là một hàm truy hồi tuyến tính bất kỳ. Mục tiêu của ta là xây dựng một ma trận  $X$  sao cho

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Một ma trận thỏa mãn là

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

Ở  $k-1$  hàng đầu tiên, mỗi phần tử bằng 0 ngoại trừ một phần tử bằng 1. Những hàng này thay thế  $f(i)$  bằng  $f(i+1)$ ,  $f(i+1)$  bằng  $f(i+2)$ , và cứ tiếp tục như vậy. Hàng cuối cùng chứa các hệ số của công thức truy hồi để tính giá trị mới  $f(i+k)$ .

Bây giờ,  $f(n)$  có thể được tính trong độ phức tạp  $O(k^3 \log n)$  bằng công thức

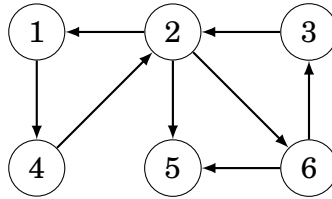
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

## 23.3 Đồ thị và ma trận

### Đếm số lượng đường đi

Lũy thừa của một ma trận kề của một đồ thị có một tính chất thú vị. Khi  $V$  là ma trận kề của một đồ thị không có trọng số, thì ma trận  $V^n$  chứa số lượng đường đi gồm  $n$  cạnh giữa các đỉnh trong đồ thị.

Ví dụ, xét đồ thị sau:



Ma trận kề của đồ thị này 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

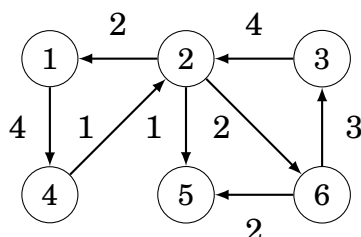
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

sẽ chứa số lượng đường đi gồm 4 cạnh giữa các đỉnh trong đồ thị. Chẳng hạn,  $V^4[2,5] = 2$ , vì có hai đường đi gồm 4 cạnh từ đỉnh 2 đến đỉnh 5:  $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$  và  $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$ .

## Đường đi ngắn nhất

Sử dụng ý tưởng tương tự trong một đồ thị có trọng số, ta có thể tính với mỗi cặp đỉnh đường đi ngắn nhất giữa chúng sao cho đường đi này có đúng  $n$  cạnh. Để tính được điều này, ta phải định nghĩa phép nhân ma trận theo một cách mới, để cực tiểu hoá độ dài đường đi ngắn nhất thay vì tính số lượng đường đi.

Ví dụ, xét đồ thị sau:



Ta xây dựng một ma trận kề mà trong đó nếu ô  $(i, j)$  bằng  $\infty$  tức nghĩa là không tồn tại cạnh  $(i, j)$ , ngược lại thì trọng số cạnh  $(i, j)$  chính bằng giá trị trong ô đó. Ma trận kề của đồ thị này 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 công thức

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

thành công thức

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j]$$

cho phép nhân ma trận, ta tính được giá trị nhỏ nhất thay vì tổng, và lấy tổng các phần tử trong  $A, B$  thay vì tích. Nhờ sự thay đổi này, lũy thừa ma trận giờ tương ứng với đường đi ngắn nhất trong đồ thị.

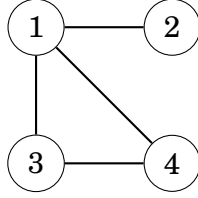
Ví dụ, vì

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

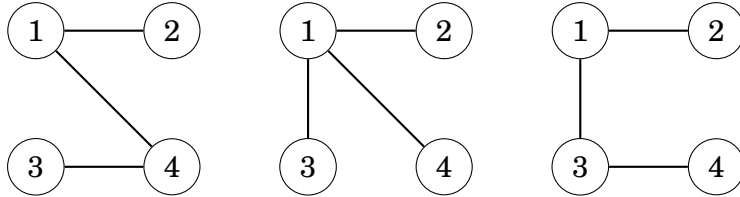
có thể kết luận rằng đường đi ngắn nhất gồm 4 cạnh từ đỉnh 2 đến đỉnh 5 là 8. Một đường đi như vậy là  $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ .

## Định lý Kirchhoff

**Định lý Kirchhoff** giúp ta tính số lượng cây khung của một đồ thị dưới dạng định thức của một ma trận đặc biệt. Ví dụ, đồ thị



có ba cây khung:



Để tính số lượng cây khung, ta xây dựng một **ma trận Laplace**  $L$ , trong đó  $L[i, i]$  là bậc của đỉnh  $i$  và  $L[i, j] = -1$  nếu có cạnh nối giữa đỉnh  $i$  và  $j$ , ngược lại  $L[i, j] = 0$ . Ma trận Laplace của đồ thị trên là:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Có thể thấy rằng số lượng cây khung bằng định thức của ma trận nhận được khi xóa đi một hàng và một cột bất kỳ của  $L$ . Ví dụ, nếu ta xóa đi hàng và cột đầu tiên, kết quả thu được là

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

Định thức luôn như nhau, bất kể có xóa đi hàng và cột nào của  $L$ .

Lưu ý rằng công thức Cayley ở chương 22.5 là một trường hợp đặc biệt của định lý Kirchhoff, vì trong một đồ thị đầy đủ gồm  $n$  đỉnh

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

# Chương 24

## Xác suất

Xác suất là một số thực trong khoảng từ 0 đến 1 cho biết khả năng xảy ra của một biến cố. Nếu một biến cố chắc chắn 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 biểu thị bởi  $P(\dots)$ , trong đó dấu ba chấm mô tả biến cố.

Ví dụ, khi tung một xúc xắc, kết quả là một số nguyên từ 1 đến 6, và xác suất mỗi mặt xuất hiện là  $1/6$ . Ví dụ, ta có thể tính được những xác suất sau:

- $P(\text{"kết quả là 4"}) = 1/6$
- $P(\text{"kết quả không phải là 6"}) = 5/6$
- $P(\text{"kết quả là một số nguyên chẵn"}) = 1/2$

### 24.1 Tính toán

Để tính xác suất của một biến cố, ta có thể sử dụng toán tổ hợp hoặc mô phỏng lại quá trình xảy ra biến cố. Ví dụ, chúng ta hãy tính xác suất để rút được ba lá bài có giá trị giống nhau từ một bộ bài (Ví dụ,  $\spadesuit 8$ ,  $\clubsuit 8$  và  $\diamondsuit 8$ ).

#### Phương pháp 1

Ta có thể tính được xác suất bằng công thức

$$\frac{\text{số lượng kết quả mong muốn}}{\text{tổng số kết quả}}.$$

Ở bài này, kết quả mong muốn là những kết quả trong đó giá trị của các lá bài bốc được là như nhau. Có  $13 \binom{4}{3}$  kết quả như vậy, vì có 13 khả năng cho giá trị của các quân bài và  $\binom{4}{3}$  cách để chọn 3 lá bài từ 4 lá bài cho trước.

Có tổng cộng  $\binom{52}{3}$  kết quả, vì ta cần chọn 3 lá bài từ 52 lá bài. Do vậy, xác suất của biến cố trên 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 lại quá trình xảy ra biến cố. Ở ví dụ trên, ta phải rút 3 lá, do đó quá trình gồm ba bước. Ta phải bảo đảm rằng tại mỗi bước trong quá trình, ta rút được lá bài thỏa mãn.

Ta chắc chắn sẽ rút được ở lượt đầu tiên, vì chưa có hạn chế nào cả. Xác suất để rút đúng ở lượt thứ hai là  $3/51$ , vì còn lại 51 lá bài và có 3 trong số chúng có giá trị trùng với lá đầu tiên. Tương tự, xác suất rút trúng ở lượt thứ ba là  $2/50$ .

Xác suất của toàn bộ quá trình trên là

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

## 24.2 Biến cố

Một biến cố trong lý thuyết xác suất có thể được biểu diễn dưới dạng một tập hợp

$$A \subset X,$$

trong đó  $X$  bao gồm tất cả kết quả có thể xảy ra và  $A$  là một tập các kết quả. Ví dụ, khi tung một xúc xắc, tập các kết quả có thể là

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Bây giờ, chẳng hạn, "kết quả là một 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 công thức

$$P(A) = \sum_{x \in A} p(x).$$

Ví dụ, khi tung một xúc xắc,  $p(x) = 1/6$  với mỗi kết quả  $x$ , vậy nên xác suất của "kết quả là chẵn" là

$$p(2) + p(4) + p(6) = 1/2.$$

Tổng xác suất của các kết quả trong tập  $X$  phải bằng 1, nghĩa là  $P(X) = 1$ .

Do các biến cố trong lý thuyết xác suất là các tập hợp, ta có thể thao tác với chúng bằng cách sử dụng các phép toán tập hợp.

- **Phần bù**  $\bar{A}$  có nghĩa là " $A$  không xảy ra". Ví dụ, khi tung xúc xắc, phần bù của  $A = \{2, 4, 6\}$  là  $\bar{A} = \{1, 3, 5\}$ .
- **Hợp**  $A \cup B$  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**  $A \cap B$  có nghĩa là " $A$  và  $B$  đều xảy ra". Ví dụ, giao của  $A = \{2, 5\}$  và  $B = \{4, 5, 6\}$  là  $A \cap B = \{5\}$ .

## Phần bù

Xác suất của phần bù  $\bar{A}$  được tính bằng công thức

$$P(\bar{A}) = 1 - P(A).$$

Đôi khi, ta có thể giải quyết một bài toán một cách dễ dàng hơn bằng cách tính phần bù của bài toán đối ngược. Ví dụ, xác suất để tung được ít nhất một mặt sáu khi tung một xúc xắc mười lần là

$$1 - (5/6)^{10}.$$

Ở đây,  $5/6$  là xác suất mà kết quả của một lần tung xúc xắc không phải là sáu, và  $(5/6)^{10}$  là xác suất mà không lần tung nào có kết quả là sáu. Phần bù của số này chính là câu trả lời cho bài toán trên.

## Hợp

Xác suất của phép hợp  $A \cup B$  được tính bằng công thức

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Ví dụ, khi tung một xúc xắc, hợp của hai biến cố

$$A = \text{"kết quả là chẵn"}$$

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  $B$  **rời nhau**, tức là  $A \cap B$  rỗng, xác suất của biến cố  $A \cup B$  chỉ đơn giản là

$$P(A \cup B) = P(A) + P(B).$$

## Xác suất có điều kiện

### Xác suất có điều kiện

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

là xác suất của biến cố  $A$  khi biết rằng biến cố  $B$  xảy ra. Do đó, khi tính xác suất của  $A$ , ta chỉ xem xét các kết quả cũng thuộc về  $B$ .

Sử dụng dữ kiện trước đó,

$$P(A|B) = 1/3,$$

Vì tập các kết quả thuộc  $B$  là 1, 2, 3, và chỉ có một trong số chúng là chẵn. Đây chính là xác suất của một kết quả chẵn nếu chúng ta biết rằng kết quả nằm trong khoảng 1...3.

## Giao

Sử dụng xác suất có điều kiện, xác suất của phép 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** nếu

$$P(A|B) = P(A) \quad \text{và} \quad P(B|A) = P(B),$$

Điều đó có nghĩa là việc biến cố  $B$  có xảy ra không làm thay đổi xác suất của biến cố  $A$ , và ngược lại. Trong trường hợp này, xác suất của phép giao là

$$P(A \cap B) = P(A)P(B).$$

Ví dụ, khi rút một lá bài từ một bộ bài, các biến cố

$$A = \text{"chất của lá bài rút được là chuồn"}$$

và

$$B = \text{"giá trị lá bài rút được là bốn"}$$

là độc lập. Do đó, biến cố

$$A \cap B = \text{"lá bài rút được là bốn chuồn"}$$

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

**Biến ngẫu nhiên** là một giá trị được tạo ra bởi một quá trình ngẫu nhiên. Ví dụ, khi tung hai xúc xắc, một biến ngẫu nhiên có thể là

$$X = \text{"Tổng các kết quả"}.$$

Ví dụ, nếu kết quả tung được là  $[4, 6]$  (nghĩa là đầu tiên ta tung được bốn và sau đó là sáu), thì giá trị của  $X$  là 10.

Ta ký hiệu  $P(X = x)$  là xác suất mà giá trị của biến ngẫu nhiên  $X$  bằng  $x$ . Ví dụ, khi tung hai xúc xắc,  $P(X = 10) = 3/36$ , vì tổng số kết quả là 36 và có ba cách để đạt được tổng bằng 10:  $[4, 6]$ ,  $[5, 5]$  và  $[6, 4]$ .

### Giá trị kỳ vọng

**Giá trị kỳ vọng**  $E[X]$  biểu thị giá trị trung bình của một biến ngẫu nhiên  $X$ . Giá trị kỳ vọng có thể được tính bằng tổng

$$\sum_x P(X = x)x,$$



trong đó  $x$  duyệt qua tất cả các giá trị có thể của  $X$ .

Ví dụ, khi tung một xúc xắc, giá trị kỳ vọng là

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Một tính chất hữu ích của giá trị kỳ vọng là **tuyến tính**. 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 tung hai xúc xắc, tổng kỳ vọng là

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Bây giờ, 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 trống kỳ vọng. Mỗi quả bóng có một xác suất như nhau để được đặt trong bất kỳ hộp nào. Ví dụ, nếu  $n = 2$ , có các khả năng như sau:



Trong trường hợp này, số hộp trống kỳ vọng là

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

Trong trường hợp tổng quát, xét một hộp bất kì, xác suất hộp đó trống là

$$\left(\frac{n-1}{n}\right)^n,$$

bởi vì không có quả bóng nào được đặt trong hộp đó. Do đó, bằng cách sử dụng tính chất tuyến tính, giá trị kỳ vọng của số hộp trống là

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

## Phân phối

**Phân phối** của một biến ngẫu nhiên  $X$  cho biết xác suất của mỗi giá trị mà  $X$  có thể có. Phân phối bao gồm các giá trị  $P(X = x)$ . Ví dụ, khi tung hai xúc xắc, 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**, biến ngẫu nhiên  $X$  có thể có  $n$  giá trị  $a, a+1, \dots, b$  và xác suất của mỗi giá trị là  $1/n$ . Ví dụ, khi tung xúc xắc,  $a = 1$ ,  $b = 6$  và  $P(X = x) = 1/6$  với mỗi giá trị  $x$ .

Giá trị kỳ vọng của  $X$  trong một phân phối đều là

$$E[X] = \frac{a+b}{2}.$$

Trong một **phân phối nhị thức**, có  $n$  lần thử được thực hiện và xác suất thành công của mỗi lần thử là  $p$ . Biến ngẫu nhiên  $X$  đếm số lần thử thành công, và xác suất của mỗi 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 những lần thử thành công và không thành công, và  $\binom{n}{x}$  là số cách ta có thể chọn thứ tự cho các lần thử.

Ví dụ, khi tung một xúc xắc mười lần, xác suất để tung được mặt sáu đúng ba lần là  $(1/6)^3(5/6)^7 \binom{10}{3}$ .

Giá trị kỳ vọng của  $X$  trong một phân phối nhị thức là

$$E[X] = pn.$$

Trong một **phân phối cấp số nhân**, xác suất thành công của một lần thử là  $p$ , và ta sẽ tiếp tục cho đến khi lần thử thành công đầu tiên xảy ra. Biến ngẫu nhiên  $X$  đếm số lượng lần thử kỳ vọng, 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 những lần thử không thành công và  $p$  tương ứng với lần thử đầu tiên thành công.

Ví dụ, nếu ta tung xúc xắc cho đến khi tung được mặt sáu, xác suất để ta cần tung đúng 4 lần là  $(5/6)^3 1/6$ .

Giá trị kỳ vọng của  $X$  trong một phân phối cấp số nhân là

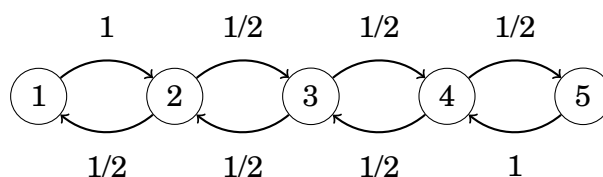
$$E[X] = \frac{1}{p}.$$

## 24.4 Chuỗi Markov

**Chuỗi Markov** là một quá trình ngẫu nhiên chứa các trạng thái và sự chuyển tiếp giữa chúng. Với mỗi trạng thái, ta biết xác suất để di chuyển từ trạng thái đó đến những trạng thái khác. Một chuỗi Markov có thể được biểu diễn dưới dạng một đồ thị với mỗi đỉnh tương ứng với một trạng thái và mỗi cạnh tương ứng với một sự chuyển tiếp.

Ví dụ, xét một bài toán, chúng ta đang ở tầng một của một tòa nhà  $n$  tầng. Ở mỗi lượt di chuyển, ta ngẫu nhiên di chuyển lên một tầng hoặc xuống một tầng, ngoại trừ việc ta luôn di chuyển lên khi ở tầng một và di chuyển xuống khi ở tầng  $n$ . Tính xác suất để chúng ta kết thúc ở tầng  $m$  sau  $k$  lượt di chuyển?

Ở bài toán này, mỗi tầng của tòa nhà tương ứng với một trạng thái của một chuỗi Markov. Ví dụ, nếu  $n = 5$ , đồ thị như sau:



Phân phối xác suất của một chuỗi Markov là một vector  $[p_1, p_2, \dots, p_n]$ , với  $p_k$  là xác suất để trạng thái hiện tại là  $k$ . Đồng thức  $p_1 + p_2 + \dots + p_n = 1$  luôn đúng.

Trong trường hợp trên, phân phối ban đầu là  $[1, 0, 0, 0, 0]$ , vì ta luôn xuất phát ở tầng 1. Phân phối tiếp theo là  $[0, 1, 0, 0, 0]$ , vì ta chỉ có thể di chuyển từ tầng 1 đến tầng 2. Sau đó, ta có thể di chuyển lên một tầng hoặc xuống một tầng, vì vậy phân phối tiếp theo là  $[1/2, 0, 1/2, 0, 0]$ , và cứ tiếp tục như thế.

một cách hiệu quả để mô phỏng lại việc di chuyển trong một chuỗi Markov là sử dụng quy hoạch động. Ý tưởng chính là duy trì phân phối xác suất, ở mỗi bước, ta xét hết tất cả các khả năng mà ta có thể di chuyển. Bằng phương pháp này, ta có thể mô phỏng lại  $m$  bước di chuyển trong thời gian  $O(n^2m)$ .

Những chuyển tiếp trạng thái trong một chuỗi Markov cũng có thể được biểu diễn dưới dạng một ma trận, ma trận đó có thể dùng để cập nhật lại phân phối xác suất. Trong trường hợp 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 ta nhân một phân phối xác suất với ma trận này, ta nhận được phân phối mới ứng với sau khi di chuyển một bước. Ví dụ, 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 toán lũy thừa ma trận một cách hiệu quả, ta có thể tính được phân phối sau  $m$  bước trong thời gian  $O(n^3 \log m)$ .

## 24.5 Thuật toán ngẫu nhiên

Đôi lúc ta có thể sử dụng tính ngẫu nhiên để giải quyết vấn đề, ngay cả khi bài toán đó không liên quan đến xác suất. Một **thuật toán ngẫu nhiên** là thuật toán được dựa trên sự ngẫu nhiên.

**Thuật toán Monte Carlo** là thuật toán ngẫu nhiên mà đôi khi có thể đưa ra câu trả lời sai. Để một thuật toán như vậy trở nên hữu ích, xác suất của một câu trả lời sai phải nhỏ.

**Thuật toán Las Vegas** là thuật toán ngẫu nhiên luôn đưa ra câu trả lời chính xác, nhưng thời gian chạy của nó lại thay đổi ngẫu nhiên. Mục đích là thiết kế một thuật toán hiệu quả với xác suất cao.

Tiếp theo chúng ta sẽ đi qua ba bài toán ví dụ có thể được giải quyết bằng cách sử dụng tính ngẫu nhiên.

## Thông kê thứ tự

**Thông kê thứ tự  $k$**  của một mảng là phần tử ở vị trí  $k$  sau khi sắp xếp lại mảng theo thứ tự tăng dần. Có thể dễ dàng tính được thông kê thứ tự bất kì của mảng trong thời gian  $O(n \log n)$  bằng cách sắp xếp lại mảng, nhưng có thật sự cần thiết phải sắp xếp lại toàn bộ mảng chỉ để tìm một phần tử?

Trên thực tế, ta có thể tìm được thông kê thứ tự của một mảng bằng cách sử dụng một thuật toán ngẫu nhiên mà không cần phải sắp xếp lại mảng. Thuật toán đó được gọi là **quickselect**<sup>1</sup>, 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, sau đó di chuyển những phần tử nhỏ hơn  $x$  sang phần bên trái của mảng, và chuyển tất cả các phần tử khác vào phần bên phải của mảng. Việc này mất thời gian  $O(n)$  trong trường hợp có  $n$  phần tử. Giả sử rằng phần bên trái chứa  $a$  phần tử và phần bên phải chứa  $b$  phần tử. nếu  $a = k$ , phần tử  $x$  chính là thông kê thứ tự  $k$  của mảng. Ngược lại, nếu  $a > k$ , ta đệ quy xuống để tìm thông kê thứ tự  $k$  cho phần bên trái của mảng, và nếu  $a < k$ , ta tìm thông kê thứ tự  $r$  cho phần bên phải của mảng với  $r = k - a$ . Việc tìm kiếm được tiếp tục theo cách tương tự cho đến khi tìm được phần tử mong muốn.

Mỗi khi một phần tử  $x$  được chọn một cách ngẫu nhiên, kích thước của mảng giảm khoảng một nửa, do đó độ phức tạp để tìm thông kê thứ tự  $k$  là khoảng

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

Trường hợp xấu nhất của thuật toán vẫn mất thời gian  $O(n^2)$ , vì có khả năng  $x$  luôn được chọn sao cho nó là phần tử nhỏ nhất hoặc lớn nhất của mảng khiến ta cần  $O(n)$  bước để hoàn thành. Tuy nhiên, xác suất để điều này xảy ra là rất nhỏ đến nỗi mà nó chưa bao giờ xảy ra trong thực tế.

## Kiểm tra phép nhân ma trận

Bài toán tiếp theo của chúng ta là *kiểm tra* xem  $AB = C$  có đúng không nếu  $A$ ,  $B$  và  $C$  là các ma trận kích thước  $n \times n$ . Tất nhiên ta có thể giải quyết bài toán bằng cách tính kết quả của phép nhân  $AB$  (trong thời gian  $O(n^3)$  bằng cách sử dụng thuật toán cơ bản), nhưng người ta kỳ vọng rằng việc xác minh câu trả lời sẽ dễ dàng hơn là tính toán lại từ đầu.

Trên thực tế, ta có thể giải quyết bài toán trên bằng một thuật toán Monte Carlo<sup>2</sup> Với 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, sau đó tính các ma

<sup>1</sup>Năm 1961, C. A. R. Hoare đã công bố hai thuật toán có hiệu suất trung bình tốt: **quicksort** [36] cho việc sắp xếp mảng và **quickselect** [37] cho việc tìm thông kê thứ tự.

<sup>2</sup>R. M. Freivalds đã công bố thuật toán này vào năm 1977 [26], và đôi khi nó được gọi là **thuật toán Freivalds**.

trận  $ABX$  và  $CX$ . Nếu  $ABX = CX$ , ta kết luận rằng  $AB = C$ , ngược lại thì ta kết luận  $AB \neq C$ .

Độ phức tạp về thời gian của thuật toán là  $O(n^2)$ , vì ta có thể tính được ma trận  $ABX$  và  $CX$  trong thời gian  $O(n^2)$ . Ta có thể tính được ma trận  $ABX$  một cách hiệu quả bằng cách viết  $A(BX)$ , vì vậy chỉ cần thực hiện hai phép nhân ma trận kích thước  $n \times n$  và  $n \times 1$ .

Thuật toán trên có một hạn chế, có một xác suất nhỏ rằng nó kết luận sai  $AB = C$ . Ví dụ,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

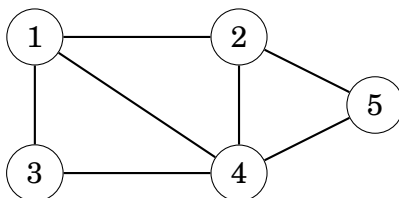
nhưng

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

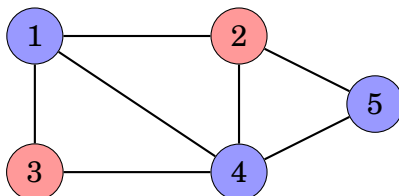
Tuy nhiên, trên thực tế, xác suất để thuật toán này mắc lỗi khá nhỏ, và ta có thể giảm xác suất này bằng cách xác minh kết quả bằng nhiều vector ngẫu nhiên trước khi kết luận rằng  $AB = C$ .

## Tô màu đồ thị

Cho một đồ thị vô hướng gồm  $n$  đỉnh và  $m$  cạnh, nhiệm vụ của chúng ta là tìm một cách để tô các đỉnh của đồ thị bằng hai màu sao cho có ít nhất  $m/2$  cạnh mà các đầu mút của nó được tô bởi hai màu khác nhau. Ví dụ, trong đồ thị



một cách tô màu hợp lệ như sau:



Đồ thị trên gồm 7 cạnh, và có 5 cạnh mà hai đầu mút được tô màu khác nhau, do đó cách tô trên hợp lệ. Bài toán có thể được giải quyết bằng thuật toán Las Vegas tô màu ngẫu nhiên cho đến khi một cách tô hợp lệ được tìm thấy. Trong một cách tô ngẫu nhiên, màu của mỗi đỉnh được chọn một cách độc lập sao cho xác suất của cả hai màu là  $1/2$ .

Trong một cách tô ngẫu nhiên, xác suất để hai đầu mút của một cạnh được tô khác màu là  $1/2$ . Do đó, giá trị kỳ vọng của số cạnh mà hai đầu mút được tô khác màu là  $m/2$ . Vì kỳ vọng rằng một cách tô màu ngẫu nhiên là hợp lệ, ta sẽ nhanh chóng tìm được một cách tô màu hợp lệ trong thực tế.



# Chương 25

## Lý thuyết trò chơi

Trong chương này, ta sẽ tập trung vào những trò chơi hai người chơi và không chứa các yếu tố ngẫu nhiên. Mục tiêu của chúng ta là tìm một chiến thuật chơi để luôn thắng bất kể đối thủ có làm gì đi nữa, nếu có tồn tại một chiến thuật như vậy.

Thực tế, các trò chơi dạng này đều có chung một chiến thuật tổng quát, và ta có thể phân tích chúng bằng **lý thuyết nim**. Đầu tiên, ta sẽ phân tích một số trò chơi đơn giản mà trong đó người chơi sẽ lấy bớt một vài que diêm từ nhiều đồng diêm khác nhau, sau đó, ta sẽ dựa vào chiến thuật dùng trong những trò chơi nêu trên mà tổng quát hóa vào các trò chơi khác.

### 25.1 Trạng thái trò chơi

Xét một trò chơi như sau: Ban đầu có một đồng diêm gồm  $n$  que diêm. Hai người chơi  $A$  và  $B$  sẽ thay phiên nhau chơi theo lượt, và người chơi  $A$  bắt đầu trước. Ở mỗi lượt chơi, người chơi sẽ phải lấy đi 1, 2 hoặc 3 que diêm từ đồng diêm, và người nào lấy được que diêm cuối cùng sẽ dành chiến thắng.

Ví dụ, nếu  $n = 10$ , trò chơi có thể diễn ra như sau:

- Người chơi  $A$  lấy đi 2 que diêm (còn lại 8 que diêm).
- Người chơi  $B$  lấy đi 3 que diêm (còn lại 5 que diêm).
- Người chơi  $A$  lấy đi 1 que diêm (còn lại 4 que diêm).
- Người chơi  $B$  lấy đi 2 que diêm (còn lại 2 que diêm).
- Người chơi  $A$  lấy đi 2 que diêm và dành chiến thắng

Trò chơi sẽ có các trạng thái  $0, 1, 2, \dots, n$ , trong đó con số của mỗi trạng thái tương ứng với số lượng que diêm còn lại.

#### Trạng thái thắng và trạng thái thua

Một **trạng thái thắng** là trạng thái mà người chơi sẽ dành chiến thắng nếu họ chơi tối ưu, và một **trạng thái thua** là một trạng thái mà người chơi sẽ thua nếu đối thủ chơi tối ưu. Thực tế, ta có thể phân loại tất cả các trạng thái của trò chơi thành một trong hai loại trạng thái này.

Xét trò chơi ở trên, trạng thái 0 chắc chắn là một trạng thái thua, vì người chơi không thể làm gì được nữa. Trạng thái 1, 2 và 3 là các trạng thái thắng, vì ta có thể lấy đi 1, 2 hoặc 3 que diêm và giành chiến thắng. Do đó, trạng thái 4 là một trạng thái thua, vì bất kể lấy bao nhiêu que diêm ở trạng thái này cũng đều dẫn đến một trạng thái thắng cho đối thủ.

Một cách tổng quát, nếu tồn tại một nước đi dẫn tới một trạng thái thua, trạng thái hiện tại sẽ là một trạng thái thắng, ngược lại trạng thái hiện tại sẽ là một trạng thái thua. Bằng nhận xét này, ta có thể phân loại tất cả các trạng thái của một trò chơi, bắt đầu từ những trạng thái thua - những trạng thái mà tại đó người chơi không thể đi thêm một nước đi nào nữa.

Các trạng thái 0...15 của trò chơi nêu trên có thể được phân loại như sau ( $W$  đại diện cho trạng thái thắng và  $L$  đại diện cho trạng thái thua):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$

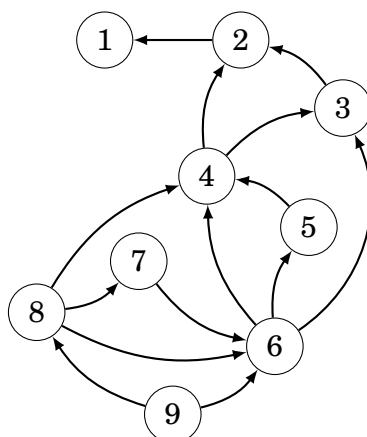
Phân tích trò chơi nêu trên là khá đơn giản: một trạng thái  $k$  là một trạng thái thua nếu  $k$  chia hết cho 4, ngược lại nó sẽ là trạng thái thắng. Một cách tối ưu để chơi trò chơi này là luôn chọn số lượng que diêm để lấy đi sao cho số lượng que diêm còn lại trong đồng diêm là một số chia hết cho 4. Tới cuối cùng, sẽ không còn bất kỳ que diêm nào và đối thủ sẽ thua cuộc.

Dĩ nhiên, chiến thuật này yêu cầu số lượng que diêm *không* chia hết cho 4 khi tới lượt của ta. Nếu như số này chia hết cho 4, ta sẽ chẳng thể làm được gì khác, và đối thủ sẽ chiến thắng trò chơi nếu họ chơi tối ưu.

## Đồ thị trạng thái

Ta sẽ xét một trò chơi với que diêm khác, trong đó với mỗi trạng thái  $k$ , ta được lấy đi  $x$  que diêm sao cho  $x$  nhỏ hơn  $k$  và là ước của  $k$ . Ví dụ, ở trạng thái 8 ta sẽ được lấy đi 1, 2 hoặc 4 que diêm, nhưng ở trạng thái 7 ta chỉ được phép lấy đi 1 que diêm.

Hình sau đây minh họa các trạng thái 1...9 của trò chơi dưới dạng một **đồ thị trạng thái**, trong đó các đỉnh là các trạng thái và các cạnh là các nước đi:





Trạng thái cuối cùng của trò chơi này sẽ luôn là 1, một trạng thái thua, vì không còn nước đi hợp lệ nào nữa. Ta phân loại các trạng thái 1...9 như sau:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Ngạc nhiên thay, trong trò chơi này, tất cả trạng thái chẵn là các trạng thái thắng, trong khi đó tất cả trạng thái lẻ là các trạng thái thua.

## 25.2 Trò chơi Nim

**Trò chơi nim** là một trò chơi đơn giản nhưng có vai trò quan trọng trong lý thuyết trò chơi, vì rất nhiều trò chơi khác có thể áp dụng cùng một chiến thuật tương tự. Đầu tiên, ta sẽ tập trung vào nim, sau đó ta sẽ tổng quát hóa chiến thuật chơi trò này sang các trò chơi khác.

Trong trò chơi nim, có  $n$  đồng diêm, mỗi đồng diêm chứa một số lượng que diêm nhất định. Hai người chơi sẽ thay phiên nhau chơi theo lượt, vào mỗi lượt, người chơi sẽ chọn một đồng diêm nào đó vẫn còn diêm và lấy đi một số que diêm bất kỳ từ đó. Người nào lấy được que diêm cuối cùng sẽ là người dành chiến thắng.

Các trạng thái trong nim có dạng  $[x_1, x_2, \dots, x_n]$ , trong đó  $x_k$  chỉ số lượng que diêm còn lại trong đồng diêm thứ  $k$ . Ví dụ,  $[10, 12, 5]$  là một trạng thái tương ứng với ba đồng diêm chứa 10, 12 và 5 que diêm. Trạng thái  $[0, 0, \dots, 0]$  là một trạng thái thua, vì ta không thể lấy đi bất kỳ que diêm nào, và đây luôn là trạng thái cuối cùng.

### Phân tích

Thực tế, ta có thể dễ dàng phân loại các trạng thái nim bằng cách tính **tổng nim**  $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$ , trong đó  $\oplus$  là phép xor<sup>1</sup>. Các trạng thái mà tổng nim bằng 0 đều là các trạng thái thua, và tất cả các trạng thái còn lại đều là trạng thái thắng. Ví dụ, tổng nim của  $[10, 12, 5]$  là  $10 \oplus 12 \oplus 5 = 3$ , nên trạng thái này là một trạng thái thắng.

Nhưng tổng nim liên quan gì đến trò chơi nim? Ta có thể giải thích điều này bằng cách quan sát sự thay đổi của tổng nim khi trạng thái của trò chơi 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 ta kỳ vọng. Trong các trạng thái thua khác, bất kỳ nước đi nào cũng có thể dẫn đến một trạng thái thắng, vì khi một giá trị  $x_k$  thay đổi, tổng nim cũng sẽ thay đổi, nên tổng nim sẽ khác 0 sau nước đi.

*Trạng thái thắng:* Ta có thể chuyển sang một trạng thái thua nếu tồn tại một đồng diêm thứ  $k$  nào đó mà  $x_k \oplus s < x_k$ . Trong trường hợp này, ta có thể

<sup>1</sup>Chiến thuật tối ưu cho nim được công bố vào năm 1901 bởi C. L. Bouton [10].

lấy đi một vài que diêm từ đồng diêm thứ  $k$  sao cho sau đó nó có  $x_k \oplus s$  que diêm. Lúc này ta đã chuyển sang được một trạng thái thua. Sẽ luôn luôn tồn tại một đồng diêm  $x_k$  nào đó có bit một tại vị trí bit một trái nhất của  $s$ .

Ta xét trạng thái  $[10, 12, 5]$  làm ví dụ. Đây là một trạng thái thắng, vì tổng nim của nó là 3. Do đó, phải có một nước đi có thể dẫn tới một trạng thái thua. Tiếp theo ta sẽ tìm ra nước đi này.

Tổng nim của trạng thái này được tính như sau:

$$\begin{array}{r|l} 10 & 1010 \\ 12 & 1100 \\ 5 & 0101 \\ \hline 3 & 0011 \end{array}$$

Trong trường hợp này, đồng diêm có 10 que diêm là đồng diêm duy nhất có bit một ở vị trí bit một trái nhất của tổng nim:

$$\begin{array}{r|l} 10 & 10\underline{1}0 \\ 12 & 1100 \\ 5 & 0101 \\ \hline 3 & 00\underline{1}1 \end{array}$$

Đồng diêm sau đó sẽ phải còn lại  $10 \oplus 3 = 9$  que diêm, nên ta sẽ lấy đi đúng 1 que. Sau đó, trạng thái sẽ trở thành  $[9, 12, 5]$ , một trạng thái thua.

$$\begin{array}{r|l} 9 & 1001 \\ 12 & 1100 \\ 5 & 0101 \\ \hline 0 & 0000 \end{array}$$

## Trò chơi Misère

Trong **trò chơi misère**, mục tiêu của người chơi đối ngược so với trò chơi nim: người chơi nào lấy đi que diêm cuối cùng sẽ là người thua cuộc. Có thể chơi trò chơi misère nim một cách tối ưu gần giống cách chơi trò chơi nim tiêu chuẩn.

Đầu tiên ta sẽ chơi trò chơi misère như một trò chơi nim tiêu chuẩn, sau đó thay đổi chiến thuật ở cuối trò chơi. Ta áp dụng chiến thuật mới khi mà ở mỗi đồng diêm còn nhiều nhất một que diêm sau nước đi tiếp theo.

Ở trò chơi tiêu chuẩn, ta nên chọn nước đi sao cho số lượng đồng diêm có đúng một que diêm là số chẵn. Tuy vậy, trong trò chơi misère, ta chọn nước đi sao cho con số này là số lẻ.

Nguyên nhân áp dụng chiến thuật này là vì luôn tồn tại một trạng thái khiến ta phải thay đổi chiến thuật, và đây là một trạng thái thắng vì nó có đúng một đồng diêm có nhiều hơn một que diêm nên tổng nim sẽ khác 0.

## 25.3 Định lý Sprague–Grundy

**Định lý Sprague–Grundy**<sup>2</sup> tổng quát hóa chiến thuật dùng trong trò chơi nim cho tất cả các trò chơi thỏa mãn những điều kiện sau:

- Có hai người chơi, họ chơi theo lượt.
- Trò chơi có thể chia thành các trạng thái, và các nước đi hợp lệ ở một trạng thái không phụ thuộc vào việc đang là lượt của ai (tức là luôn như nhau bất kể người chơi là ai).
- Trò chơi kết thúc khi một người chơi không còn nước đi.
- Trò chơi chắc chắn phải kết thúc, dù sớm hay muộn.
- Người chơi biết toàn bộ thông tin về các trạng thái cũng như các nước đi hợp lệ, và trò chơi không chứa yếu tố ngẫu nhiên.

Ý tưởng ở đây là, với mỗi trạng thái trò chơi, ta tính một con số Grundy, tương ứng với số lượng que diêm trong một đồng diêm của nim. Khi ta biết số Grundy của tất cả các trạng thái, ta có thể chơi trò chơi này giống với nim.

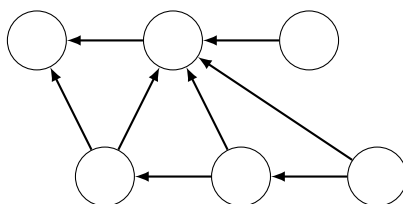
### Số Grundy

**Số Grundy** của một trạng thái trò chơi là

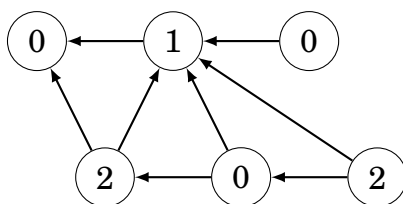
$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

trong đó  $g_1, g_2, \dots, g_n$  là số Grundy của các trạng thái có thể đi tới từ trạng thái hiện tại, và hàm mex trả về số nguyên không âm nhỏ nhất không nằm trong tập hợp. Ví dụ,  $\text{mex}(\{0, 1, 3\}) = 2$ . Nếu như tại trạng thái này không có nước đi nào hợp lệ, số Grundy của nó sẽ là 0, vì  $\text{mex}(\emptyset) = 0$ .

Ví dụ, trong đồ thị trạng thái sau:



các số Grundy sẽ là:



<sup>2</sup>Định lý này được nghiên cứu độc lập bởi R. Sprague [61] và P. M. Grundy [31].

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 lượng que diêm trong một đồng diêm của nim. Nếu số Grundy là 0, ta chỉ có thể di chuyển tới các trạng thái có số Grundy dương, và nếu số Grundy là  $x > 0$ , ta có thể di chuyển tới các trạng thái có số Grundy trong khoảng  $0, 1, \dots, x - 1$ .

Lấy ví dụ bằng một trò chơi như sau: người chơi sẽ di chuyển một nhân vật trong ma trận. Mỗi ô vuông trong ma trận sẽ hoặc là sàn nhà hoặc là tường. Ở mỗi lượt, người chơi phải di chuyển nhân vật đi sang trái hoặc đi lên một số bước. Người cuối cùng đi được một nước đi hợp lệ sẽ là người dành chiến thắng.

Hình sau đây cho thấy một trạng thái xuất phát của trò chơi, trong đó @ chỉ nhân vật và \* chỉ ô vuông mà nhân vật có thể đi tới.

				*
				*
*	*	*	*	@

Trạng thái của trò chơi sẽ là tất cả các ô sàn nhà của ma trận. Trong ma trận trên, ta có 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

Ta có thể thấy mỗi trạng thái của trò chơi trên ma trận này tương ứng với một đồng diêm trong trò chơi nim. Ví dụ, số Grundy tại ô góc phải dưới là 2, nên đây là một trạng thái thắng. Ta có thể đi tới một trạng thái thua và dành chiến thắng bằng cách di chuyển 4 bước sang trái hoặc 2 bước lên trên.

Lưu ý rằng không giống với trò chơi nim gốc, ta có thể di chuyển tới một trạng thái có số Grundy lớn hơn trạng thái hiện tại. Tuy nhiên, đối thủ luôn có thể chọn một nước đi để hủy nước đi này. Do đó, việc thoát khỏi một trạng thái thua là không thể.

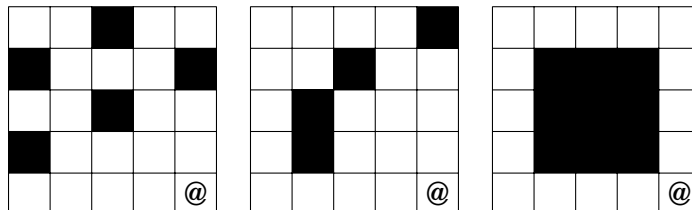
## Trò chơi con

Tiếp theo chúng ta sẽ giả sử rằng trò chơi sẽ bao gồm nhiều trò chơi con. Trong mỗi lượt, người chơi trước tiên chọn một trò chơi con và sau đó thực

hiện một nước đi trong đây. Trò chơi kết thúc khi một người chơi không thể đưa ra một nước đi hợp lệ trong bất kỳ tròn 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. Ta có thể chơi như trò chơi nim bình thường bằng cách tính tất cả số Grundy của các trò chơi con rồi tính tổng nim của chúng.

Lấy ví dụ một trò chơi gồm ba ma trận. Trong trò chơi này, ở mỗi lượt, người chơi sẽ chọn một trong các ma trận và di chuyển nhân vật trong ma trận đó. Giả sử trạng thái xuất phát của trò chơi là như sau:



Các số Grundy cho các ma trận sẽ là:

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 xuất phát, tổng nim của các số Grundy là  $2 \oplus 3 \oplus 3 = 2$ , nên người chơi thứ nhất có thể dành chiến thắng. Một nước đi tối ưu là đi lên trên 2 bước ở ma trận đầu tiên và được tổng nim là  $0 \oplus 3 \oplus 3 = 0$ .

## Trò chơi của Grundy

Đôi khi một nước đi trong trò chơi sẽ chia trò chơi thành nhiều 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 lượng các nước đi hợp lệ và

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

trong đó nước đi thứ  $k$  sẽ tạo ra các trò chơi con có số Grundy là  $a_{k,1}, a_{k,2}, \dots, a_{k,m}$ .

Một ví dụ cụ thể là **trò chơi của Grundy**. Ban đầu, có một đồng diêm duy nhất gồm  $n$  que diêm. Ở mỗi lượt, người chơi sẽ chọn một đồng diêm và chia nó thành hai đồng diêm khác rỗng sao cho hai phần có kích thước khác nhau. Người cuối cùng thực hiện được một cách chia hợp lệ là người thắng cuộc.

Đặt  $f(n)$  là số Grundy của một đồng diêm gồm  $n$  que diêm. Số Grundy có thể được tính bằng cách duyệt qua tất cả các cách chia đồng diêm thành hai

đồng diêm nhỏ hơn. Ví dụ, với  $n = 8$ , các cách chia sẽ là  $1 + 7$ ,  $2 + 6$  and  $3 + 5$ , nên

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

Trong trò chơi này, giá trị  $f(n)$  dựa vào các giá trị  $f(1), \dots, f(n-1)$ . Các trường hợp cơ sở là  $f(1) = f(2) = 0$ , vì ta không thể chia những đồng diêm có 1 hay 2 que diêm được. Các số Grundy đầu tiên sẽ là:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

Số Grundy cho trường hợp  $n = 8$  là 2, nên ta có thể dành chiến thắng trò chơi. Để chiến thắng, ta có thể chia đồng diêm thành  $1 + 7$ , vì  $f(1) \oplus f(7) = 0$ .

# Chương 26

## Các thuật toán trên xâu

Trong chương này, ta sẽ bàn về một số thuật toán xử lý xâu một cách hiệu quả. Có nhiều bài toán trên xâu có thể giải được một cách dễ dàng trong  $\mathcal{O}(n^2)$ , nhưng điều khó hơn là tìm những thuật toán có thời gian chạy  $\mathcal{O}(n)$  hoặc  $\mathcal{O}(n \log n)$ .

Ví dụ, một bài toán xử lý xâu kinh điển là bài toán **so khớp mẫu**<sup>1</sup>: Cho một xâu độ dài  $n$  và một xâu mẫu độ dài  $m$ , việc của chúng ta là phải tìm những vị trí xuất hiện của mẫu trong xâu. Ví dụ, mẫu ABC xuất hiện hai lần trong xâu ABABCBABC.

Bài toán so khớp mẫu có thể dễ dàng giải được trong thời gian  $\mathcal{O}(nm)$  bằng thuật toán vét cạn duyệt qua tất cả các vị trí mà mẫu có thể xuất hiện trong xâu. Tuy nhiên, trong chương này, chúng ta sẽ thấy rằng có nhiều thuật toán hiệu quả hơn mà chỉ cần  $\mathcal{O}(n + m)$  thời gian.

### 26.1 Các thuật ngữ về xâu

Trong suốt chương này, chúng ta giả sử rằng xâu được đánh chỉ số từ 0. Vì thế, một xâu  $s$  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 xâu được gọi là **bảng chữ cái**. Ví dụ, bảng chữ cái  $\{A, B, \dots, Z\}$  bao gồm các ký tự Tiếng Anh viết hoa.

Một **xâu con** là một dãy các ký tự liên tiếp trong một xâu. Chúng ta sử dụng ký hiệu  $s[a \dots b]$  để chỉ một xâu con của  $S$  bắt đầu từ vị trí  $a$  và kết thúc tại vị trí  $b$ . Một xâu độ dài  $n$  có  $n(n+1)/2$  xâu con. Ví dụ, các xâu con của xâu ABCD là: A, B, C, D, AB, BC, CD, ABC, BCD và ABCD.

Một **dãy con** là một dãy các ký tự (không nhất thiết liên tiếp nhau) nằm theo thứ tự ban đầu trong một xâu. Một xâu độ dài  $n$  có  $2^n - 1$  dãy con. Ví dụ, các dãy con của ABCD gồm: A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD và ABCD.

Một **tiền tố** là một xâu con bắt đầu tại vị trí đầu tiên của xâu, và một **hậu tố** là một xâu con kết thúc tại vị trí cuối cùng của xâu. 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.

<sup>1</sup>Đa số các tài liệu tiếng Việt gọi là *so khớp chuỗi*

Một **hoán vị vòng** có thể được tạo nên bằng cách lần lượt di chuyển các ký tự ở đầu xâu đến cuối xâu (hoặc ngược lại). Ví dụ, các hoán vị vòng của ABCD gồm ABCD, BCDA, CDAB và DABC.

**Chu kỳ** là tiền tố của một xâu, sao cho xâu đó có thể được dựng lên bằng cách lặp lại tiền tố đó nhiều lần. Phần lặp lại cuối cùng có thể không đầy đủ và chỉ chứa một vài ký tự đầu của chu kỳ đó. Ví dụ, chu kỳ nhỏ nhất của xâu ABCABCA là ABC.

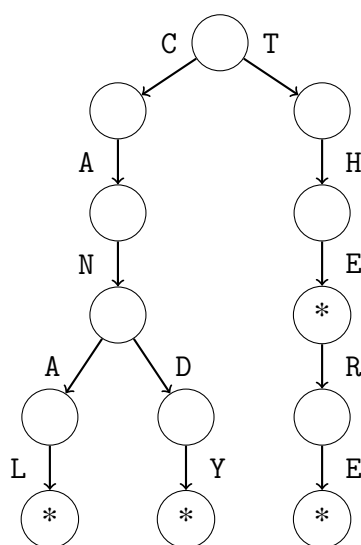
Một **xâu biên** là một xâu mà vừa là tiền tố và hậu tố của một xâu. Ví dụ, các biên của ABACABA là A, ABA và ABACABA.

Các xâu được so sánh với nhau thông qua **thứ tự từ điển** (tương ứng với thứ tự trong bảng chữ cái). Điều đó có nghĩa là  $x < y$  nếu  $x \neq y$  và  $x$  là tiền tố của  $y$ , hoặc tồn tại một vị trí  $k$  sao cho  $x[i] = y[i]$  với mọi  $i < k$  và  $x[k] < y[k]$ .

## 26.2 Cấu trúc Trie

**Trie** là một cây có gốc, dùng để duy trì một tập hợp các xâu. Mỗi xâu trong tập được lưu trữ dưới dạng một chuỗi các ký tự xuất phát từ gốc. Nếu hai xâu có cùng một tiền tố, chúng cũng sẽ trùng một phần chuỗi ban đầu trên cây.

Ví dụ, xét trie dưới đây:



Cây Trie trên tương ứng với tập các xâu {CANAL, CANDY, THE, THERE}. Ký tự \* trong một số đỉnh đánh dấu rằng có một xâu trong tập kết thúc tại đỉnh đó. Cần có một ký tự đánh dấu như vậy, bởi vì một xâu có thể là tiền tố của một xâu khác. Ví dụ, cũng ở trie trên, THE là một tiền tố của THERE.

Chúng ta có thể kiểm tra trong thời gian  $\mathcal{O}(n)$  liệu một trie có chứa một xâu độ dài  $n$  hay không, vì chúng ta có thể đi lần theo từng ký tự của chuỗi xuất phát từ đỉnh gốc. Ta cũng có thể thêm một xâu độ dài  $n$  vào trie trong thời gian  $\mathcal{O}(n)$  bằng cách đi theo chuỗi và thêm các đỉnh mới vào cây khi cần thiết.



Sử dụng một cây trie, chúng ta có thể tìm ra tiền tố dài nhất của một xâu bất kỳ, sao cho tiền tố đó nằm trong tập hợp các xâu chứa trong trie. Hơn nữa, bằng việc lưu trữ thêm thông tin ở mỗi đỉnh, chúng ta có thể tính số lượng xâu có trong tập mà nhận một xâu cho trước làm tiền tố.

Một cây trie có thể được lưu trữ dưới dạng một mảng hai chiều

```
1 int trie[N][A];
```

với  $N$  là số lượng đỉnh tối đa (tổng tối đa độ dài của các xâu trong tập hợp) và  $A$  là số lượng ký tự của bảng chữ cái. Mỗi đỉnh của trie được đánh số  $0, 1, 2, \dots$  sao cho đỉnh gốc là đỉnh  $0$ , và  $\text{trie}[s][c]$  là đỉnh tiếp theo được đi đến trong chuỗi khi ta di chuyển từ đỉnh  $s$  thông qua ký tự  $c$ .

## 26.3 Băm chuỗi

**Băm chuỗi** là một kỹ thuật giúp chúng ta kiểm tra hai xâu có trùng khớp với nhau hay không một cách hiệu quả<sup>2</sup>. Ý tưởng của việc băm chuỗi là ta sẽ so sánh giá trị băm của các chuỗi, thay vì so sánh từng ký tự một của chúng.

### Tính toán giá trị băm

**Giá trị băm** của một chuỗi là một con số tính được từ các ký tự của xâu đó. Nếu hai xâu giống nhau, giá trị băm của hai xâu cũng giống nhau, nên ta có thể so khớp hai xâu thông qua giá trị băm của chúng.

Một cách thường dùng để cài đặt băm xâu là thông qua hàm **băm đa thức**, điều đó có nghĩa là giá trị băm của xâu  $s$  độ dài  $n$  bằng

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

với  $s[0], s[1], \dots, s[n-1]$  được xem như là mã của mỗi ký tự trong xâu  $s$ , và  $A, B$  là các hằng số được chọn trước.<sup>3</sup>

Ví dụ, mã của các ký tự trong xâu ALLEY là:

A	L	L	E	Y
65	76	76	69	89

Vì thế, nếu  $A = 3$  và  $B = 97$ , giá trị băm của ALLEY là

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

<sup>2</sup>Kỹ thuật này được phổ biến bởi thuật toán so khớp pattern Karp-Rabin [42].

<sup>3</sup>Trong một số tài liệu tiếng Việt,  $B$  còn được gọi là hằng số *modulo*.

## Tiền xử lý

Với kỹ thuật băm đa thức, chúng ta có thể tính giá trị băm của bất kỳ xâu con nào của xâu  $s$  trong  $\mathcal{O}(1)$  sau khi tiền xử lý trong thời gian  $\mathcal{O}(n)$ . Ý tưởng là xây dựng một mảng  $h$  sao cho  $h[k]$  lưu lại giá trị băm của tiền tố  $s[0 \dots k]$ . Các giá trị của mảng có thể được tính thông qua công thức truy hồi sau:

$$\begin{aligned}h[0] &= s[0] \\h[k] &= (h[k-1]A + s[k]) \bmod B\end{aligned}$$

Thêm vào đó, ta xây dựng mảng  $p$  với  $p[k] = A^k \bmod B$ :

$$\begin{aligned}p[0] &= 1 \\p[k] &= (p[k-1]A) \bmod B.\end{aligned}$$

Xây dựng những mảng này tốn thời gian  $\mathcal{O}(n)$ . Sau đó, giá trị băm của bất kỳ xâu con  $s[a \dots b]$  đều có thể được tính trong  $\mathcal{O}(1)$  bằng công thức

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

giá sử rằng  $a > 0$ . Nếu  $a = 0$ , giá trị băm đơn giản chỉ là  $h[b]$ .

## Sử dụng giá trị băm

Chúng ta có thể so sánh xâu một cách hiệu quả sử dụng giá trị băm. Thay vì phải so sánh từng ký tự trong xâu, ta sẽ so sánh giá trị băm của chúng. Nếu giá trị băm bằng nhau, hai xâu *khả năng cao* là bằng nhau, và nếu giá trị băm khác nhau, hai xâu *chắc chắn* khác nhau.

Kỹ thuật băm thường được dùng để khiến một thuật toán vét cạn trở nên hiệu quả hơn. Ví dụ, xét bài toán so khớp mẫu như sau: Cho một xâu  $s$  và một xâu mẫu  $p$ , tìm các vị trí mà  $p$  xuất hiện trong  $s$ . Một thuật toán duyệt vét cạn sẽ đi qua tất cả các vị trí mà  $p$  có thể xuất hiện, và so sánh các xâu đó theo lần lượt từng ký tự. Độ phức tạp thời gian của thuật toán này là  $\mathcal{O}(n^2)$ .

Chúng ta có thể khiến thuật toán duyệt hiệu quả hơn bằng kỹ thuật băm, vì thuật toán của chúng ta cần so sánh các xâu con của  $s$  với  $p$ . Việc băm xâu giúp mỗi phép so sánh chỉ tốn thời gian  $\mathcal{O}(1)$ , vì ta chỉ sử dụng các giá trị băm để so sánh. Điều này dẫn tới việc thuật toán sẽ chạy trong thời gian  $\mathcal{O}(n)$ , là độ phức tạp tốt nhất đối với bài này.

Bằng việc kết hợp băm và *tìm kiếm nhị phân*, ta cũng có thể tìm ra thứ tự từ điển của hai xâu trong thời gian logarit. Điều này có thể được thực hiện bằng cách tính độ dài tiền tố chung của hai xâu bằng tìm kiếm nhị phân. Một khi đã biết được độ dài của tiền tố chung, chúng ta chỉ cần kiểm tra ký tự tiếp theo sau tiền tố đó để quyết định thứ tự trước sau của hai xâu này.

## Va chạm và tham số modulo

Có nguy cơ **va chạm** xảy ra khi so sánh các giá trị băm, điều đó có nghĩa là hai chuỗi có nội dung khác nhau nhưng lại có cùng một giá trị băm. Trong trường hợp này, một thuật toán sử dụng giá trị băm sẽ kết luận rằng hai chuỗi này bằng nhau, nhưng thực tế chúng lại khác nhau, và thuật toán có thể cho ra kết quả sai.

Va chạm luôn có thể xảy ra, vì số lượng chuỗi khác nhau lớn hơn số lượng giá trị băm khác nhau. Tuy nhiên, khả năng va chạm hiếm khi xảy ra nếu chúng ta chọn hai hằng số  $A$  và  $B$  đủ tốt. Một cách chọn số thông dụng là chọn hai hằng số ngẫu nhiên gần  $10^9$ , ví dụ như sau:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Khi sử dụng những hằng số như vậy, ta có thể sử dụng kiểu dữ liệu long long để tính giá trị băm, vì hai tích  $AB$  và  $BB$  sẽ nằm gọn trong giới hạn long long. Nhưng liệu có  $10^9$  giá trị băm khác nhau đã đủ hay chưa?

Chúng ta hãy cùng nhau xem xét ba trường hợp mà kỹ thuật băm có thể được sử dụng:

*Trường hợp 1:* Chuỗi  $x$  và  $y$  được so sánh với nhau. Xác suất va chạm là  $1/B$ , với giả sử khả năng xuất hiện của tất cả các giá trị băm là như nhau.

*Trường hợp 2:* Một chuỗi  $x$  được so sánh với các chuỗi  $y_1, y_2, \dots, y_n$ . Khả năng xuất hiện một hoặc nhiều lần va chạm là

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

*Trường hợp 3:* Tất cả các chuỗi  $x_1, x_2, \dots, x_n$  được so sánh với nhau. Xác suất xảy ra ít nhất một lần va chạm là

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Bảng sau đây cho thấy xác suất va chạm xảy ra khi  $n = 10^6$  với các giá trị khác nhau của  $B$ :

hằng số $B$	trường hợp 1	trường hợp 2	trường hợp 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 trên cho ta thấy trong trường hợp 1, khả năng xảy ra va chạm không đáng kể khi  $B \approx 10^9$ . Ở trường hợp 2, va chạm dù xảy ra nhưng xác suất vẫn nhỏ. Tuy nhiên, trong trường hợp 3 mọi thứ đã trở nên khác biệt: gần như chắc chắn sẽ có va chạm khi  $B \approx 10^9$ .

Hiện tượng xảy ra ở trường hợp 3 thường được biết đến bằng cái tên **ngịch lý ngày sinh nhật**: nếu có  $n$  người trong một phòng, khả năng có hai người có cùng ngày sinh nhật khá lớn dù  $n$  có nhỏ. Tương tự, với kỹ thuật băm, khi tất cả các giá trị băm được so sánh với nhau, khả năng cao là sẽ xuất hiện hai giá trị băm giống nhau.

Chúng ta có thể khiến xác suất va chạm xảy ra nhỏ hơn bằng cách tính toán *nhiều* giá trị băm với các hằng số modulo khác nhau. Lúc này, rất khó để xảy ra trường hợp va chạm đồng thời ở tất cả các giá trị băm của mỗi modulo. Ví dụ, việc sử dụng hai giá trị băm với tham số  $B \approx 10^9$  sẽ tương đương với một giá trị băm với tham số  $B \approx 10^{18}$ , khiến xác suất va chạm rất nhỏ.

Một số người sử dụng các hằng số  $B = 2^{32}$  và  $B = 2^{64}$ , điều này cũng khá tiện lợi vì các phép toán trên số nguyên 32 và 64 bit được xem như việc tính toán trên modulo  $2^{32}$  và  $2^{64}$ . Tuy nhiên, đây *không* phải là một lựa chọn đúng đắn, vì ta có thể cố ý dựng các giá trị đầu vào để luôn luôn xảy ra va chạm khi sử dụng các hằng số modulo có dạng  $2^x$  [51].

## 26.4 Thuật toán Z

**Mảng Z** của một chuỗi  $s$  độ dài  $n$  (ký hiệu là  $z$ ) lưu độ dài của chuỗi con dài nhất của  $s$  bắt đầu từ vị trí  $k$  và là tiền tố của  $s$  với mỗi  $k = 0, 1, \dots, n-1$ . Vì thế,  $z[k] = p$  cho ta biết  $s[0 \dots p-1] = s[k \dots k+p-1]$ . Rất nhiều bài toán xử lý chuỗi có thể được giải quyết hiệu quả bằng mảng Z.

Ví dụ, mảng Z của chuỗi ACBACDACBACBACDA có giá trị 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 ví dụ này,  $z[6] = 5$ , bởi vì chuỗi con ACBAC độ dài 5 là tiền tố của  $s$ , nhưng chuỗi con ACBACB độ dài 6 không phải là tiền tố của  $s$ .

### Mô tả thuật toán

Tiếp theo, ta sẽ mô tả **thuật toán Z**<sup>4</sup>, là một thuật toán xây dựng mảng Z một cách hiệu quả trong thời gian  $\mathcal{O}(n)$ . Thuật toán tính toán các giá trị của mảng Z từ trái sang phải bằng cách sử dụng cả thông tin đã lưu trữ trước đó trong mảng, lẫn kết quả so sánh các chuỗi con theo từng ký tự.

Để tính toán mảng Z một cách hiệu quả, thuật toán duy trì một đoạn  $[x, y]$  sao cho  $s[x \dots y]$  là một tiền tố của  $s$ , và  $y$  càng lớn càng tốt. Vì ta đã biết rằng hai chuỗi  $s[0 \dots y-x]$  và  $s[x \dots y]$  bằng nhau, nên chúng ta có thể tính các giá trị Z cho các vị trí  $x+1, x+2, \dots, y$ .

<sup>4</sup>Thuật toán Z được mô tả trong [32] là thuật toán đơn giản nhất cho việc so khớp mẫu trong thời gian tuyến tính, và ý tưởng ban đầu được cho là từ [50].

Với mỗi vị trí  $k$ , trước tiên ta kiểm tra giá trị của  $z[k-x]$ . Nếu  $k+z[k-x] < y$ , thì ta biết rằng  $z[k] = z[k-x]$ . Tuy nhiên, nếu  $k+z[k-x] \geq y$ , thì  $s[0 \dots y-k]$  bằng  $s[k \dots y]$ , và để xác định giá trị của  $z[k]$  ta vẫn phải tiếp tục lần lượt so sánh các ký tự nằm ở phía sau  $y$ . Mặc dù vậy, thuật toán vẫn chạy trong thời gian  $\mathcal{O}(n)$ , vì chúng ta bắt đầu so sánh từ vị trí  $y-k+1$  và  $y+1$ .

Ví dụ, chúng ta hãy cùng nhau 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 được giá trị  $z[6] = 5$ , khoảng  $[x, y]$  hiện tại là  $[6, 10]$ :

0	1	2	3	4	5	$x$		$y$		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	?	?	?	?	?	?	?	?	?


Bây giờ chúng ta có thể tính các giá trị tiếp theo trong mảng  $Z$  một cách hiệu quả, vì ta đã biết  $s[0 \dots 4]$  và  $s[6 \dots 10]$  bằng nhau. Đầu tiên, vì  $z[1] = z[2] = 0$ , nên ngay lập tức ta cũng biết rằng  $z[7] = z[8] = 0$ :

0	1	2	3	4	5	$x$		$y$		10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



Tiếp theo, vì  $z[3] = 2$ , nên ta có  $z[9] \geq 2$ :

0	1	2	3	4	5	$x$		$y$		10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



Tuy nhiên, chúng ta không có thông tin gì từ vị trí 10 trở về sau của xâu, nên chúng ta cần tiếp tục so sánh từng ký tự của xâu con.

						$x$			$y$						
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	?	?	?	?	?	?	?

Thì ra  $z[9] = 7$ , nên đoạn  $[x, y]$  mới là  $[9, 15]$ :

									$x$				$y$		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?

Lúc này, tất cả các giá trị còn lại của mảng  $Z$  đều có thể được tính bằng các thông tin đã lưu từ trước đó trong mảng:

										$x$		$y$			
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 $Z$

Việc chọn sử dụng băm xâu hay thuật toán  $Z$  thường là tùy vào sở thích của từng người. Không giống như kỹ thuật băm, thuật toán  $Z$  luôn hoạt động đúng và không có nguy cơ va chạm. Tuy nhiên, thuật toán  $Z$  khó cài đặt hơn, và một số bài tập chỉ có thể được giải bằng kỹ thuật băm.

Ta lấy lại ví dụ về bài toán so khớp mẫu một lần nữa, nhiệm vụ của chúng ta là tìm tất cả những lần xuất hiện của mẫu  $p$  trong xâu  $s$ . Chúng ta đã giải quyết bài toán này một cách hiệu quả với kỹ thuật băm xâu, tuy nhiên thuật toán  $Z$  cho ta một cách nữa để giải quyết bài toán.

Một ý tưởng thường dùng trong xử lý xâu là xây dựng một xâu mới bằng cách nối nhiều xâu lại với nhau, và ngăn cách bởi các ký tự đặc biệt. Trong bài toán này, chúng ta có thể xây dựng một xâu  $p\#s$ , với  $p$  và  $s$  được ngăn cách bởi ký tự  $\#$  không xuất hiện trong cả hai xâu. Mảng  $Z$  của  $p\#s$  cho chúng ta vị trí của những vị trí mà  $p$  xuất hiện trong  $s$ , bởi vì những vị trí đó có giá trị trong mảng bằng độ dài của  $p$ .

Ví dụ, nếu  $s = \text{HATTIVATTI}$  và  $p = \text{ATT}$ , mảng  $Z$  sẽ trông giống 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

Vị trí 5 và 10 chứa giá trị 3, điều đó có nghĩa là mẫu ATT xuất hiện ở hai vị trí tương ứng trong xâu HATTIVATTI.

Độ phức tạp thời gian của thuật toán vừa nêu trên là tuyến tính, vì ta chỉ cần xây dựng mảng Z và duyệt các giá trị trong đó.

## Cài đặt

Đây là một cách cài đặt hàm Z ngắn gọn, nó 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 chia căn

**Thuật toán chia căn** là một thuật toán mà độ phức tạp thời gian của nó có chứa căn bậc hai. Độ phức tạp  $O(\sqrt{n})$  tốt hơn  $O(n)$  nhưng tệ hơn  $O(\log n)$ . Nhưng chắc chắn rằng, nhiều thuật toán chia căn chạy nhanh và có tính thực tiễn cao.

Ví dụ, xét bài toán dựng một cấu trúc dữ liệu hỗ trợ hai thao tác sau trên mảng: thay đổi giá trị một phần tử tại vị trí bất kì và tính tổng các giá trị thuộc một đoạn con liên tiếp. Trong các chương trước, chúng ta đã giải quyết bài toán này bằng cây phân đoạn và cây chỉ số nhị phân, với độ phức tạp mỗi thao tác là  $O(\log n)$ . Tuy nhiên, bây giờ ta sẽ giải bài toán này bằng cách khác là sử dụng một cấu trúc "chia căn" cho phép chỉnh sửa phần tử trong  $O(1)$  và tính tổng trong  $O(\sqrt{n})$ .

Ý tưởng là chia mảng thành các khối có kích thước  $\sqrt{n}$  và mỗi khối lưu trữ tổng các phần tử trong nó. Ví dụ, một mảng có 16 phần tử sẽ được chia thành các khối chứa 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 cập nhật phần tử rất dễ, vì ta chỉ cần chỉnh sửa tổng của một khối duy nhất sau mỗi thao tác, điều này có thể thực hiện trong độ phức tạp  $O(1)$ . Ví dụ, hình dưới đây mô tả sự thay đổi giá trị một phần tử và của tổng của khối tương ứng chứa nó:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Để tính tổng các phần tử trong một đoạn, chúng ta chia tổng đó thành ba phần, gồm một vài phần tử riêng biệt (nằm ở hai đầu) và xen giữa chúng là các khối:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Bởi vì số lượng phần tử riêng lẻ là  $O(\sqrt{n})$  và số lượng khối cũng là  $O(\sqrt{n})$ , nên truy vấn tính tổng mất thời gian là  $O(\sqrt{n})$ . Việc đặt kích thước khối là  $\sqrt{n}$  giúp *cân bằng* hai yếu tố: mảng được chia thành  $\sqrt{n}$  khối, mỗi khối chứa  $\sqrt{n}$  phần tử (Hai yếu tố số lượng khối và số phần tử trong mỗi khối đã được cân bằng).

Trên thực tế, không cần lấy chính xác giá trị  $\sqrt{n}$  làm tham số, thay vào đó ta có thể dùng tham số  $k$  và  $n/k$  trong đó  $k$  có thể khác với  $\sqrt{n}$ . Tham số tối ưu phụ thuộc vào bài toán và dữ liệu đầu vào. Ví dụ, nếu thuật toán thường xuyên duyệt qua các khối nhưng hiếm khi kiểm tra từng giá trị riêng biệt trong đó, có thể sẽ tốt hơn nếu ta chia mảng thành  $k < \sqrt{n}$  khối, mỗi khối chứa  $n/k > \sqrt{n}$  phần tử.

## 27.1 Kết hợp thuật toán

Trong phần này chúng ta sẽ thảo luận về hai thuật toán chia căn, mỗi thuật toán này đều được xây dựng dựa trên việc kết hợp hai thuật toán thành một. Trong cả hai trường hợp được giới thiệu dưới đây, nếu chỉ sử dụng riêng lẻ một thuật toán mà không kết hợp với thuật toán còn lại, ta có thể giải bài toán với độ phức tạp  $O(n^2)$ . Tuy nhiên, bằng việc kết hợp chúng, độ phức tạp chỉ còn  $O(n\sqrt{n})$ .

### Xử lý theo trường hợp

Giả sử ta có một lưới hai chiều bao gồm  $n$  ô. Mỗi ô được gán một kí tự, và nhiệm vụ của chúng ta là phải tìm hai ô có cùng kí tự mà khoảng cách giữa chúng nhỏ nhất, ở đây khoảng cách giữa hai ô  $(x_1, y_1)$  và  $(x_2, y_2)$  là  $|x_1 - x_2| + |y_1 - y_2|$ . Ví dụ, xét bảng sau:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

Ở đây, khoảng cách bé nhất là 2 giữa hai ô chứa kí tự 'E'.

Chúng ta có thể giải quyết vấn đề này bằng cách xét riêng từng kí tự. Với cách tiếp cận này, vấn đề mới bây giờ là tính khoảng cách tối thiểu giữa hai ô chứa kí tự  $c$  *cố định*. Xét hai thuật toán:

**Thuật toán 1:** Duyệt qua mọi cặp ô chứa kí tự  $c$ , và tính khoảng cách tối thiểu của chúng. Cách này sẽ tốn độ phức tạp là  $O(k^2)$  trong đó  $k$  là số lượng ô chứa kí tự  $c$ .

**Thuật toán 2:** Sử dụng thuật toán duyệt theo chiều rộng bắt đầu đồng thời tại mọi ô chứa kí tự  $c$ . Khoảng cách tối thiểu giữa hai ô chứa kí tự  $c$  sẽ tính được trong độ phức tạp  $O(n)$ .

Một cách để giải bài toán là chọn một trong hai thuật toán và áp dụng nó cho mọi chữ cái. Nếu ta dùng thuật toán thứ nhất, độ phức tạp là  $O(n^2)$ , bởi vì có trường hợp tất cả mọi ô đều chứa cùng một chữ cái, khi đó  $k = n$ . Tương tự nếu ta dùng thuật toán thứ hai, độ phức tạp cũng là  $O(n^2)$ , bởi vì có thể tất cả mọi ô đều chứa kí tự khác nhau, khi đó ta sẽ cần tới  $n$  lần duyệt BFS.

Tuy nhiên, ta có thể *kết hợp* hai thuật toán và với mỗi kí tự ta sẽ lựa chọn thuật toán phù hợp để áp dụng tùy thuộc vào số lần xuất hiện kí tự đó trên bảng. Giả sử rằng kí tự  $c$  xuất hiện  $k$  lần. Nếu  $k \leq \sqrt{n}$ , ta dùng thuật toán 1, ngược lại nếu  $k > \sqrt{n}$ , ta dùng thuật toán 2. Bằng cách làm này, tổng thời gian chạy của thuật toán chỉ rơi vào  $O(n\sqrt{n})$ .

Đầu tiên, giả sử rằng ta dùng thuật toán 1 cho kí tự  $c$ . Vì  $c$  xuất hiện nhiều nhất  $\sqrt{n}$  lần trên bảng, ta so sánh mỗi ô chứa kí tự  $c$  tối đa  $O(\sqrt{n})$  lần với các ô khác. Do đó, độ phức tạp dùng cho việc tính toán tất cả ô như vậy là  $O(n\sqrt{n})$ . Trường hợp khác, giả sử rằng ta dùng thuật toán 2 cho kí tự  $c$ . Vì có nhiều nhất là  $\sqrt{n}$  kí tự như vậy, nên việc tính toán cho tất cả kí tự này cũng mất độ phức tạp  $O(n\sqrt{n})$ .

## Xử lí hàng loạt

Bài toán tiếp theo mà ta xem xét cũng liên quan tới một bảng 2 chiều gồm  $n$  ô. Ban đầu, tất cả mọi ô đều là màu trắng, trừ một ô. Ta thực hiện  $n - 1$  thao tác, mỗi thao tác trước tiên sẽ yêu cầu tính khoảng cách tối thiểu từ một ô trắng cho trước tới một ô đen, và sau đó tô màu đen cho ô trắng này.

Ví dụ, xem xét thao tác dưới đây:

		*	

Đầu tiên, ta tính toán khoảng cách tối thiểu từ ô trắng được đánh dấu \* tới một ô đen. Khoảng cách tối thiểu là 2, bởi vì có một ô đen cách về bên trái 2 bước. Tiếp đến, ta tô màu ô đó thành đen:


Xem xét hai thuật toán dưới đây:

*Thuật toán 1:* Dùng duyệt theo chiều sâu để tính cho mỗi ô trắng khoảng cách tới ô đen gần nó nhất. Điều này tốn độ phức tạp  $O(n)$ , và sau khi duyệt, ta có thể biết khoảng cách tối thiểu từ một ô trắng bất kì tới một ô đen trong độ phức tạp  $O(1)$ .

*Thuật toán 2:* Duy trì danh sách của các ô đã được tô đen, tại mỗi thao tác ta duyệt qua toàn bộ danh sách này rồi sau đó thêm ô đen mới vào danh sách. Mỗi thao tác sẽ mất độ phức tạp  $O(k)$  trong đó  $k$  là độ dài danh sách.

Chúng ta hợp nhất hai thuật toán trên bằng cách chia các thao tác thành  $O(\sqrt{n})$  nhóm, mỗi cái bao gồm  $O(\sqrt{n})$  thao tác. Vào lúc bắt đầu mỗi nhóm, ta thực hiện thuật toán 1. Sau đó, ta dùng thuật toán 2 để tính toán cho mỗi thao tác trong nhóm. Ta sẽ xóa danh sách tạo bởi thuật toán 2 mỗi khi xử lý xong một nhóm. Trong mỗi thao tác, khoảng cách tối thiểu tới một ô đen phải là kết quả của thuật toán 1 hoặc là kết quả của thuật toán 2.

Thuật toán kết hợp có độ phức tạp  $O(n\sqrt{n})$ . Lí do thứ nhất là, thuật toán 1 được thực thi  $O(\sqrt{n})$  lần, và mỗi lần như vậy tốn thời gian  $O(n)$ . Thứ hai, khi dùng thuật toán 2 trong mỗi nhóm, danh sách chỉ bao gồm  $O(\sqrt{n})$  ô (bởi vì ta xóa danh sách sau khi xử lý xong mỗi nhóm) do đó mỗi thao tác sẽ chỉ tốn thời gian  $O(\sqrt{n})$ .

## 27.2 Tổng các số nguyên

Một vài thuật toán chia căn được xây dựng dựa trên quan sát: nếu một số nguyên dương  $n$  được biểu diễn thành tổng của các số nguyên dương, tổng như thế sẽ chứa nhiều nhất  $O(\sqrt{n})$  số *phân biệt*. Lý do cho điều này là để xây dựng một tổng chứa nhiều giá trị khác nhau nhất, ta nên chọn những số *nhỏ*. Nếu ta chọn những số  $1, 2, \dots, k$ , tổng tạo được sẽ là

$$\frac{k(k+1)}{2}.$$

Do đó, số lượng giá trị phân biệt tối đa là  $k = O(\sqrt{n})$ . Tiếp sau đây, chúng ta sẽ thảo luận hai bài toán có thể giải được hiệu quả dựa vào quan sát này.

### Bài toán cái túi

Giả sử ta được cho một danh sách các vật có khối lượng nguyên, mà tổng cân nặng là  $n$ . Nhiệm vụ là tìm ra tất cả tổng khối lượng có thể tạo được từ một tập con của những vật trên. Ví dụ, nếu danh sách là  $\{1, 3, 3\}$ , các tổng có thể tạo được là:

- 0 (tập hợp rỗng)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Sử dụng cách tiếp cận của bài toán cái túi (xem Chương 7.4), vấn đề có thể được giải quyết như sau: ta định nghĩa hàm  $\text{possible}(x, k)$  có giá trị là 1 nếu tổng  $x$  có thể tạo được chỉ từ  $k$  vật đầu tiên, và 0 nếu ngược lại. Bởi vì

tổng cân nặng là  $n$ , có nhiều nhất  $n$  vật do đó mọi giá trị của hàm có thể tính được trong thời gian  $O(n^2)$  bằng quy hoạch động.

Tuy nhiên, ta có thể làm thuật toán hiệu quả hơn bằng cách tận dụng thực tế rằng có nhiều nhất  $O(\sqrt{n})$  cân nặng *khác nhau*. Vì vậy, ta có thể xử lý theo từng nhóm, mỗi nhóm chứa các vật có khối lượng bằng nhau. Ta có thể xử lý mỗi nhóm trong  $O(n)$ , từ đó cho ra thuật toán có độ phức tạp  $O(n\sqrt{n})$ .

Ý tưởng là sử dụng một mảng lưu lại tổng các trọng số có thể tạo được từ các nhóm đã được xử lý trước đó. Mảng chứa  $n$  phần tử: phần tử  $k$  là 1 nếu tổng  $k$  có thể tạo được và 0 nếu ngược lại. Để xử lý một nhóm, chúng ta quét mảng từ trái sang phải và ghi nhận những tổng trọng số mới có thể tạo được bằng việc kết hợp nhóm này với các nhóm trước đó.

## Xây dựng xâu

Cho một xâu  $s$  có độ dài  $n$  và một tập hợp các xâu  $D$  có tổng độ dài là  $m$ , xét bài toán đếm số cách tạo nên  $s$  từ việc nối các xâu trong  $D$ . Ví dụ, nếu  $s = \text{ABAB}$  và  $D = \{A, B, AB\}$ , ta có 4 cách:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Chúng ta có thể giải bài toán này bằng quy hoạch động: Đặt  $\text{count}(k)$  là số cách để xây dựng tiền tố  $s[0 \dots k]$  từ các xâu trong  $D$ . Lúc này  $\text{count}(n-1)$  cho ta biết đáp án bài toán, và ta có thể giải nó trong  $O(n^2)$  sử dụng cấu trúc trie.

Tuy nhiên, ta có thể giải quyết một cách hiệu quả hơn sử dụng kỹ thuật băm chuỗi (hash) và tận dụng thực tế rằng có nhiều nhất  $O(\sqrt{m})$  độ dài xâu khác nhau trong  $D$ . Đầu tiên, ta xây dựng một tập  $H$  bao gồm tất cả giá trị hash của các xâu trong  $D$ . Sau đó, khi tính giá trị của  $\text{count}(k)$ , ta duyệt qua tất cả giá trị  $p$  sao cho tồn tại một xâu có độ dài  $p$  trong  $D$ , tính toán giá trị hash của  $s[k-p+1 \dots k]$  và kiểm tra xem nó có thuộc  $H$  không. Bởi vì có nhiều nhất  $O(\sqrt{m})$  độ dài xâu khác nhau, cách làm này dẫn tới một thuật toán có độ phức tạp là  $O(n\sqrt{m})$ .

## 27.3 Thuật toán Mo

**Thuật toán Mo**<sup>1</sup> có thể được áp dụng trong những bài toán yêu cầu xử lý các truy vấn trên đoạn con trong một mảng *tĩnh*, tức là, giá trị mảng không đổi giữa các truy vấn. Trong mỗi truy vấn, ta được cho một khoảng  $[a, b]$ , và cần phải tính toán một giá trị nào đó dựa trên các phần tử nằm trong đoạn

<sup>1</sup>Theo [12], thuật toán này được đặt tên theo Mo Tao, một lập trình viên thi đấu người Trung Quốc, tuy nhiên kỹ thuật này đã xuất hiện từ trước trong các văn bản [44].

từ vị trí  $a$  tới  $b$  này. Bởi vì mảng tĩnh, các truy vấn có thể được tính toán theo bất kì thứ tự nào. Thuật toán Mo xử lý các truy vấn theo thứ tự đặc biệt, thứ tự này đảm bảo thuật toán sẽ hoạt động hiệu quả.

Thuật toán Mo duy trì một *phạm vi hoạt động* của mảng, và tại mỗi thời điểm ta biết được câu trả lời cho truy vấn ứng với phạm vi hoạt động này. Thuật toán xử lí lần lượt từng truy vấn, và luôn di chuyển hai đầu mút của phạm vi hoạt động bằng cách chèn thêm và loại bớt các phần tử. Độ phức tạp thời gian của thuật toán là  $O(n\sqrt{n}f(n))$  trong đó  $n$  là kích thước mảng, có  $n$  truy vấn và độ phức tạp mỗi thao tác chèn và xóa một phần tử là  $O(f(n))$ .

Thủ thuật trong thuật toán Mo chính là thứ tự xử lý các truy vấn: Mảng được chia thành các khối gồm  $k = O(\sqrt{n})$  phần tử, và truy vấn  $[a_1, b_1]$  được tính toán trước truy vấn  $[a_2, b_2]$  khi

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$  hoặc
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$  và  $b_1 < b_2$ .

Do đó, tất cả các truy vấn có đầu mút trái nằm trong cùng một khối được xử lý lần lượt theo thứ tự tăng dần của đầu mút bên phải. Với thứ tự này, thuật toán chỉ thực hiện  $O(n\sqrt{n})$  thao tác, bởi vì đầu mút trái di chuyển  $O(n)$  lần với mỗi lần đi  $O(\sqrt{n})$  bước, và đầu mút phải di chuyển  $O(\sqrt{n})$  lần với mỗi lần đi  $O(n)$  bước. Do đó, chúng di chuyển tổng cộng  $O(n\sqrt{n})$  lần xuyên suốt thuật toán.

## Ví dụ

Ví dụ, xét bài toán trong đó ta được cho trước một tập các truy vấn, mỗi truy vấn tương ứng với một khoảng trong mảng. Nhiệm vụ của ta là với mỗi truy vấn, đếm số lượng giá trị *khác nhau* trong khoảng đó.

Trong thuật toán Mo, các truy vấn luôn được sắp xếp theo cách giống nhau, nhưng phụ thuộc vào bài toán mà ta có cách lưu trữ khác nhau để tìm ra câu trả lời cho mỗi truy vấn. Trong bài toán này, ta có thể duy trì một mảng count trong đó  $\text{count}[x]$  cho biết số lần giá trị  $x$  xuất hiện trong phạm vi hoạt động.

Khi ta chuyển từ truy vấn này sang truy vấn khác, phạm vi hoạt động sẽ 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: đầu mút trái di chuyển một bước sang phải, và đầu mút phải di chuyển hai bước sang phải.

Sau mỗi bước, mảng count cần được cập nhật. Sau khi thêm vào giá trị  $x$ , ta tăng giá trị  $\text{count}[x]$  lên 1, và nếu sau đó  $\text{count}[x] = 1$ , ta sẽ tăng đáp

án cho truy vấn hiện tại thêm 1. Tương tự, sau khi xóa giá trị  $x$ , ta giảm giá trị  $\text{count}[x]$  đi 1, và nếu sau đó  $\text{count}[x] = 0$ , ta giảm đáp án hiện tại đi 1.

Trong bài toán này, thời gian để thực hiện mỗi bước là  $O(1)$ , cho nên độ phức tạp tổng thể của thuật toán là  $O(n\sqrt{n})$ .





# Chương 28

## Cây phân đoạn

Cây phân đoạn là một cấu trúc dữ liệu linh hoạt có thể được dùng để giải quyết rất nhiều bài toán. Tuy nhiên, vẫn còn nhiều chủ đề liên quan đến nó mà ta chưa bàn tới. Trong chương này, ta sẽ xem xét một số biến thể nâng cao của cây phân đoạn.

Cho đến nay, chúng ta đã triển khai các thao tác trên cây phân đoạn bằng cách *đi từ dưới lên trên* cây. Ví dụ, ta đã tính tổng của một đoạn như sau (Chương 9.3):

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Tuy nhiên, trong những cây phân đoạn nâng cao hơn, thường ta sẽ cần triển khai các thao tác theo cách khác, theo thứ tự *đi từ trên xuống dưới*. Theo hướng tiếp cận này, hàm trên sẽ được cài đặt như sau:

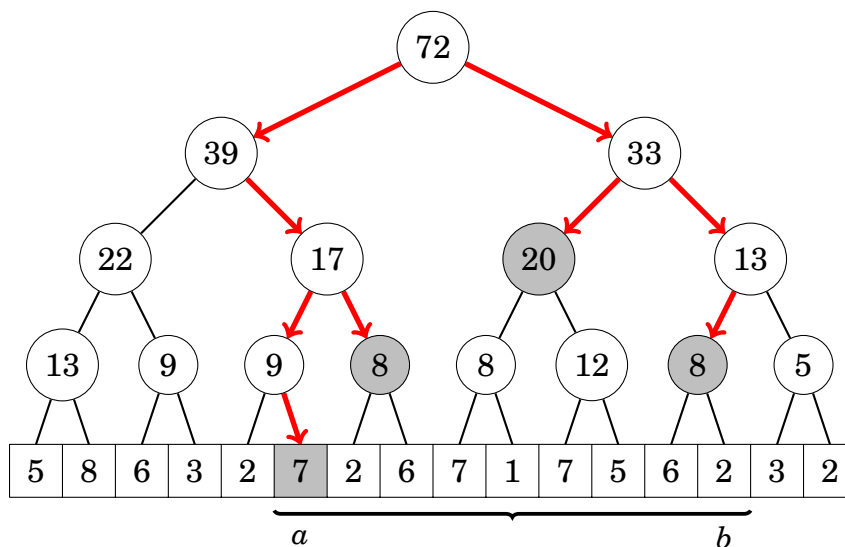
```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Lúc này, ta có thể tính bất kỳ giá trị  $\text{sum}_q(a,b)$  (tổng các số thuộc đoạn con  $[a,b]$  của mảng) như sau:

```
int s = sum(a, b, 1, 0, n-1);
```

Tham số  $k$  cho biết vị trí nút hiện tại trong tree (tên mảng lưu cây phân đoạn). Ban đầu  $k$  bằng 1, bởi vì chúng ta bắt đầu từ nút gốc của cây. Đoạn  $[x, y]$  tương ứng với nút  $k$  và lúc đầu bằng  $[0, n - 1]$ . Khi tính toán tổng, nếu đoạn  $[x, y]$  nằm ngoài đoạn  $[a, b]$ , thì tổng là 0, và nếu đoạn  $[x, y]$  nằm hoàn toàn bên trong đoạn  $[a, b]$ , tổng có thể lấy được từ tree. Nếu đoạn  $[x, y]$  có một phần giao với đoạn  $[a, b]$ , ta tiếp tục tìm kiếm một cách đệ quy trong nửa trái và nửa phải của đoạn  $[x, y]$ . Nửa trái là đoạn  $[x, d]$  và nửa phải là đoạn  $[d + 1, y]$  với  $d = \lfloor \frac{x+y}{2} \rfloor$ .

Hình sau thể hiện quá trình tìm kiếm trên cây khi tính giá trị  $\text{sum}_q(a, b)$ . Nút màu xám là những nút mà tại đó đệ quy dừng lại, không gọi xuống sâu hơn nữa vì có thể lấy trực tiếp tổng từ tree.



Trong cách cài đặt này, các truy vấn sẽ mất độ phức tạp thời gian là  $O(\log n)$  vì tổng số lượng nút được thăm là  $O(\log n)$

## 28.1 Lan truyền thông minh

Sử dụng kỹ thuật **lan truyền thông minh** (thuật ngữ gốc: **lazy propagation**), ta có thể dựng một cây phân đoạn hỗ trợ *cả hai* thao tác cập nhật đoạn con và truy vấn trên đoạn con trong độ phức tạp thời gian  $O(\log n)$ . Ý tưởng là thực hiện các thao tác cập nhật và truy vấn theo thứ tự từ trên xuống và với các thao tác cập nhật, ta xử lý *một cách "lười biếng"*, tức chỉ lan truyền thông tin xuống các nút con chỉ khi thật sự cần thiết.

Trong một cây phân đoạn "lười", các nút của cây chứa hai loại thông tin. Như trong một cây phân đoạn thông thường, các nút chứa tổng hoặc một số các giá trị khác liên quan đến đoạn con tương ứng. Ngoài ra, nút có thể chứa thông tin liên quan đến những cập nhật "lười", tức những cập nhật chưa được lan truyền xuống các nút con của nó.

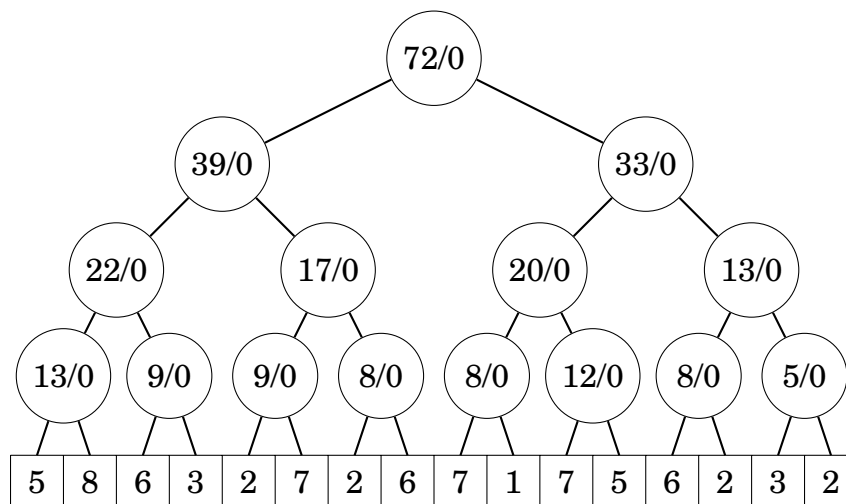
Có hai loại cập nhật đoạn: mỗi giá trị có thể được *tăng* thêm một vài đơn vị hoặc được *gán* một giá trị nào đó. Cả hai loại cập nhật đều có thể được

triển khai bằng những ý tưởng tương tự nhau. Thậm chí ta có thể dựng một cây hỗ trợ cùng một lúc cả hai loại thao tác này.

## Cây lưới

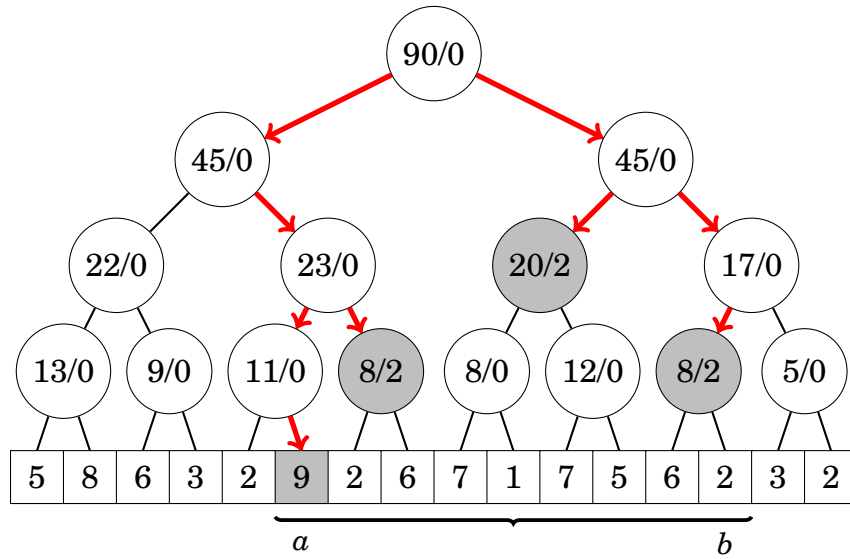
Ví dụ, ta cần phải tạo ra một cây phân đoạn có thể hỗ trợ hai thao tác: tăng từng phần tử trong đoạn  $[a, b]$  lên cùng một giá trị (hằng), và tính tổng các số trong đoạn  $[a, b]$ .

Ta sẽ dựng một cây, mà trong đó mỗi nút chứa hai giá trị  $s/z$ :  $s$  ký hiệu cho tổng các giá trị trong đoạn, và  $z$  là giá trị của một thao tác cập nhật lưới, nghĩa là các giá trị trong đoạn cần được tăng lên  $z$ . Trong cây sau,  $z = 0$  trong tất cả các nút, tức đang không có cập nhật lưới nào.



Khi mà các phần tử trong đoạn  $[a, b]$  được tăng lên  $u$  ta đi từ nút gốc tới các nút lá và thay đổi các nút trong cây như sau: Nếu đoạn  $[x, y]$  của nút nằm hoàn toàn trong đoạn  $[a, b]$ , ta tăng giá trị  $z$  của nút lên  $u$  và dừng lại. Nếu đoạn  $[x, y]$  chỉ có một phần thuộc vào đoạn  $[a, b]$ , ta tăng giá trị  $s$  lên  $hu$ , trong đó  $h$  là độ dài phần giao nhau giữa hai đoạn  $[a, b]$  và  $[x, y]$ , và gọi đệ quy để tiếp tục đi xuống cây.

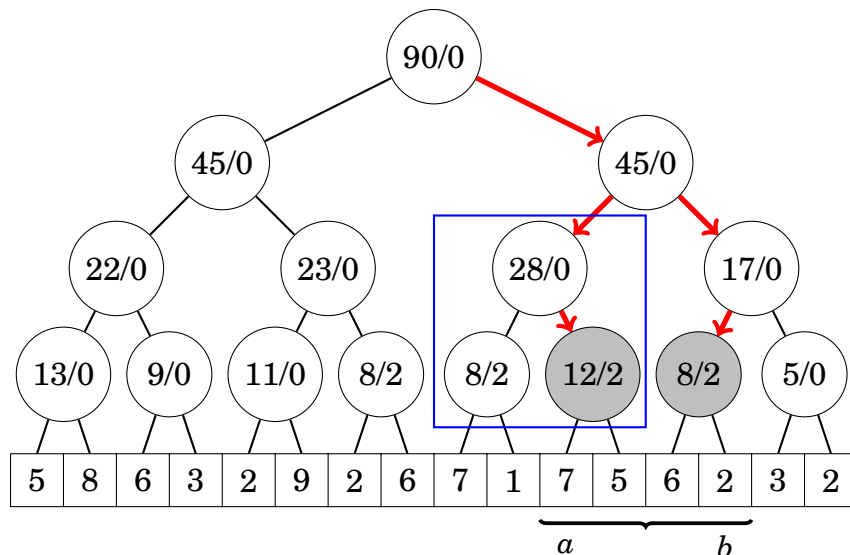
Ví dụ, hình sau biểu diễn một cây sau khi tăng các phần tử trong đoạn  $[a, b]$  lên 2:



Ta cũng có thể tính tổng của các phần tử trong đoạn  $[a, b]$  bằng cách đi từ trên xuống dưới trong cây. Nếu đoạn  $[x, y]$  của nút nằm hoàn toàn trong đoạn  $[a, b]$ , ta thêm giá trị  $s$  của nút vào tổng. Còn không, ta tiếp tục tìm kiếm bằng cách đệ quy xuống dưới cây.

Trong cả hai thao tác cập nhật và truy vấn, giá trị của thao tác cập nhật lười luôn được truyền đến các nút con trước khi tính toán, xử lý nút đó. Ý tưởng đằng sau việc này là thao tác cập nhật sẽ được truyền xuống chỉ khi thật sự cần thiết, giúp đảm bảo hiệu quả.

Ảnh sau cho thấy sự thay đổi của cây khi ta tính toán giá trị  $\text{sum}_a(a, b)$ . Các nút được bọc trong hình chữ nhật là các nút mà giá trị của nó bị thay đổi do thao tác cập nhật lười đã truyền thông tin đi xuống dưới.



Chú ý rằng đôi lúc ta cần kết hợp nhiều thao tác cập nhật lười. Vấn đề này xảy ra khi một nút đã sẵn có một cập nhật lười, lại được gán thêm một cập nhật lười khác. Trong trường hợp tính tổng, việc kết hợp rất dễ, vì sự kết hợp của hai thao tác  $z_1$  và  $z_2$  tương đương với một cập nhật  $z_1 + z_2$ .

## Cập nhật đa thức

Ta có thể tổng quát hóa các thao tác cập nhật lười để cập nhật đoạn theo các đa thức có dạng:

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

Trong trường hợp này, giá trị cập nhật của giá trị tại vị trí thứ  $i$  trong đoạn  $[a, b]$  là  $p(i - a)$ . Ví dụ như, cộng đa thức  $p(u) = u + 1$  vào đoạn  $[a, b]$  nghĩa là giá trị ở vị trí  $a$  tăng lên 1, giá trị ở vị trí  $a + 1$  tăng lên 2, và cứ như vậy.

Để hỗ trợ các thao tác cập nhật đa thức, mỗi nút được gán  $k + 2$  giá trị với  $k$  bằng bậc của đa thức. Giá trị  $s$  là tổng của các phần tử trong đoạn, và các giá trị  $z_0, z_1, \dots, z_k$  là hệ số của đa thức tương ứng với thao tác cập nhật lười.

Lúc này, tổng của các giá trị trong đoạn  $[x, y]$  sẽ bằng

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

Giá trị của tổng như trên có thể tính được một cách hiệu quả bằng cách sử dụng các công thức tổng. Ví dụ, hạng tử  $z_0$  tương ứng với tổng  $(y - x + 1)z_0$  và hạng tử  $z_1 u$  tương ứng với tổng

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

Khi lan truyền một thao tác cập nhật đi xuống trong cây, các chỉ số của  $p(u)$  thay đổi, vì trong mỗi đoạn  $[x, y]$  khác nhau, ta lại đi tính giá trị đa thức  $p(u)$  tại các điểm  $u = 0, 1, \dots, y - x$ . Tuy nhiên, ta vẫn có thể khắc phục được điều này, bởi vì  $p'(u) = p(u + h)$  là một đa thức ngang bậc với  $p(u)$ . Ta sẽ "tính tiền" các giá trị  $p(0), p(1), \dots, p(y - x)$  của nút con bên phải thành các giá trị  $p(h), p(h + 1), \dots, p(h + y - x)$ . Ví dụ như, 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 có tính "tĩnh", tức là mỗi nút có một vị trí cố định trong mảng và cây luôn cần một lượng bộ nhớ cố định. Trong một **cây phân đoạn động**, bộ nhớ chỉ được cấp phát cho các nút thực sự được truy cập trong thuật toán và điều này có thể tiết kiệm được một lượng lớn bộ nhớ.

Các nút của một cây phân đoạn động có thể được biểu diễn bằng struct sau:

```
struct node {
    int value;
    int x, y;
```

```
node *left, *right;
node(int v, int x, int y): value(v), x(x), y(y) {}
};
```

Ở đây value là giá trị của nút,  $[x,y]$  là đoạn tương ứng mà nút đó quản lý, left và right trỏ vào nút trái và nút phải của cây con.

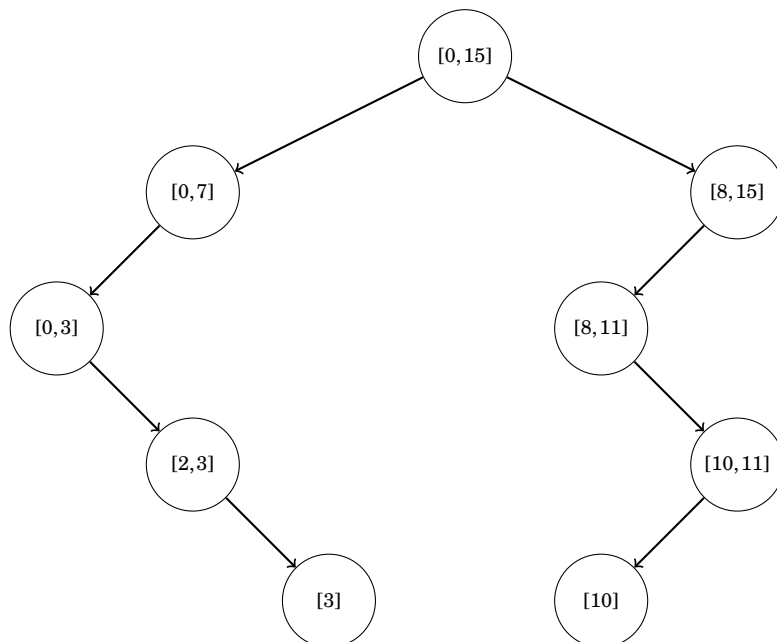
Sau đó, các nút có thể được tạo như sau:

```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

## Cây thưa

Một cây phân đoạn động rất hữu dụng khi mảng lưu trữ nó *thưa thớt*, tức là đoạn  $[0, n-1]$  của các chỉ số được cho phép tuy lớn, nhưng phần lớn giá trị của mảng vẫn bằng 0. Trong khi một cây phân đoạn thông thường sử dụng  $O(n)$  bộ nhớ, thì một cây phân đoạn động chỉ sử dụng  $O(k \log n)$  bộ nhớ với  $k$  là tổng số thao tác được thực hiện.

Một **cây phân đoạn thưa** ban đầu chỉ có một nút  $[0, n-1]$  mà giá trị của nó bằng 0, hay tất cả các giá trị trong mảng đều bằng 0. Sau khi cập nhật, các nút mới sẽ được thêm vào cây một cách linh động. Ví dụ, nếu  $n = 16$  và các phần tử ở vị trí 3 và 10 đã được sửa đổi thì cây sẽ chứa các nút như sau:



Bất kỳ đường đi nào từ nút gốc tới nút lá đều chứa  $O(\log n)$  nút nên với mỗi thao tác ta tạo thêm nhiều nhất là  $O(\log n)$  nút mới cho cây. Vì thế, sau  $k$  thao tác, cây sẽ có nhiều nhất  $O(k \log n)$  nút.

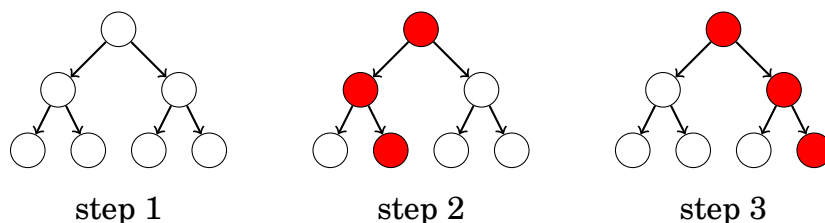
Lưu ý rằng nếu chúng ta biết trước tất cả các phần tử cần được cập nhật ngay lúc bắt đầu thuật toán, không cần thiết sử dụng một cây phân đoạn động, bởi vì ta có thể dùng một cây thông thường kết hợp với phương pháp nén chỉ số (Chương 9.4). Tuy nhiên, cách này không khả thi khi các chỉ số được tạo mới trong suốt quá trình chạy thuật toán.

## Cây phiên bản

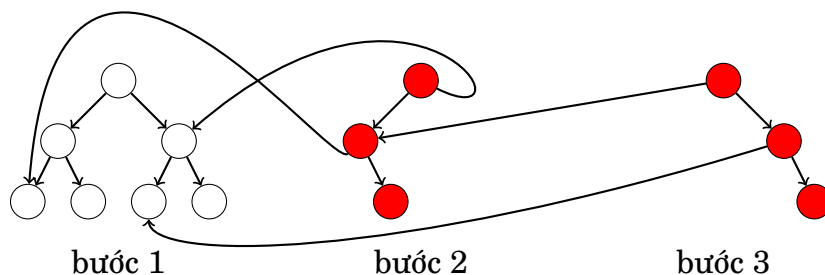
Áp dụng cách cài đặt cây động như trên, ta cũng có thể tạo ra một **cây phiên bản** (thuật ngữ gốc là **persistent segment tree**) lưu lại *lịch sử thay đổi* của một cây. Trong cách cài đặt này, ta có thể truy cập vào tất cả phiên bản của cây từng tồn tại trong suốt thuật toán một cách hiệu quả.

Khi đã có lịch sử thay đổi, chúng ta có thể thực hiện các truy vấn trong bất kỳ cây nào tồn tại trước đó như trong một cây phân đoạn thông thường, bởi vì toàn bộ cấu trúc của mỗi cây đều được lưu lại. Chúng ta cũng có thể tạo ra các cây mới dựa trên các phiên bản trước của cây và sửa đổi chúng một cách độc lập.

Xét các cập nhật sau đây, trong đó các nút màu đỏ thay đổi và các nút khác vẫn giữ nguyên:



Sau mỗi thao tác cập nhật, phần lớn các nút của cây giữ nguyên. Vì thế, một cách hiệu quả để lưu trữ lịch sử chỉnh sửa là biểu diễn mỗi phiên bản của cây dưới dạng tổ hợp giữa những nút mới và những cây con từ phiên bản trước đó. Trong ví dụ này, lịch sử chỉnh sửa có thể được lưu trữ như sau:



Cấu trúc của bất kỳ cây nào trước đó có thể được dựng lại bằng cách đi lần theo con trỏ tương ứng với nút gốc tại phiên bản đó. Vì mỗi thao tác chỉ thêm  $O(\log n)$  nút mới vào cây, ta có thể lưu trữ toàn bộ lịch sử chỉnh sửa của cây.

## 28.3 Kết hợp với Cấu trúc dữ liệu khác

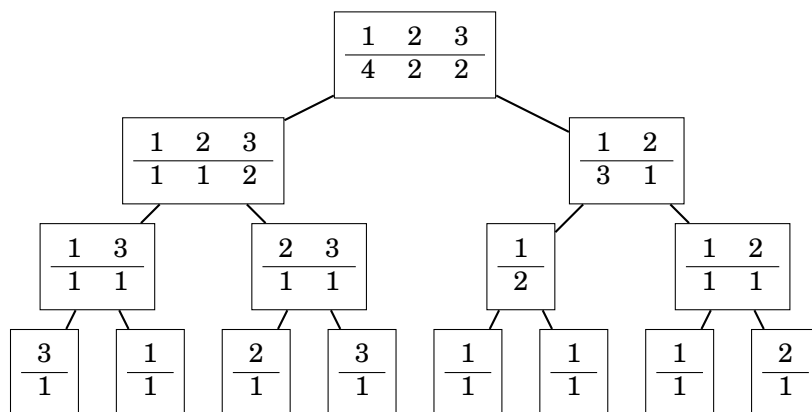
Thay vì chứa các giá trị đơn lẻ, các nút trong một cây phân đoạn còn có thể chứa các cấu trúc dữ liệu duy trì thông tin về đoạn tương ứng mà nó quản lí. Trong một cây như vậy, các thao tác thường có độ phức tạp  $O(f(n)\log n)$  với  $f(n)$  là thời gian cần thiết để xử lý mỗi nút trong một thao tác.

Ví dụ, xét một cây phân đoạn hỗ trợ các truy vấn có dạng "Phần tử  $x$  xuất hiện bao nhiêu lần trong đoạn  $[a, b]$ ?" Ví dụ, phần tử 1 xuất hiện 3 lần trong đoạn sau:

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

Để hỗ trợ các truy vấn như vậy, ta dựng một cây phân đoạn mà trong đó mỗi nút được gán một cấu trúc dữ liệu có thể cho biết số lần xuất hiện của bất kì phần tử  $x$  nào trong đoạn nó quản lí. Sử dụng cây này, có thể tính được câu trả lời cho mỗi truy vấn bằng cách tổng hợp kết quả của các nút nằm trong đoạn truy vấn.

Ví dụ, cây phân đoạn sau tương ứng với mảng trên:



Ta có thể dựng cây mà mỗi nút của nó chứa một cấu trúc map (ánh xạ). Trong trường hợp này, thời gian cần thiết để xử lý từng nút là  $O(\log n)$ , nên tổng độ phức tạp thời gian của một truy vấn là  $(\log^2 n)$ . Cây sử dụng  $O(n \log n)$  bộ nhớ, bởi vì có tất cả  $O(\log n)$  tầng và mỗi tầng chứa  $O(n)$  phần tử.

## 28.4 Cây hai chiều

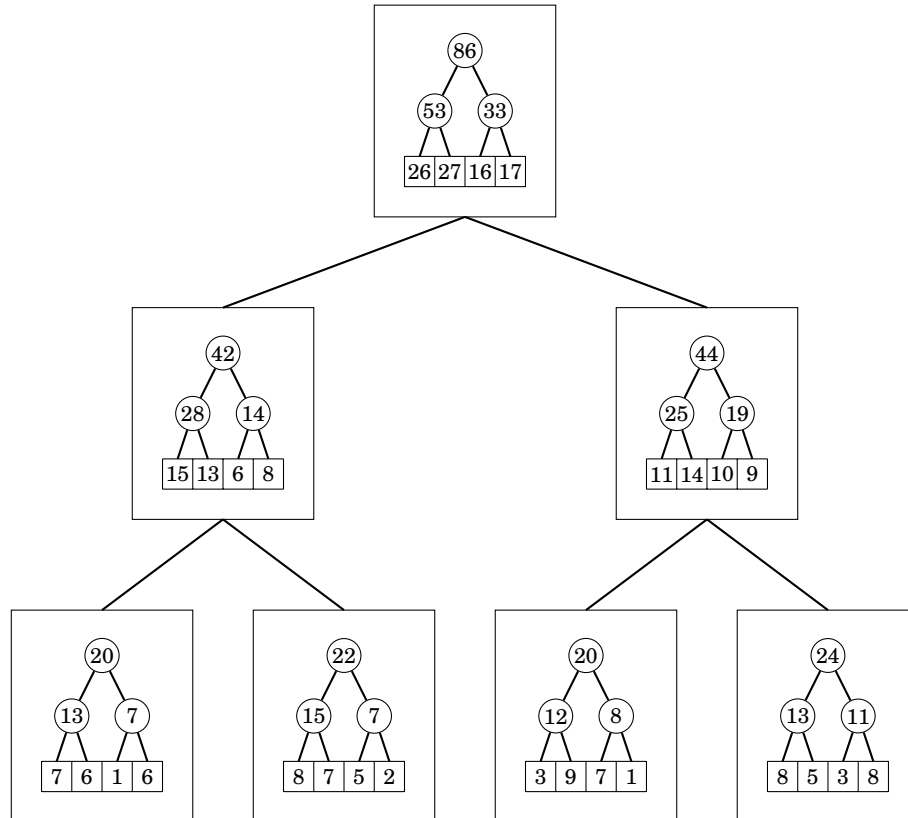
Một cây **cây phân đoạn hai chiều** hỗ trợ các truy vấn liên quan đến bảng con hình chữ nhật của một mảng hai chiều. Một cây như vậy có thể được cài đặt theo kiểu cây phân đoạn bao chứa nhau: một cây lớn quản lý các hàng của mảng, và mỗi nút chứa một cây nhỏ quản lý các cột.

Ví dụ, trong mảng sau



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

Tổng của bất kì bảng con nào có thể được tính bằng cây phân đoạn sau:



Các thao tác trong một cây phân đoạn hai chiều tốn  $O(\log^2 n)$  thời gian, bởi vì cây lớn và mỗi cây nhỏ bao gồm  $O(\log n)$  tầng. Cây cần  $O(n^2)$  bộ nhớ, bởi vì mỗi cây nhỏ chứa  $O(n)$  giá trị.

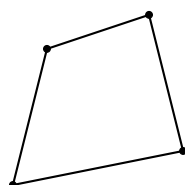


# Chương 29

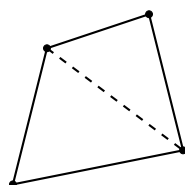
## Hình học

Trong những bài toán hình học, thường sẽ khó tìm được một hướng giải có cài đặt nhẹ nhàng và không sinh ra nhiều trường hợp đặc biệt.

Ví dụ, xét bài toán trong đó ta được cho bốn đỉnh của một tứ giác (đa giác có bốn đỉnh), và chúng ta cần tính diện tích của nó. Ví dụ, đầu vào có thể là một đa giác như hình dưới đây:



Một cách để tiếp cận bài toán là chia đôi tứ giác thành hai tam giác bằng một đoạn thẳng nối giữa hai đỉnh đối nhau:

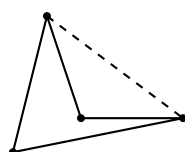


Sau đó, kết quả chính là tổng diện tích của hai tam giác. Diện tích của tam giác có thể tính bằng nhiều cách, ví dụ, sử dụng **Công thức Heron**

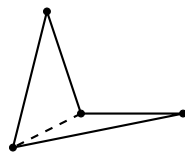
$$\sqrt{s(s-a)(s-b)(s-c)},$$

trong đó  $a$ ,  $b$  và  $c$  là độ dài các cạnh của tam giác và  $s = (a + b + c)/2$ .

Đó là một cách để giải quyết bài toán, tuy nhiên có một lỗ hổng: làm sao để chia tứ giác thành hai tam giác? Trong vài trường hợp, chúng ta không thể chỉ chọn hai đỉnh đối nhau bất kì được. Ví dụ, trong tình huống dưới đây, đoạn thẳng được chọn *nằm ngoài* tứ giác:



Tuy nhiên, một cách vẽ đoạn thẳng khác lại đúng:



Con người sẽ dễ dàng nhận biết được chọn đoạn thẳng nào là chính xác, nhưng điều đó lại tương đối khó đối với máy tính.

Tuy nhiên, chúng ta có thể giải quyết vấn đề bằng cách sử dụng một phương pháp khác thuận hơn cho việc lập trình. Cụ thể là, có một công thức tổng quát

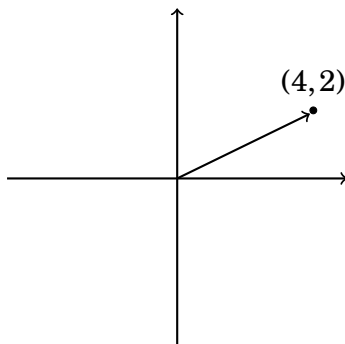
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

có thể 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, và ta thậm chí có thể tổng quát hóa để áp dụng nó cho *tất cả* đa giác.

## 29.1 Số phức

**Số phức** là số có thể viết dưới dạng  $x + yi$ , với  $i = \sqrt{-1}$  là **đơn vị ảo**. Ý nghĩa về mặt hình học của một số phức là nó biểu diễn một điểm  $(x, y)$  trong không gian hai chiều hay một vector từ gốc tọa độ đến điểm  $(x, y)$ .

Ví dụ,  $4 + 2i$  tương ứng với điểm và vector sau:



Lớp số phức complex của C++ sẽ hữu dụng khi giải bài tập về hình học. Sử dụng lớp complex chúng ta có thể biểu diễn điểm và vector dưới dạng số phức, và lớp này cũng chứa nhiều 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 điểm hoặc vector. Thêm vào đó, đoạn mã định nghĩa macro X và Y tương ứng với tọa độ x và y.

```
typedef long long C;  
typedef complex<C> P;  
#define X real()  
#define Y imag()
```

Ví dụ, đoạn mã sau khai báo 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 khai báo hai vector  $v = (3, 1)$  và  $u = (2, 2)$ , 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, những kiểu dữ liệu cho tọa độ phù hợp thường là `long` (số nguyên) hoặc `long double` (số thực). Việc sử dụng kiểu số nguyên mỗi khi có thể là một điều tốt, vì tính toán với số nguyên sẽ chính xác. Nếu cần thiết phải dùng số thực, sai số tính toán cần được xem xét đến khi so sánh các số. Một cách an toàn để kiểm tra  $a$  và  $b$  có bằng nhau không là so sánh  $|a - b| < \epsilon$ , với  $\epsilon$  là một số rất nhỏ (ví dụ,  $\epsilon = 10^{-9}$ ).

## Hàm

Trong ví dụ sau đây, kiểu dữ liệu của tọa độ là `long double`.

Hàm `abs(v)` tính độ dài  $|v|$  của vector  $v = (x, y)$  sử dụng công thức  $\sqrt{x^2 + y^2}$ . Hàm này cũng có thể được dùng để tính khoảng cách giữa 2 điểm  $(x_1, y_1)$  và  $(x_2, y_2)$ , bởi vì khoảng cách của chúng bằng với độ dài vector  $(x_2 - x_1, y_2 - y_1)$ .

Đoạn mã sau tính khoảng cách giữa hai điểm  $(4, 2)$  và  $(3, -1)$ :

```
P a = {4, 2};  
P b = {3, -1};  
cout << abs(b-a) << "\n"; // 3.16228
```

Hàm `arg(v)` tính góc được tạo bởi vector  $v = (x, y)$  và trục  $x$ . Hàm trả về giá trị góc dưới đơn vị radian, với  $r$  radian bằng với  $180r/\pi$  độ. Góc của vector hướng về phía bên phải là 0, giảm dần theo chiều kim đồng hồ và tăng dần ngược chiều kim đồng hồ.

Hàm `polar(s, a)` xây dựng một vector có độ dài  $s$  và tạo với trục  $x$  một góc  $a$ . Có thể quay một vector một góc  $a$  bằng cách nhân nó với một vector khác có độ dài 1 và góc  $a$ .

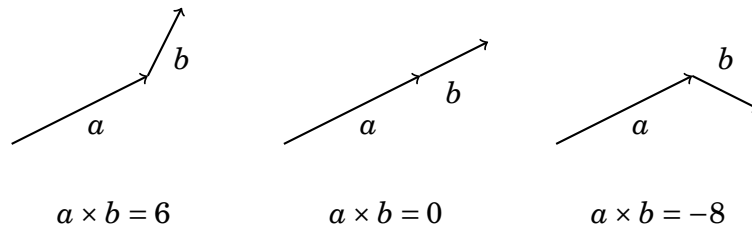
Đoạn mã sau tính góc của vector  $(4, 2)$ , xoay nó  $1/2$  radian theo ngược chiều kim đồng hồ, và sau đó tính góc lại 1 lần nữa:

```
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

**Tích chéo**  $a \times b$  của hai vector  $a = (x_1, y_1)$  và  $b = (x_2, y_2)$  được tính bởi công thức  $x_1y_2 - x_2y_1$ . Tích chéo cho chúng 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 dưới đây 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 chéo, sử dụng lớp complex:

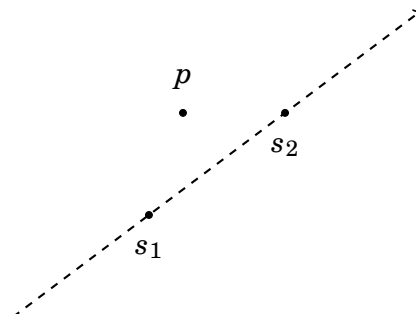
```
P a = {4,2};  
P b = {1,2};  
C p = (conj(a)*b).Y; // 6
```

Đoạn mã trên cho ra kết quả đúng, bởi vì hàm `conj` lấy tọa độ  $y$  của vector rồi đảo dấu, và khi hai vector  $(x_1, -y_1)$  và  $(x_2, y_2)$  nhân lại với nhau, tọa độ  $y$  của tích này bằng  $x_1y_2 - x_2y_1$ .

### Vị trí tương đối của điểm

Tích chéo có thể được dùng để kiểm tra xem một điểm nằm bên trái hay bên phải của một đường thẳng. Giả sử đường thẳng đi qua hai điểm  $s_1$  và  $s_2$ , chúng ta đang nhìn từ  $s_1$  đến  $s_2$  và điểm cần xác định là  $p$ .

Ví dụ, như trong hình sau,  $p$  nằm bên trái của đường thẳng:

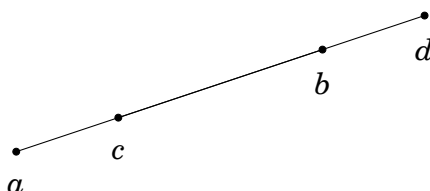


Tích chéo  $(p - s_1) \times (p - s_2)$  cho chúng ta biết vị trí điểm  $p$ . Nếu tích chéo là dương,  $p$  nằm bên trái, và nếu tích chéo là âm,  $p$  nằm bên phải. Cuối cùng, nếu tích chéo bằng không, các điểm  $s_1$ ,  $s_2$  và  $p$  cùng nằm trên một đường thẳng.

## Giao điểm của đoạn thẳng

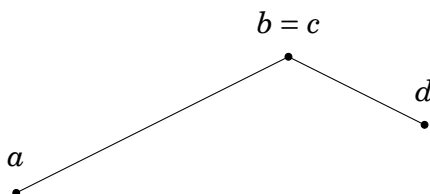
Tiếp theo ta xét bài toán kiểm tra xem liệu hai đoạn thẳng  $ab$  và  $cd$  có cắt nhau không. Những trường hợp có thể xảy ra là:

*Trường hợp 1:* Các đoạn thẳng nằm trên cùng một đường thẳng và chúng có phần chung. Trong trường hợp này, có vô số giao điểm. Ví dụ, trong hình sau đây, tất cả các điểm giữa  $c$  và  $b$  đều là giao điểm:



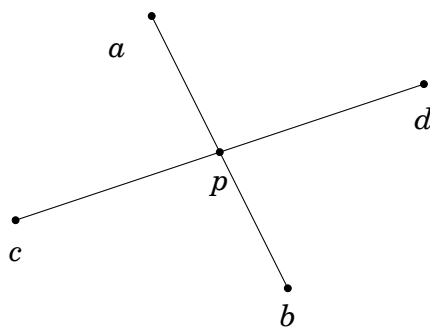
Trong trường hợp này, chúng ta có thể sử dụng tích chéo để kiểm tra xem tất cả các điểm có nằm trên cùng một đường thẳng không. Sau đó, chúng ta có thể sắp xếp các điểm và kiểm tra các đoạn thẳng có chồng lên nhau hay không.

*Trường hợp 2:* Các đoạn thẳng có một đỉnh chung và đó là giao điểm duy nhất. Ví dụ, trong hình ảnh sau đây, giao điểm là  $b = c$ :



Trường hợp này rất dễ kiểm tra, bởi vì chỉ có bốn khả năng đối với giao điểm:  $a = c$ ,  $a = d$ ,  $b = c$  và  $b = d$

*Trường hợp 3:* Có đúng một giao điểm đó không phải là đỉnh của bất kỳ đoạn thẳng nào. Trong hình dưới đây, điểm  $p$  là giao điểm:



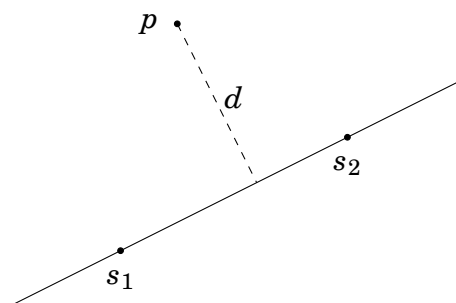
Trong trường hợp này, các đoạn thẳng giao nhau khi cả hai điểm  $c$  và  $d$  nằm trên hai phía khác nhau của đường thẳng đi qua  $a$  và  $b$ , và hai điểm  $a$  và  $b$  nằm trên hai phía khác nhau của đường thẳng đi qua  $c$  và  $d$ . Chúng ta có thể sử dụng tích chéo để kiểm tra trường hợp này.

## Khoảng cách từ điểm đến đường thẳng

Một tính chất khác của tích chéo là có thể tính diện tích của một tam giác bằng công thức

$$\frac{|(a - c) \times (b - c)|}{2},$$

với  $a$ ,  $b$  và  $c$  là ba đỉnh của tam giác. Tận dụng điều này, ta có thể rút ra được công thức tính khoảng cách ngắn nhất giữa một điểm và một đường thẳng. Ví dụ, trong hình dưới đây  $d$  là khoảng cách ngắn nhất từ điểm  $p$  tới đường thẳng được xác định bởi các điểm  $s_1$  và  $s_2$ :

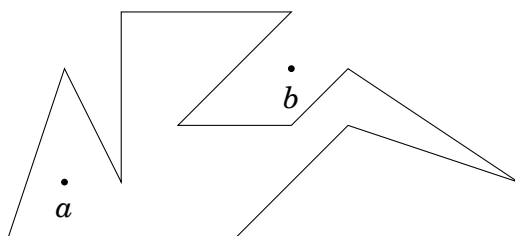


Diện tích của tam giác tạo bởi các điểm  $s_1$ ,  $s_2$  và  $p$  có thể được tính bằng hai cách, đó là  $\frac{1}{2}|s_2 - s_1|d$  và  $\frac{1}{2}((s_1 - p) \times (s_2 - p))$ . Vì vậy, khoảng cách ngắn nhất là

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

## Điểm nằm trong đa giác

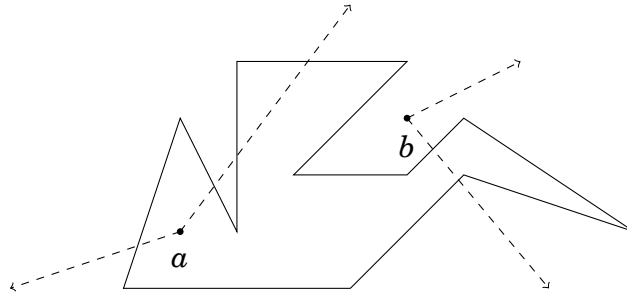
Bây giờ ta hãy xét bài toán kiểm tra xem một điểm nằm bên trong hay bên ngoài một đa giác. Ví dụ, trong hình dưới đây, điểm  $a$  nằm bên trong đa giác và điểm  $b$  nằm bên ngoài đa giác.



Một cách thuận tiện để giải quyết bài toán là chiếu một tia từ điểm đó theo một hướng bất kì và đếm số lần nó chạm vào biên của đa giác. Nếu số đó là lẻ, điểm đó nằm trong đa giác, còn nếu số đó là chẵn, thì điểm đó nằm ngoài đa giác.



Ví dụ, chúng ta có thể kẻ các tia sau:



Các tia từ  $a$  chạm 1 và 3 lần vào biên của đa giác, vì vậy  $a$  nằm trong đa giác. Tương tự, các tia từ  $b$  chạm vào 0 và 2 lần biên của đa giác, vì vậy  $b$  nằm ngoài đa giác.

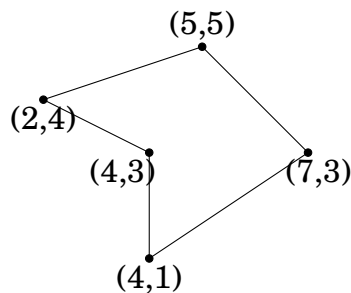
## 29.3 Diện tích đa giác

Công thức chung để tính diện tích của một đa giác, ta thường gọi nó là **công thức shoelace**, 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 điểm  $p_1 = (x_1, y_1)$ ,  $p_2 = (x_2, y_2)$ , ...,  $p_n = (x_n, y_n)$  nằm theo thứ tự sao cho  $p_i$  và  $p_{i+1}$  là các đỉnh kề nhau của đa giác, và đỉnh đầu tiên trùng đỉnh cuối cùng, nói cách khác,  $p_1 = p_n$ .

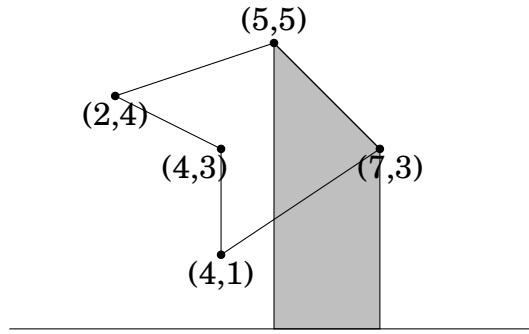
Ví dụ, diện tích của đa giác



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 tính tổng các hình thang có cạnh là một cạnh của đa giác, và một cạnh khác nằm trên đường  $y = 0$ . Ví dụ:



Diện tích của hình thang là

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

với các đỉnh  $p_i$  và  $p_{i+1}$  của đa giác. Nếu  $x_{i+1} > x_i$ , diện tích là dương, và nếu  $x_{i+1} < x_i$ , diện tích là âm.

Diện tích của đa giác là tổng diện tích của tất cả các hình thang như vậy, ta có công thức

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Lưu ý rằng ta phải lấy giá trị tuyệt đối của kết quả, bởi vì giá trị của tổng có thể dương hoặc âm, tùy thuộc vào việc chúng ta đi theo chiều kim đồng hồ hay ngược chiều kim đồng hồ dọc theo biên của đa giác.

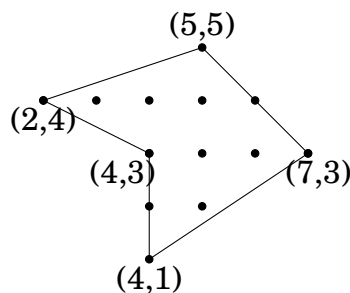
## Định lý Pick

**Định lý Pick** cho ta một cách khác để tính toán diện tích của một đa giác với điều kiện là tất cả các đỉnh của nó có tọa độ nguyên. Dựa 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 các cạnh biên của đa giác.

Ví dụ, diện tích của đa giác:



là  $6 + 7/2 - 1 = 17/2$ .

## 29.4 Hàm khoảng cách

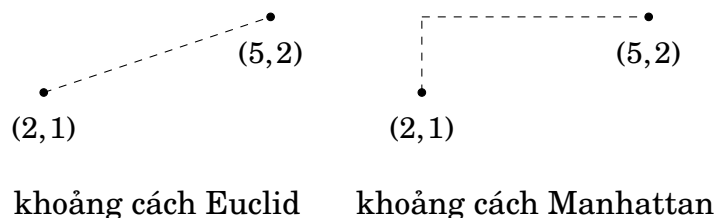
**Hàm khoảng cách** định nghĩa cách tính khoảng cách giữa hai điểm. Hàm khoảng cách thường thấy giữa hai điểm là **khoảng cách Euclid** trong đó khoảng cách giữa hai điểm  $(x_1, y_1)$  và  $(x_2, y_2)$  được tính bởi

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Một khoảng cách khác là **khoảng cách Manhattan** trong đó khoảng cách giữa hai điểm  $(x_1, y_1)$  và  $(x_2, y_2)$  là

$$|x_1 - x_2| + |y_1 - y_2|.$$

Ví dụ, xét hình sau đây:



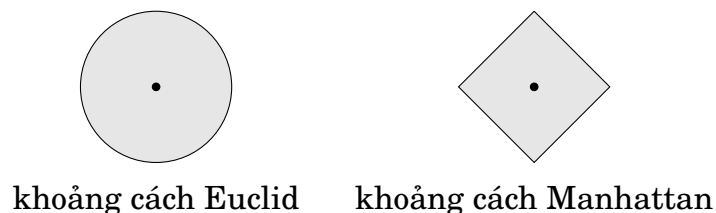
Khoảng cách Euclid giữa hai điểm là

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

và khoảng cách Manhattan là

$$|5 - 2| + |2 - 1| = 4.$$

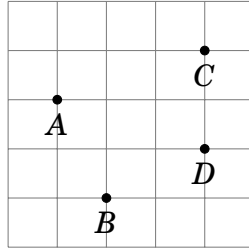
Hình ảnh sau đây cho thấy vùng các điểm có khoảng cách nhỏ hơn hoặc bằng 1 so với điểm trung tâm, ứng với khoảng cách Euclid và Manhattan:



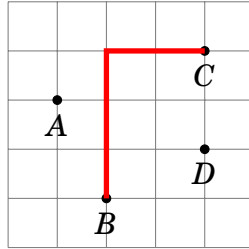
### Xoay tọa độ

Một số vấn đề sẽ dễ giải quyết hơn nếu sử dụng khoảng cách Manhattan thay vì khoảng cách Euclid. Ví dụ, xét một bài toán trong đó có  $n$  điểm trong mặt phẳng hai chiều và nhiệm vụ của chúng ta là tính khoảng cách Manhattan lớn nhất giữa hai điểm bất kỳ.

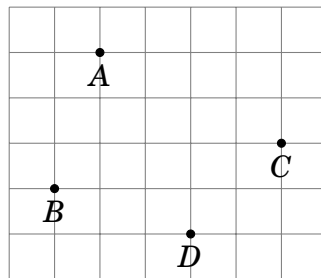
Ví dụ, chúng ta có tập điểm sau:



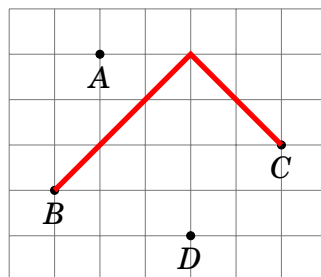
Khoảng cách Manhattan lớn nhất là 5 giữa hai điểm  $B$  và  $C$ :



Một kỹ thuật hữu ích liên quan đến khoảng cách Manhattan là xoay tất cả các tọa độ theo một góc 45 độ sao cho từ điểm  $(x, y)$  trở thành  $(x + y, y - x)$ . Ví dụ, sau khi xoay các điểm trên, ta có hình sau:



Và khoảng cách tối đa 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 ta có hai cách để tính 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  $p_2 = (3, 3)$ , sau khi xoay trở thành  $p'_1 = (1, -1)$  và  $p'_2 = (6, 0)$  và khoảng cách Manhattan là

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

Xoay tọa độ cho ta một cách dễ hơn để tính khoảng cách Manhattan, bởi vì ta có thể tính toán x và y riêng biệt. Để khoảng cách Manhattan giữa hai điểm đạt cực đại, chúng ta nên tìm hai điểm sau khi xoay có

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

là lớn nhất.

Chúng ta có thể làm điều này một cách dễ dàng, vì có thể tính riêng biệt theo x và y sau đó lấy kết quả lớn nhất.



## Chương 30

### Thuật toán đường quét

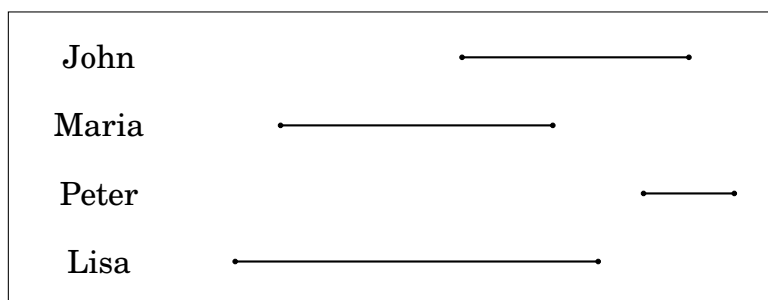
Nhiều bài toán hình học có thể được giải được bằng các thuật toán **đường quét**. Ý tưởng của thuật toán là biểu diễn vấn đề dưới dạng một tập các sự kiện tương ứng với các điểm trong mặt phẳng. Các sự kiện được xử lý theo thứ tự tăng dần theo tọa độ  $x$  hoặc  $y$  của chúng.

Ví dụ, xét bài toán sau: Có một công ty gồm  $n$  nhân viên, và ta biết thời gian mỗi nhân viên đến và rời đi trong ngày. Nhiệm vụ của chúng ta là tính toán số nhân viên nhiều nhất cùng ở trong văn phòng tại một thời điểm.

Vấn đề này có thể được giải quyết bằng cách mô hình hóa lại bài toán, trong đó mỗi nhân viên được gán 2 sự kiện tương ứng với thời gian đến và đi của họ. Sau khi sắp xếp các sự kiện, ta duyệt qua chúng và quản lý số lượng người trong văn phòng.

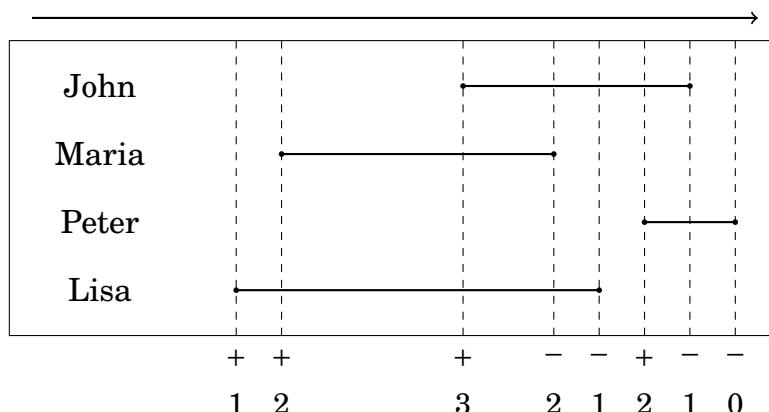
nhân viên	thời gian đến	thời gian đi
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

tương ứng với các sự kiện sau:



Chúng ta duyệt các sự kiện từ trái sang phải và duy trì một biến đếm. Mỗi khi có một người đến, ta tăng giá trị của biến đếm lên một, và khi một người rời đi, ta giảm giá trị của biến đếm xuống một. Đáp án của bài toán là giá trị lớn nhất của biến đếm trong quá trình.

Ở ví dụ trên, các sự kiện được xử lý như sau:

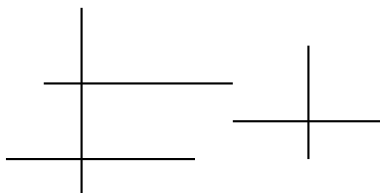


Các ký hiệu + và - cho thấy khi nào thì giá trị của biến đếm tăng hoặc giảm, và giá trị của biến đếm được thể hiện phía dưới. Giá trị lớn nhất của biến đếm là 3 trong khoảng thời gian sau khi John đến và trước lúc Maria rời đi.

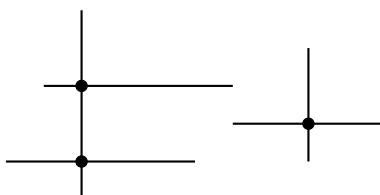
Thời gian chạy của thuật toán là  $O(n \log n)$ , vì sắp xếp tốn  $O(n \log n)$  và phần còn lại của thuật toán mất  $O(n)$ .

## 30.1 Bài toán giao điểm

Cho một tập hợp gồm  $n$  đoạn thẳng, mỗi đoạn có thể nằm ngang hoặc nằm dọc, xét bài toán đếm tổng số giao điểm. Ví dụ, khi các đoạn thẳng là



có ba giao điểm:



Chúng ta có thể dễ dàng giải quyết bài toán trong  $O(n^2)$ , bởi vì chúng ta có thể duyệt qua tất cả các cặp đoạn thẳng và kiểm tra chúng có cắt nhau hay không. Tuy nhiên, ta có thể giải quyết bài toán một cách hiệu quả hơn trong độ phức tạp  $O(n \log n)$  sử dụng thuật toán đường quét cùng với một cấu trúc dữ liệu truy vấn phạm vi.

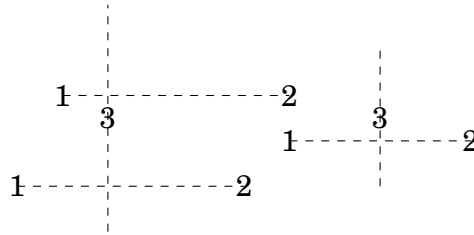
Ý tưởng là xử lý các điểm cuối của các đoạn thẳng từ trái sang phải và tập trung vào 3 loại sự kiện:

- (1) bắt đầu đoạn ngang



- (2) kết thúc đoạn ngang
- (3) đoạn dọc

Các sự kiện sau đây tương ứng với ví dụ ở trên:



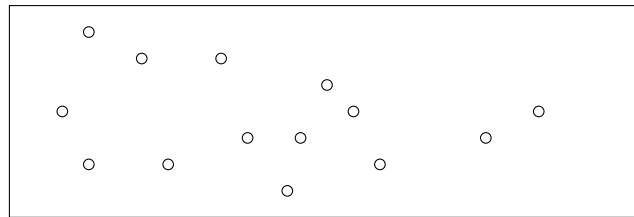
Chúng ta duyệt qua các sự kiện từ trái sang phải và sử dụng cấu trúc dữ liệu để duy trì một tập hợp các tọa độ y của những đoạn ngang đang hoạt động. Với sự kiện loại 1, chúng ta thêm tọa độ y của đoạn vào tập hợp và với sự kiện loại 2, chúng ta xóa tọa độ y khỏi tập hợp

Số điểm giao được tính ở sự kiện 3. Khi có một đoạn thẳng nằm từ  $y_1$  tới  $y_2$ , ta sẽ đếm số lượng các đoạn nằm ngang đang hoạt động mà có tọa độ y thuộc đoạn từ  $y_1$  tới  $y_2$ , rồi cộng con số này vào tổng số giao điểm.

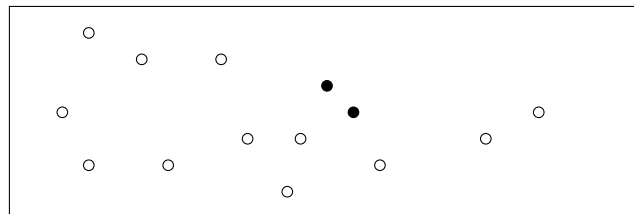
Để lưu tọa độ y của các đoạn nằm ngang, chúng ta có thể sử dụng cây phân đoạn hoặc cây chỉ số nhị phân, có thể sẽ cần nén các tọa độ. Khi các cấu trúc như vậy được sử dụng, việc xử lý từng sự kiện mất  $O(\log n)$  thời gian, vì vậy tổng thời gian chạy của thuật toán là  $O(n \log n)$ .

## 30.2 Bài toán cặp điểm gần nhất

Cho một tập hợp gồm  $n$  điểm, vấn đề tiếp theo của chúng ta là tìm 2 điểm có khoảng cách Euclid nhỏ nhất. Ví dụ, nếu có các điểm như sau



Chúng ta nên tìm những điểm sau:



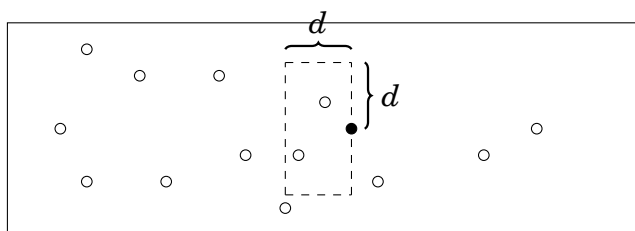
Đây là một bài toán khác có thể giải được trong độ phức tạp  $O(n \log n)$  sử dụng thuật toán đường quét<sup>1</sup>. Chúng ta duyệt qua các điểm từ trái sang

<sup>1</sup>Bên cạnh cách tiếp cận này, còn có một thuật toán chia để trị với độ phức tạp  $O(n \log n)$  [56]. Thuật toán chia các điểm thành hai tập con rồi đệ quy để giải quyết bài toán với mỗi tập đó.

phải và duy trì một giá trị  $d$ : khoảng cách tối thiểu giữa hai điểm bất kì tính đến hiện tại. Tại mỗi điểm, ta tìm điểm gần nhất ở bên trái. Nếu khoảng cách nhỏ hơn  $d$ , thì nó là khoảng cách tối thiểu mới, ta cập nhật giá trị của  $d$ .

Nếu điểm hiện tại mà ta đang xét là  $(x, y)$  và có một điểm ở bên trái với khoảng cách đến điểm hiện tại nhỏ hơn  $d$ , tọa độ  $x$  của một điểm đó phải nằm trong  $[x - d, x]$  và tọa độ  $y$  phải nằm trong  $[y - d, y + d]$ . Như vậy, ta chỉ cần xét các điểm nằm trong phạm vi đó, điều này làm cho thuật toán trở nên hiệu quả.

Ví dụ, trong hình ảnh sau đây, khu vực được đánh dấu bằng các đường nét đứt chứa các điểm có thể nằm trong bán kính  $d$  tính từ điểm đang xét:



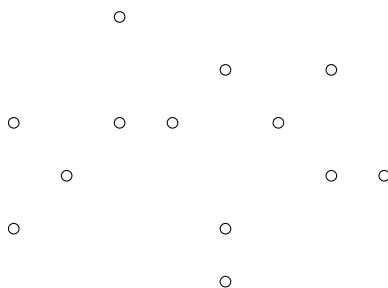
Thuật toán hiệu quả vì phạm vi ta xét luôn chứa  $O(1)$  điểm! Ta có thể duyệt qua những điểm đó trong  $O(\log n)$  bằng cách duy trì một tập hợp các điểm có tọa độ  $x$  nằm trong khoảng  $[x - d, x]$ , và được sắp tăng dần theo tọa độ  $y$  của mỗi điểm.

Độ phức tạp của thuật toán là  $O(n \log n)$ , bởi vì chúng ta duyệt qua  $n$  điểm và với mỗi điểm tìm cho nó một điểm gần nhất bên trái trong  $O(\log n)$ .

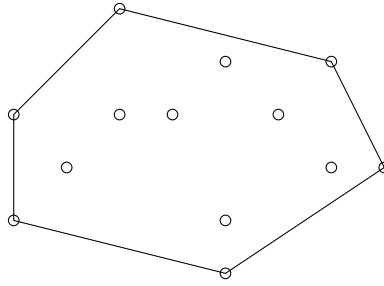
### 30.3 Bài Toán bao lồi

**Bao lồi** là đa giác lồi nhỏ nhất chứa tất cả các điểm của một tập cho trước. Lồi có nghĩa là đoạn thẳng nối giữa hai đỉnh bất kỳ của đa giác hoàn toàn nằm bên trong đa giác.

Ví dụ, cho các điểm sau:



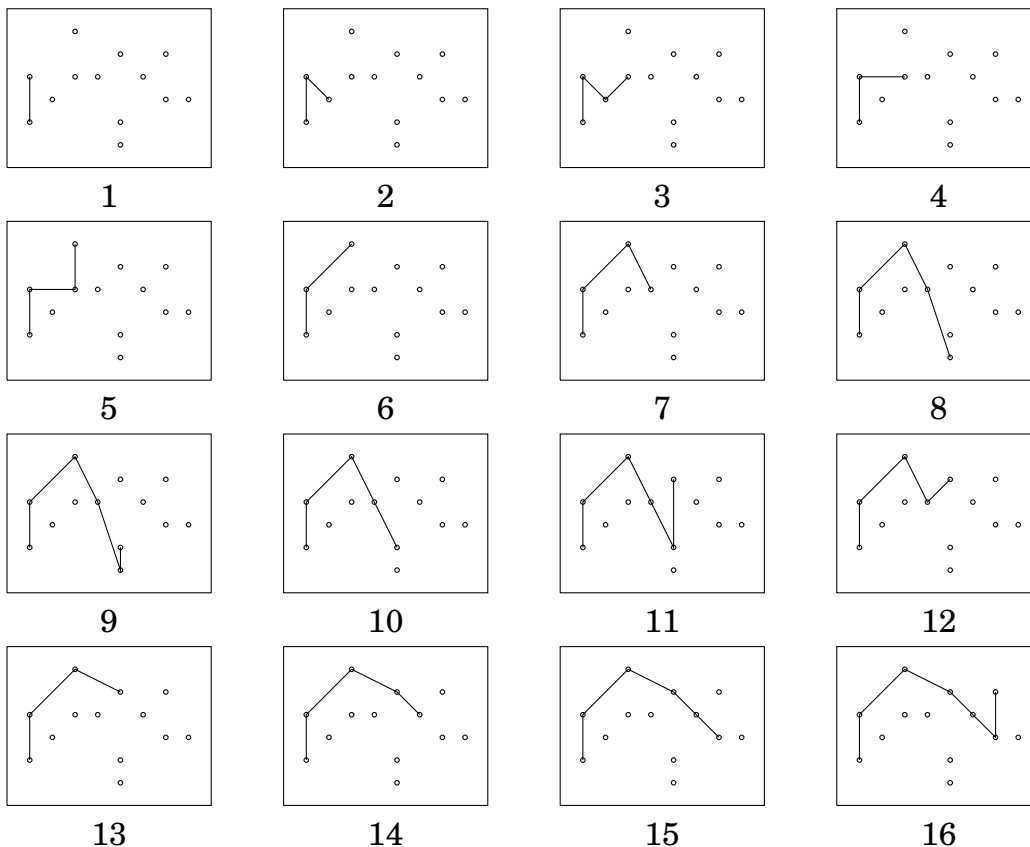
Bao lồi trông như sau

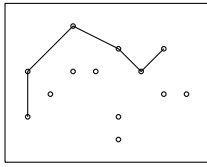


**Thuật toán Andrew** [3] là một cách làm dễ dàng để xây dựng bao lồi cho một tập hợp các điểm trong độ phức tạp  $O(n \log n)$ . Bước đầu tiên của thuật toán là xác định vị trí các điểm ngoài cùng bên trái và các điểm ngoài cùng bên phải, và sau đó xây dựng bao lồi gồm 2 phần: đầu tiên là phần thân trên và tiếp đến là phần thân dưới. Hai phần tương tự nhau, nên ta sẽ tập trung vào xây dựng phần thân trên trước.

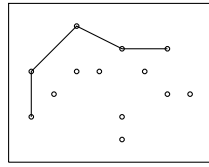
Đầu tiên, chúng ta sắp xếp các điểm theo hai tiêu chí lần lượt là tọa độ x và tọa độ y. Sau đó, ta duyệt qua các điểm và thêm từng điểm vào bao. Mỗi khi thêm một điểm vào bao, chúng ta đảm bảo rằng đoạn thẳng cuối cùng không rẽ trái. Chừng nào mà đoạn cuối này vẫn rẽ trái, ta vẫn cứ loại bỏ điểm thứ hai kể từ cuối.

Những hình ảnh sau đây minh họa cách hoạt động của thuật toán Andrew:

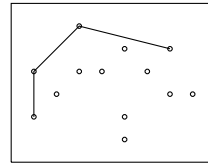




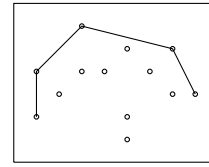
17



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] SZKOpul, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.



- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.



# Chỉ mục

- bao hàm-loại trừ, 229
- Biến ngẫu nhiên, 246
- biểu diễn bit, 103
- bài toán 2SAT, 172
- bài toán 2SUM, 86
- Bài toán 3SAT, 174
- Bài toán 3SUM, 87
- bài toán con hậu, 56
- bài toán NP-khó, 23
- băm, 263
- băm chuỗi, 263
- băm đa thức, 263
- bảng chữ cái, 261
- Bảng nhớ, 73
- bảng thưa, 93
- bạc, 119
- bạc ra, 119
- bạc vào, 119
- bổ đề Burnside, 230
- bộ ba Pythagore, 220
- bộ chung nhỏ nhất, 215
  
- cha, 143
- chu kỳ, 262
- chu trình, 118, 130
- Chu trình Euler, 188
- chu trình âm, 135
- Chuỗi Markov, 248
- cofactor, 237
- complex, 290
- con, 143
- con trở lặp, 43
- cut, 196
- cycle, 159, 165
- cycle detection, 165
- các lớp độ phức tạp, 22
- Cái túi, 78
  
- cây, 118, 143
- cây chỉ số nhị phân, 94
- cây con, 144
- cây có gốc, 143
- cây khung, 151
- cây khung lớn nhất, 152
- cây khung nhỏ nhất, 151
- cây nhị phân, 149
- Cây phân đoạn, 279
- cây phân đoạn, 97
- Cây phân đoạn hai chiều, 286
- Công thức Binet, 15
- công thức Cayley, 231
- Công thức Euclid, 220
- Công thức Faulhaber, 11
- Công thức Heron, 289
- công thức shoelace, 295
- cạnh, 117
- Cấp số nhân, 11
- cấu trúc dữ liệu các tập không giao nhau, 155
- cặp ghép, 202
- cặp ghép cực đại, 202
- cặp ghép hoàn hảo, 203
- Cặp điểm gần nhất, 303
- cực tiểu đoạn tịnh tiến, 89
  
- danh sách cạnh, 123
- Danh sách kề, 121
- data structure, 39
- deque, 47
- determinant, 237
- Dijkstra's algorithm, 163
- duyet phân tập, 61
- duyet theo chiều rộng, 127
- duyet theo chiều sâu, 125
- dynamic array, 39

dynamic segment tree, 283  
 dây bit, 46  
 dãy chèn lệch, 102  
 dãy con, 261  
 dãy con tăng dài nhất, 76  
 Dây De Bruijn, 192  
 dãy ngoặc, 227  
 Dịch chuyển bit, 105  
  
 Fenwick tree, 94  
 Fibonacci number, 238  
 flow, 195  
 Floyd's algorithm, 166  
 functional graph, 164  
  
 giai thừa - factorial, 14  
 Giao hoán - intersection, 12  
 Giao điểm, 302  
 giao điểm của đoạn thẳng, 293  
 giá trị băm, 263  
 giá trị kỳ vọng, 246  
 Giả thuyết Goldbach, 213  
 Giả thuyết Legendre, 213  
 Giải thuật tham lam, 63  
 Giải thuật Warnsdorf, 194  
 gốc, 143  
  
 Hamiltonian circuit, 191  
 heuristic, 194  
 hoán vị, 55  
 hoán vị không bất động, 230  
 Hàm khoảng cách, 297  
 hàm mex, 257  
 Hàm phi Euler, 215  
 hàm so sánh, 34  
 hàng đợi ưu tiên, 48  
 hình học, 289  
 hậu tố, 261  
 hằng số cài đặt, 24  
 hệ số nhị thức, 224  
 hệ số đa thức, 226  
 hội - conjunction, 13  
  
 identity matrix, 236  
 in-order, 149  
 independent set, 204  
 input and output, 4  
 inverse matrix, 238  
  
 khoảng cách chỉnh sửa, 80  
 khoảng cách Euclid, 297  
 khoảng cách Hamming, 108  
 khoảng cách Levenshtein, 80  
 khoảng cách Manhattan, 297  
 Khác biệt - difference, 12  
 Kirchhoff's theorem, 242  
 Kỹ thuật Euler tour, 180  
 Kết hợp - union, 12  
  
 Laplacean matrix, 242  
 lazy propagation, 280  
 lazy segment tree, 280  
 linear recurrence, 238  
 logarithm, 15  
 logarithm-tự-nhiên, 15  
 logic, 13  
 lowest common ancestor, 179  
 lá, 143  
 lát cắt hẹp nhất (cực tiểu), 199  
 Lý thuyết tập hợp, 12  
 lượng từ - quantifier, 14  
  
 ma trận kề, 122  
 macro, 9  
 matrix, 235  
 matrix multiplication, 236  
 matrix power, 237  
 maximum flow, 195  
 maximum independent set, 204  
 maximum query, 91  
 minimum cut, 196  
 minimum query, 91  
 Mã hóa Huffman, 69  
 Mã hóa nhị phân, 68  
 mã Prüfer, 232  
 Mã đi tuần, 193  
 mảng duyệt cây, 176  
 mảng tổng tiền tố, 92  
 Mảng Z, 266  
  
 next\_permutation, 55  
 nghịch lý ngày sinh nhật, 265  
 nghịch thế, 28  
 nghịch đảo đồng dư, 217  
 nguyên tố cùng nhau, 215  
 Ngôn ngữ lập trình, 3

nén chỉ số, 101  
 Nén dữ liệu, 68  
 Nút tổ tiên, 175  
  
 pair, 33  
 persistent segment tree, 285  
 Phân phối, 247  
 Phân phối cấp số nhân, 248  
 Phân phối nhị thức, 248  
 Phân phối đều, 247  
 phân tích khâu trừ, 85  
 phân tích thừa số nguyên tố, 211  
 Phép nhân ma trận, 250  
 phương pháp hai con trỏ, 85  
 phương trình Diophantine, 218  
 phần dư, 7  
 Phần phụ, 12  
 phần tử nhỏ hơn gần nhất, 87  
 phủ định - negation, 13  
 post-order, 149  
 pre-order, 149  
  
 quay lui, 56  
 queue, 48  
 quickselect, 250  
 quicksort, 250  
 Quy hoạch động, 71  
  
 random\_shuffle, 44  
 range query, 91  
 reverse, 44  
  
 set, 41  
 so khớp mẫu, 261  
 sort, 44  
 spanning tree, 242  
 sparse segment tree, 284  
 square matrix, 235  
 stack, 48  
 string, 40  
 successor graph, 164  
 sum query, 91  
 suy ra - implication, 13  
 sàng Eratosthenes, 214  
 sắp xếp, 27  
 sắp xếp, 31  
 sắp xếp nổi bọt, 28  
 Sắp xếp trộn, 29  
 sắp xếp đếm phân phối, 31  
 số Catalan, 226  
 Số Fibonacci, 15  
 số Fibonacci, 220  
 Số Grundy, 257  
 số hoàn hảo, 212  
 Số học, 211  
 Số nguyên, 6  
 số nguyên tố, 211  
 số nguyên tố sinh đôi, 213  
 số phức, 290  
 số thực, 8  
  
 Tam giác Pascal, 225  
 thuật toán Andrew, 305  
 thuật toán Bellman–Ford, 133  
 Thuật toán chia căn, 271  
 Thuật toán Dijkstra, 136  
 Thuật Toán Edmonds–Karp, 199  
 Thuật toán Euclid, 215  
 Thuật toán Euclid mở rộng, 218  
 thuật toán Floyd–Warshall, 139  
 thuật toán Ford–Fulkerson, 196  
 thuật toán Freivalds, 250  
 Thuật toán Hierholzer, 189  
 thuật toán Kadane, 25  
 thuật toán Kosaraju, 170  
 thuật toán Kruskal, 152  
 Thuật toán Las Vegas, 249  
 Thuật toán Mo, 275  
 Thuật toán Monte Carlo, 249  
 Thuật toán ngẫu nhiên, 249  
 thuật toán Prim, 157  
 thuật toán SPFA, 136  
 Thuật toán tỉ lệ, 199  
 Thuật toán Z, 266  
 Thành phần liên thông, 118  
 thành phần liên thông mạnh, 169  
 thống kê thứ tự, 250  
 thứ tự từ điển, 262  
 tiền tố, 261  
 topological sorting, 159  
 toán tử and, 104  
 toán tử not, 105  
 toán tử or, 105  
 toán tử so sánh, 33  
 toán tử xor, 105

transpose, 235  
 trie, 262  
 Truy vấn trên cây, 175  
 Trò chơi của Grundy, 259  
 trò chơi misère, 256  
 trò chơi nim, 255  
 Trạng thái thua, 253  
 Trạng thái thắng, 253  
 tuple, 33  
 typedef, 9  
 tuyển - disjunction, 13  
 tìm kiếm nhị phân, 34  
 tích chéo, 292  
 tính chia hết, 211  
 tô màu, 120, 251  
 tương đương - equivalence, 13  
 Tập con - subset, 12  
 Tập hơn con, 53  
 Tập hợp, 12  
 Tập hợp không chuỗi, 207  
 Tập phủ đường, 205  
 tập phủ đỉnh, 204  
 tập phủ đỉnh cực tiểu, 204  
 tổ hợp, 223  
 tổng hòa hợp, 214  
 tổng nim, 255  
 tổng điều hòa - harmonic sum, 12  
 Từ khóa, 68  
  
 universal set, 12  
  
 va chạm, 265  
 vector, 39, 235, 290  
 vị từ - predicate, 14  
  
 Xác suất, 243  
 Xác suất có điều kiện, 245  
 sâu, 261  
 sâu biên, 262  
 sâu con, 261  
 sâu xoay, 261  
  
 ánh xạ, 42  
  
 Đường kính, 145  
 Đường đi Euler, 187  
  
 Đường đi Hamilton, 191  
 Đường đi ngắn nhất, 133  
 Định lý Dilworth, 207  
 Định lý Dirac, 192  
 Định lý Euler, 216  
 Định lý Fermat, 216  
 Định lý Hall, 203  
 Định lý König, 204  
 Định lý Lagrange, 220  
 Định lý Ore, 192  
 Định lý Sprague–Grundy, 257  
 Định lý số dư Trung Quốc, 219  
 Định lý Wilson, 221  
 Định lý Zeckendorf, 220  
 Độc lập thống kê, 246  
 điểm, 290  
 đoạn con có tổng lớn nhất, 24  
 đoạn tịnh tiến, 89  
 đơn đồ thị, 121  
 đường quét, 301  
 đường đi, 117  
 đỉnh, 117  
 đỉnh kề, 119  
 định lý Pick, 296  
 đồng, 48  
 đồ thị, 117  
 đồ thị chính quy, 119  
 đồ thị có hướng, 118  
 đồ thị có trọng số, 119  
 đồ thị hai phía, 120, 130  
 đồ thị liên thông, 118, 129  
 đồ thị liên thông mạnh, 169  
 đồ thị thành phần liên thông, 169  
 đồ thị đầy đủ, 119  
 đồng dư thức, 7, 216  
 độ phức tạp bình phương, 22  
 độ phức tạp hằng số, 22  
 độ phức tạp logarit, 22  
 độ phức tạp lập phương, 23  
 độ phức tạp thời gian, 19  
 độ phức tạp tuyến tính, 22  
 độ phức tạp đa thức, 23  
  
 ước chung lớn nhất, 215  
 ước số, 211