

Manual do Programador Competitivo

Antti Laaksonen

Rascunho de 30 de julho de 2024

Sumário

Prefácio	vii
I Técnicas básicas	1
1 Introdução	3
1.1 Linguagens de programação	3
1.2 Entrada e saída	4
1.3 Trabalhando com números	6
1.4 Encurtando código	8
1.5 Matemática	10
1.6 Competições e recursos	15
2 Complexidade de tempo	19
2.1 Regras de cálculo	19
2.2 Classes de complexidade	22
2.3 Estimar a eficiência	23
2.4 Soma máxima de subvetor	24
3 Ordenação	27
3.1 Teoria da ordenação	27
3.2 Ordenação em C++	32
3.3 Busca binária	34
4 Estruturas de Dados	39
4.1 Vetores Dinâmicos	39
4.2 Estruturas de Conjunto	41
4.3 Estruturas de Mapa	42
4.4 Iteradores e Intervalos	43
4.5 Outras Estruturas	46
4.6 Comparação com Ordenação	50
5 Busca completa	53
5.1 Gerando subconjuntos	53
5.2 Gerando permutações	55
5.3 Backtracking	56
5.4 Podando a busca	58
5.5 Encontro no meio	60

6	Algoritmos gulosos	63
6.1	Problema das moedas	63
6.2	Escalonamento	64
6.3	Tarefas e prazos	66
6.4	Minimizando somas	67
6.5	Compressão de dados	68
7	Programação dinâmica	71
7.1	Problema das moedas	71
7.2	Maior subsequência crescente	76
7.3	Caminhos em uma grade	77
7.4	Problemas da mochila	79
7.5	Distância de edição	80
7.6	Contando os ladrilhos	82
8	Análise amortizada	85
8.1	Método dos dois ponteiros	85
8.2	Elementos menores mais próximos	87
8.3	Mínimo da janela deslizante	89
9	Consultas de intervalo	91
9.1	Consultas de vetor estático	92
9.2	Árvore binária indexada	94
9.3	Árvore de segmentos	97
9.4	Técnicas adicionais	101
10	Manipulação de bits	103
10.1	Representação de bits	103
10.2	Operações de bits	104
10.3	Representando conjuntos	106
10.4	Otimizações de bits	108
10.5	Programação dinâmica	110
II	Algoritmos de grafos	115
11	Conceitos básicos sobre grafos	117
11.1	Terminologia de grafos	117
11.2	Representação de grafos	121
12	Travessia de Grafos	125
12.1	Busca em Profundidade	125
12.2	Busca em Largura	127
12.3	Aplicações	129

13 Caminhos mínimos	133
13.1 Algoritmo de Bellman–Ford	133
13.2 Algoritmo de Dijkstra	136
13.3 Algoritmo de Floyd–Warshall	139
14 Algoritmos em Árvores	143
14.1 Travessia de Árvore	144
14.2 Diâmetro	145
14.3 Todos os Caminhos Mais Longos	147
14.4 Árvores Binárias	149
15 Árvores geradoras	151
15.1 Algoritmo de Kruskal	152
15.2 Estrutura Union-Find	155
15.3 Algoritmo de Prim	157
16 Grafos direcionados	161
16.1 Ordenação topológica	161
16.2 Programação dinâmica	163
16.3 Caminhos sucessores	166
16.4 Detecção de ciclo	167
17 Conectividade forte	169
17.1 Algoritmo de Kosaraju	170
17.2 Problema 2SAT	172
18 Consultas em Árvores	175
18.1 Encontrando Ancestrais	175
18.2 Subárvores e Caminhos	176
18.3 Ancestral Comum Mais Baixo	179
18.4 Algoritmos Offline	182
19 Caminhos e circuitos	187
19.1 Caminhos Eulerianos	187
19.2 Caminhos Hamiltonianos	191
19.3 Sequências de De Bruijn	192
19.4 Passeio do Cavalo	193
20 Fluxos e cortes	195
20.1 Algoritmo de Ford–Fulkerson	196
20.2 Caminhos disjuntos	200
20.3 Emparelhamentos Máximos	201
20.4 Coberturas por Caminhos	205
Bibliografia	209
Índice Remissivo	215

Prefácio

O objetivo deste livro é oferecer uma introdução completa à programação competitiva. É necessário que você já conheça os conceitos básicos de programação, mas não é preciso ter experiência prévia com programação competitiva.

O livro é especialmente destinado a estudantes que desejam aprender algoritmos e, possivelmente, participar da *International Olympiad in Informatics* (IOI) ou do *International Collegiate Programming Contest* (ICPC). No Brasil, a Olimpíada Brasileira de Informática (OBI) classifica para a IOI, e a Maratona de Programação da Sociedade Brasileira de Computação é a fase regional do ICPC. É claro que o livro também é adequado para qualquer pessoa interessada em programação competitiva.

Leva muito tempo para se tornar um bom programador competitivo, mas também é uma oportunidade para aprender muito. Você pode ter certeza de que o seu entendimento geral sobre algoritmos ficará muito melhor se dedicar um tempo para ler este livro, resolver problemas e participar de competições.

Esta tradução e o livro em si estão em constante desenvolvimento. Você pode enviar seu *feedback* da versão original do livro para ahslaaks@cs.helsinki.fi, ou enviar um *pull request* diretamente para fazer correções na tradução do livro.

Helsinki, agosto de 2019
Antti Laaksonen

Parte I

Técnicas básicas

Capítulo 1

Introdução

Programação competitiva combina dois tópicos: (1) o design de algoritmos e (2) a implementação de algoritmos.

O **design de algoritmos** consiste em solução de problemas e pensamento matemático. São necessárias habilidades para analisar problemas e resolvê-los de forma criativa. Um algoritmo para resolver um problema deve ser tanto correto quanto eficiente, e o cerne do problema muitas vezes é inventar um algoritmo eficiente.

O conhecimento teórico de algoritmos é importante para programadores competitivos. Tipicamente, uma solução para um problema é uma combinação de técnicas bem conhecidas e novas ideias. As técnicas que aparecem na programação competitiva também formam a base para a pesquisa científica de algoritmos.

A **implementação de algoritmos** requer boas habilidades de programação. Na programação competitiva, as soluções são avaliadas testando um algoritmo implementado usando um conjunto de casos de teste. Portanto, não é suficiente que a ideia do algoritmo seja correta, mas a implementação também deve ser correta.

Um bom estilo de codificação em competições é direto e conciso. Os programas devem ser escritos rapidamente, porque não há muito tempo disponível. Ao contrário da engenharia de *software* tradicional, os programas são curtos (geralmente com no máximo algumas centenas de linhas de código) e dispensam manutenção após a competição.

1.1 Linguagens de programação

Atualmente, as linguagens de programação mais populares usadas em competições são C++, Python e Java. Por exemplo, no Google Code Jam 2017, entre os 3.000 melhores participantes, 79% usaram C++, 16% usaram Python and 8% usaram Java [29]. Alguns participantes também usaram outras linguagens.

Muitas pessoas pensam que C++ é a melhor escolha para um programador competitivo, e o C++ está quase sempre disponível nos sistemas de competição. Os benefícios de usar C++ são ser uma linguagem muito eficiente e contar com uma biblioteca padrão com uma grande coleção de estruturas de dados e algoritmos.

Por outro lado, é bom dominar várias linguagens e entender suas forças. Por exemplo, se inteiros grandes são necessários para um problema, Python pode ser uma boa escolha, porque contém operações embutidas para cálculos com inteiros grandes. Ainda assim, a maioria dos problemas em competições de programação são definidos de forma que o uso de uma linguagem de programação específica não crie uma vantagem injusta.

Todos os exemplos de programas neste livro são escritos em C++ e as estruturas de dados e algoritmos da biblioteca padrão são frequentemente usados. Os programas seguem o padrão C++11, que pode ser usado na maioria das competições hoje em dia. Se você ainda não consegue programar em C++, agora é um bom momento para começar a aprender.

Esboço de código em C++

Um esboço de código típico em C++ para programação competitiva se parece com isso:

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

using namespace std;

int main() {
    // solucao vai aqui
}
```

A linha `#include` no início do código é uma funcionalidade do compilador g++ que nos permite incluir toda a biblioteca padrão. Assim, não é necessário incluir separadamente bibliotecas como `iostream`, `vector` e `algorithm`, mas elas ficam disponíveis automaticamente.

A linha `using` declara que as classes e funções da biblioteca padrão podem ser usadas diretamente no código. Sem a linha `using`, teríamos que escrever, por exemplo, `std::cout`, mas agora basta escrever `cout`.

O código pode ser compilado usando o seguinte comando:

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

Este comando produz um arquivo binário `test` a partir do código-fonte `test.cpp`. O compilador segue o padrão C++11 (`-std=c++11`), otimiza o código (`-O2`) e exibe avisos sobre possíveis erros (`-Wall`).

1.2 Entrada e saída

Na maioria das competições, comandos padrões são usados para ler a entrada e escrever a saída. Em C++, os comandos padrões são `cin` para entrada e `cout` para saída. Além disso, as funções em C `scanf` e `printf` podem ser usadas.

A entrada para o programa geralmente consiste de números e strings que são separados por espaços e novas linhas. Eles podem ser lidos pelo comando `cin` da

seguinte forma:

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

Esse tipo de código sempre funciona, assumindo que há pelo menos um espaço ou uma quebra de linha entre cada elemento da entrada. Por exemplo, o código acima pode ler ambas as entradas a seguir:

```
123 456 monkey
```

```
123    456  
monkey
```

O comando `cout` é usado para saída da seguinte forma:

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

As entradas e saídas às vezes são um gargalo no programa. As seguintes linhas no início do código tornam as entradas e saídas mais eficientes.

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

Note que a quebra de linha `"\n"` é mais rápida do que o `endl`, porque o `endl` sempre força uma operação de *flush*.

As funções `scanf` e `printf` da linguagem C, são uma alternativa aos comandos padrões do C++. Elas são geralmente um pouco mais rápidas, mas também são mais difíceis de usar. O código seguinte lê dois números inteiros da entrada:

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

O código seguinte imprime dois números inteiros:

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

Às vezes, o programa deve ler uma linha inteira da entrada, possivelmente contendo espaços. Isso pode ser feito usando a função `getline`:

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

Se a quantidade de dados for desconhecida, o seguinte laço é útil:

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

Este laço lê elementos da entrada um após o outro, até que não haja mais dados disponíveis na entrada.

Em alguns sistemas de competições, arquivos são usados para entrada e saída. Uma solução simples para isso é escrever o código como de costume usando comandos padrões, mas adicionar as seguintes linhas no início do código:

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

Depois disso, o programa lê a entrada do arquivo "input.txt" e escreve a saída para o arquivo "output.txt".

1.3 Trabalhando com números

Inteiros

O tipo inteiro mais utilizado em programação competitiva é o `int`, que é um tipo de 32 bits com uma faixa de valores de $-2^{31} \dots 2^{31} - 1$, ou cerca de $-2 \cdot 10^9 \dots 2 \cdot 10^9$. Se o tipo `int` não for suficiente, o tipo de 64 bits `long long` pode ser utilizado. Ele possui uma faixa de valores de $-2^{63} \dots 2^{63} - 1$, ou aproximadamente $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

O código a seguir define uma variável do tipo `long long`:

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

O sufixo `LL` significa que o tipo do número é `long long`.

Um erro comum ao usar o tipo `long long` é que o tipo `int` ainda é usado em algum lugar do código. Por exemplo, o seguinte código contém um erro sutil:

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

Embora a variável `b` seja do tipo `long long`, ambos os números na expressão `a*a` são do tipo `int` e o resultado também é do tipo `int`. Devido a isso, a variável `b` conterá um resultado incorreto. O problema pode ser resolvido alterando o tipo de `a` para `long long` ou alterando a expressão para `(long long)a*a`.

Normalmente, os problemas de competição são definidos de forma que o tipo `long long` seja suficiente. Ainda assim, é bom saber que o compilador `g++` também oferece um tipo de 128 bits chamado `__int128_t` com uma faixa de valores de $-2^{127} \dots 2^{127} - 1$, ou aproximadamente $-10^{38} \dots 10^{38}$. No entanto, este tipo não está disponível em todos os sistemas de competição.

Aritmética modular

Denotamos por $x \bmod m$ o resto da divisão de x por m . Por exemplo, $17 \bmod 5 = 2$, porque $17 = 3 \cdot 5 + 2$.

Às vezes, a resposta para um problema é um número muito grande, mas é suficiente para imprimir o "módulo m ", ou seja, o resto quando a resposta é dividida por m (por exemplo, "módulo $10^9 + 7$ "). A ideia é que, mesmo que a resposta real seja muito grande, é suficiente usar os tipos `int` e `long long`.

Uma propriedade importante do resto é que, na adição, subtração e multiplicação, o resto pode ser obtido antes da operação:

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

Assim, podemos obter o resto após cada operação e os números nunca se tornarão muito grandes.

Por exemplo, o código seguinte calcula $n!$, o fatorial de n , módulo m :

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

Normalmente, queremos que o resto esteja sempre entre $0 \dots m-1$. No entanto, em C++ e em outras linguagens, o resto de um número negativo é zero ou negativo. Uma maneira fácil de garantir que não haja restos negativos é primeiro calcular o resto como de costume e depois adicionar m se o resultado for negativo:

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

No entanto, isso só é necessário quando há subtrações no código e o resto pode se tornar negativo.

Números com ponto flutuante

Os tipos usuais de números de ponto flutuante em programação competitiva são o `double` de 64 bits e, como uma extensão no compilador g++, o `long double` de 80 bits. Na maioria dos casos, o tipo `double` é suficiente, mas o `long double` é mais preciso.

A precisão necessária da resposta geralmente é fornecida no enunciado do problema. Uma maneira fácil de imprimir a resposta é usar a função `printf` e fornecer o número de casas decimais na string de formatação. Por exemplo, o código seguinte imprime o valor de x com 9 casas decimais:

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

Uma dificuldade ao usar números de ponto flutuante é que alguns números não podem ser representados com precisão como números de ponto flutuante, o que resultará em erros de arredondamento. Por exemplo, o resultado do código seguinte é surpreendente:

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

Devido a um erro de arredondamento, o valor de x é um pouco menor do que 1, enquanto o valor correto seria 1.

É arriscado comparar números de ponto flutuante com o operador `==`, pois é possível que os valores devam ser iguais, mas não são devido a erros de precisão. Uma maneira melhor de comparar números de ponto flutuante é assumir que dois números são iguais se a diferença entre eles for menor que ϵ , onde ϵ é um número pequeno.

Na prática, os números podem ser comparados da seguinte forma ($\epsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a e b sao iguais
}
```

Observe que, embora os números de ponto flutuante sejam imprecisos, inteiros até um certo limite ainda podem ser representados com precisão. Por exemplo, usando `double`, é possível representar com precisão todos os inteiros cujo valor absoluto é no máximo 2^{53} .

1.4 Encurtando código

Códigos curtos são ideais em programação competitiva, porque os programas devem ser escritos o mais rápido possível. Por causa disso, os programadores competitivos geralmente definem nomes mais curtos para tipos de dados e outras partes do código.

Nomes para tipos

Usando o comando `typedef` é possível dar um nome mais curto para um tipo de dados. Por exemplo, o nome `long long` é longo, então podemos definir o nome mais curto `ll`:

```
typedef long long ll;
```

Depois disso, o código

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

pode ser encurtado como segue:


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

O comando `typedef` pode também ser usado com tipos de dados mais complexos. Por exemplo, o código seguinte dá o nome `vi` para um vetor de inteiros e o nome `pi` para um `pair` que contém dois inteiros.

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

Macros

Outro jeito de encurtar o código é definindo **macros**. Um macro significa que certas strings no código serão mudadas antes da compilação. Em C++, macros são definidos usando a palavra-chave `#define`.

Por exemplo, podemos definir os seguintes macros:

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

Depois disso, o código

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

pode ser encurtado como segue:

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

Um macro pode também ter parâmetros que possibilitam encurtar laços e outras estruturas. Por exemplo, podemos definir o seguinte macro:

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

Depois disso, o código

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

pode ser encurtado como segue:

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

De vez em quando, macros causam bugs que podem ser difíceis de detectar. Por exemplo, considere o seguinte macro que calcula o quadrado de um número:

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

O macro *nem sempre* funciona como esperado. Por exemplo, o código

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

corresponde ao código

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

Uma versão melhor do macro é como segue:

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

Agora o código

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

corresponde ao código

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

1.5 Matemática

A matemática desempenha um papel importante nas competições de programação, e não é possível se tornar um programador competitivo de sucesso sem ter boas habilidades matemáticas. Essa seção discute alguns conceitos matemáticos importantes e fórmulas que serão necessárias mais adiante no livro.

Fórmulas de soma

Cada soma da forma

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

onde k é um inteiro positivo, tem uma fórmula de forma fechada que é um polinômio de grau $k + 1$. Por exemplo¹,

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

¹ Existe uma fórmula mais geral para somas, chamada de **fórmula de Faulhaber**, mas ela é muito complexa para ser apresentada aqui.

e

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

Uma **progressão aritmética** é uma sequência de números onde a diferença entre quaisquer dois números consecutivos é constante. Por exemplo,

$$3, 7, 11, 15$$

é uma progressão aritmética com constante 4. A soma de uma progressão aritmética pode ser calculada usando a fórmula

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

onde a é o primeiro número, b é o último número e n é a quantidade de números. Por exemplo,

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

A fórmula é baseada no fato que a soma consiste de n números e o valor de cada número é $(a+b)/2$ em média.

A **progressão aritmética** é uma sequência de números onde a razão entre quaisquer dois números consecutivos é constante. Por exemplo,

$$3, 6, 12, 24$$

é uma progressão aritmética com constante 2. A soma de uma progressão geométrica pode ser calculada usando a fórmula

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

onde a é o primeiro número, b é o último número e a razão entre números consecutivos é k . Por exemplo,

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

Esta fórmula pode ser derivada como segue. Seja

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

Multiplicando ambos os lados por k , obtemos

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

e resolvendo a equação

$$kS - S = bk - a$$

obtemos a fórmula.

Um caso especial da soma de progressão aritmética é a fórmula

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

A **soma harmônica** é uma soma da forma

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

Um limite superior para uma soma harmônica é $\log_2(n) + 1$. Ou seja, podemos modificar cada termo $1/k$ para que k se torna a potência de dois mais próxima que não excede k . Por exemplo, quando $n = 6$, podemos estimar a soma como segue:

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

Este limite superior consiste de $\log_2(n) + 1$ partes ($1, 2 \cdot 1/2, 4 \cdot 1/4$, etc.), e o valor de cada parte é no máximo 1.

Teoria dos conjuntos

Um **conjunto** é uma coleção de elementos. Por exemplo, o conjunto

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

contém os elementos 2, 4 e 7. O símbolo \emptyset denota um conjunto vazio, e $|S|$ denota o tamanho do conjunto S , ou seja, o número de elementos no conjunto. Por exemplo, no conjunto acima, $|X| = 3$.

Se um conjunto S contém um elemento x , nós escrevemos que $x \in S$, e senão escrevemos que $x \notin S$. Por exemplo, no conjunto acima

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

Agora conjuntos podem ser construídos usando operações de conjuntos:

- A **intersecção** $A \cap B$ consiste dos elementos que estão em ambos A e B . Por exemplo, se $A = \{1, 2, 5\}$ e $B = \{2, 4\}$, então $A \cap B = \{2\}$.
- A **união** $A \cup B$ consiste dos elementos que estão em A ou B ou em ambos. Por exemplo, se $A = \{3, 7\}$ e $B = \{2, 3, 8\}$, então $A \cup B = \{2, 3, 7, 8\}$.
- O **complemento** \bar{A} consiste dos elementos que não estão em A . A interpretação de um complemento depende do **conjunto universo**, que contém todos os elementos possíveis. Por exemplo, se $A = \{1, 2, 5, 7\}$ e o conjunto universo é $\{1, 2, \dots, 10\}$, então $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- A **diferença** $A \setminus B = A \cap \bar{B}$ consiste dos elementos que estão em A mas não estão em B . Note que B pode conter elementos que não estão em A . Por exemplo, se $A = \{2, 3, 7, 8\}$ e $B = \{3, 5, 8\}$, então $A \setminus B = \{2, 7\}$.

Se cada elemento de A também pertence a S , dizemos que A é um **subconjunto** de S , denotado por $A \subset S$. Um conjunto S sempre tem $2^{|S|}$ subconjuntos, incluindo o conjunto vazio. Por exemplo, os subconjuntos do conjunto $\{2, 4, 7\}$ são

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

Alguns conjuntos usados frequentemente são \mathbb{N} (números naturais), \mathbb{Z} (inteiros), \mathbb{Q} (números racionais) e \mathbb{R} (números reais). O conjunto \mathbb{N} pode ser definidos de duas maneiras, dependendo da situação: como $\mathbb{N} = \{0, 1, 2, \dots\}$ ou $\mathbb{N} = \{1, 2, 3, \dots\}$.

Podemos também criar um conjunto usando uma regra da forma

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

onde $f(n)$ é alguma função. O conjunto contém todos os elementos da forma $f(n)$, onde n é um elemento em S . Por exemplo, o conjunto

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

contém todos os inteiros pares.

Lógica

O valor de uma expressão lógica é ou **verdadeiro** (1) ou **falso** (0). Os operadores lógicos mais importantes são \neg (**negação**), \wedge (**conjunção**), \vee (**disjunção**), \Rightarrow (**implicação**) e \Leftrightarrow (**equivalência**). A seguinte tabela mostra o significado destes operadores:

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

A expressão $\neg A$ tem valor oposto do valor de A . A expressão $A \wedge B$ é verdadeira se ambos A e B são verdadeiros, e a expressão $A \vee B$ é verdadeira se A ou B ou ambos são verdadeiros. A expressão $A \Rightarrow B$ é verdade se quando A for verdadeiro, B também for verdadeiro. A expressão $A \Leftrightarrow B$ é verdadeira se A e B ambos forem verdadeiros ou ambos falsos.

Um **predicado** é uma expressão que é verdadeira ou falsa dependendo de seus parâmetros. Predicados são geralmente denotados por letras maiúsculas. Por exemplo, podemos definir um predicado $P(x)$ que é verdadeira exatamente quando x é um número primo. Usando esta definição, $P(7)$ é verdadeiro mas $P(8)$ é falso.

Um **quantificador** conecta uma expressão lógica a elementos de um conjunto. Os quantificadores mais importantes são \forall (**para todos**) e \exists (**existe**). Por exemplo,

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

significa que para cada elemento x no conjunto, existe um elemento y no conjunto de tal forma que y é menor que x . Isso é verdadeiro no conjunto dos inteiros, mas falso no conjunto dos números naturais.

Usando a notação descrita acima, podemos expressar muitos tipos de proposições lógicas. Por exemplo,

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

significa que se um número x é maior que 1 e não é um número primo, então existem números a e b que são maiores que 1 e cujo produto é x . Esta proposição é verdadeira no conjunto dos inteiros.

Funções

A função $\lfloor x \rfloor$ arredonda o número x para baixo, e a função $\lceil x \rceil$ arredonda o número x para cima. Por exemplo,

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

As funções $\min(x_1, x_2, \dots, x_n)$ e $\max(x_1, x_2, \dots, x_n)$ retornam os menores e maiores valores x_1, x_2, \dots, x_n . Por exemplo,

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

O **fatorial** $n!$ pode ser definido como

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

ou recursivamente

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

Os **números de Fibonacci** aparecem em várias situações. Eles podem ser definidos recursivamente como segue:

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

Os primeiros números de Fibonacci são

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

Existe também uma fórmula de forma fechada para calcular os números de Fibonacci, que é algumas vezes chamada de **fórmula de Binet**:

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

Logaritmos

O **logaritmo** de um número x é denotado $\log_k(x)$, onde k é a base do logaritmo. De acordo com esta definição, $\log_k(x) = a$ exatamente quando $k^a = x$.

Uma propriedade útil dos logaritmos é que $\log_k(x)$ é equivalente ao número de vezes necessário para dividir x por k para alcançar o número 1. Por exemplo, $\log_2(32) = 5$ porque 5 divisões por 2 são necessárias:

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

Logaritmos são frequentemente usadas na análise de algoritmos, porque muitos algoritmos eficientes dividem alguma coisa em cada passo. Então, podemos estimar a eficiência destes algoritmos usando logaritmos.

O logaritmo de um produto é

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

e conseqüentemente,

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

Além disso, o logaritmo de um quociente é

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

Outra fórmula útil é

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

e usando isso, é possível calcular logaritmos para qualquer base se existe uma maneira de calcular logaritmos para uma base fixa.

O **logaritmo natural** $\ln(x)$ de um número x é um logaritmo cuja base é $e \approx 2.71828$. Outra propriedade de logaritmos é que o número de dígitos de um inteiro x na base b é $\lfloor \log_b(x) + 1 \rfloor$. Por exemplo, a representação de 123 na base 2 é 1111011 e $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Competições e recursos

IOI

A Olimpíada Internacional de Informática (IOI) é um concurso anual de programação para alunos do ensino médio. Cada país pode enviar uma equipe de quatro alunos para o concurso. Geralmente há cerca de 300 participantes de 80 países.

O IOI consiste em dois concursos de cinco horas de duração. Em ambos os concursos, os participantes são convidados a resolver três tarefas algorítmicas de várias dificuldades. As tarefas são divididas em subtarefas, cada uma das quais tem uma pontuação atribuída. Mesmo que os competidores sejam divididos em equipes, eles competem como indivíduos.

O programa da IOI [41] regula os tópicos que podem aparecer em tarefas da IOI. Quase todos os tópicos do programa IOI são cobertos por este livro.

Os participantes do IOI são selecionados por meio de concursos nacionais. Antes do IOI, muitos concursos regionais são organizados, como a Olimpíada Brasileira de Informática (OBI), a Olimpíada Báltica de Informática (BOI), a Olimpíada da Europa Central em Informática (CEOI) e a Olimpíada de Informática da Ásia-Pacífico (APIO).

Alguns países organizam concursos de prática online para futuros participantes do IOI, como o Concurso Aberto da Croácia em Informática [11] e a Olimpíada de Computação dos EUA [68]. Além disso, uma grande coleção de problemas de concursos poloneses está disponível online [60].

ICPC

O Concurso Internacional de Programação Colegiada (ICPC) é um concurso anual de programação para estudantes universitários. Cada equipe do concurso é composta por três alunos, e ao contrário do IOI, os alunos trabalham juntos; há apenas um computador disponível para cada equipe.

O ICPC é composto por várias etapas, e finalmente o melhores equipes são convidadas para as Finais Mundiais. Embora existam dezenas de milhares de participantes no concurso, há apenas um pequeno número² de vagas para as finais disponíveis, assim, avançar para as finais é uma grande conquista em algumas regiões.

Em cada prova do ICPC, as equipes têm cinco horas para resolver cerca de dez problemas de algoritmos. Uma solução para um problema só é aceita se resolver todos os casos de teste de forma eficiente. Durante a competição, os competidores poderão visualizar os resultados de outras equipes, mas na última hora o placar fica congelado e não é possível ver os resultados das últimas submissões.

Os temas que podem aparecer no ICPC não são tão bem especificados como aqueles no IOI. De qualquer forma, é claro que mais conhecimento é necessário no ICPC, especialmente mais habilidades matemáticas.

Competições online

Existem também muitos concursos online abertos a todos. No momento, o site de concursos mais ativo é o Codeforces, que organiza concursos semanais. No Codeforces, os participantes são divididos em duas divisões: iniciantes competem em Div2 e programadores mais experientes em Div1. Outros sites de concursos incluem AtCoder, CS Academy, HackerRank e Topcoder.

Algumas empresas organizam concursos online com finais presenciais. Exemplos de tais concursos são Facebook Hacker Cup, Google Code Jam e Yandex.Algorithm. Claro, as empresas também usam esses concursos para recrutamento: ter um bom desempenho em uma competição é uma boa maneira de provar suas habilidades.

Books

Já existem alguns livros (além deste) que focam em programação competitiva e resolução algorítmica de problemas:

- 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]

²O número exato de vagas para as finais variam de ano para ano; em 2017, havia 133 vagas para a final.

Os primeiros dois livros são voltados para iniciantes, enquanto que o último livro contém material avançado.

Claro, livros de algoritmos gerais também são adequados para programadores competitivos. Alguns livros populares são:

- 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]

Capítulo 2

Complexidade de tempo

A eficiência dos algoritmos é importante na programação competitiva. Normalmente, é fácil projetar um algoritmo que resolve o problema lentamente, mas o verdadeiro desafio é inventar um algoritmo rápido. Se o algoritmo for muito lento, ele receberá apenas pontos parciais ou nenhum ponto.

A **complexidade de tempo** de um algoritmo estima quanto tempo o algoritmo irá utilizar para determinada entrada. A ideia é representar a eficiência como uma função cujo parâmetro é o tamanho da entrada. Ao calcular a complexidade de tempo, podemos descobrir se o algoritmo é suficientemente rápido sem precisar implementá-lo.

2.1 Regras de cálculo

A complexidade de tempo de um algoritmo é denotada por $O(\dots)$ onde os três pontos representam alguma função. Normalmente, a variável n denota o tamanho da entrada. Por exemplo, se a entrada é uma matriz de números, n será o tamanho da matriz, e se a entrada é uma string, n será o comprimento da string.

Laços de repetição

Uma razão comum pela qual um algoritmo é lento é porque ele contém muitos laços que percorrem a entrada. Quanto mais laços aninhados o algoritmo contém, mais lento ele é. Se houver k laços aninhados, a complexidade de tempo é $O(n^k)$.

Por exemplo, a complexidade de tempo do seguinte código é $O(n)$:

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

E a complexidade de tempo do seguinte código é $O(n^2)$:

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

```
}
```

Ordem de magnitude

A complexidade de tempo não nos fornece o número exato de vezes que o código dentro de um laço é executado, mas apenas mostra a ordem de magnitude. Nos exemplos a seguir, o código dentro do laço é executado $3n$, $n+5$ e $\lceil n/2 \rceil$ vezes, mas a complexidade de tempo de cada código é $O(n)$.

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

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

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

Como outro exemplo, a complexidade de tempo do seguinte código é $O(n^2)$:

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

Fases

Se o algoritmo consiste em fases consecutivas, a complexidade de tempo total é a maior complexidade de tempo de uma única fase. A razão para isso é que a fase mais lenta geralmente é o gargalo do código.

Por exemplo, o seguinte código consiste em três fases com complexidades de tempo $O(n)$, $O(n^2)$ e $O(n)$. Portanto, a complexidade de tempo total é $O(n^2)$.

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

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

Várias variáveis

Às vezes, a complexidade de tempo depende de vários fatores. Nesse caso, a fórmula da complexidade de tempo contém várias variáveis.

Por exemplo, a complexidade de tempo do seguinte código é $O(nm)$:

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

Recursão

A complexidade de tempo de uma função recursiva depende do número de vezes que a função é chamada e da complexidade de tempo de uma única chamada. A complexidade de tempo total é o produto desses valores.

Por exemplo, considere a seguinte função:

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

A chamada $f(n)$ causa n chamadas de função, e a complexidade de tempo de cada chamada é $O(1)$. Assim, a complexidade de tempo total é $O(n)$.

Como outro exemplo, considere a seguinte função:

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

Nesse caso, cada chamada de função gera outras duas chamadas, exceto quando $n = 1$. Vamos ver o que acontece quando g é chamada com o parâmetro n . A tabela a seguir mostra as chamadas de função produzidas por essa única chamada:

chamada da função	número de chamadas
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

Com base nisso, a complexidade de tempo é

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

2.2 Classes de complexidade

A lista a seguir contém complexidades de tempo comuns de algoritmos:

$O(1)$ O tempo de execução de um algoritmo de **tempo constante** não depende do tamanho da entrada. Um algoritmo de tempo constante típico é uma fórmula direta que calcula a resposta.

$O(\log n)$ Um algoritmo **logarítmico** frequentemente reduz pela metade o tamanho da entrada em cada etapa. O tempo de execução de tal algoritmo é logarítmico, porque $\log_2 n$ equivale ao número de vezes que n precisa ser dividido por 2 para obter 1.

$O(\sqrt{n})$ Um **algoritmo de raiz quadrada** é mais lento do que $O(\log n)$, mas mais rápido do que $O(n)$. Uma propriedade especial das raízes quadradas é que $\sqrt{n} = n/\sqrt{n}$, então a raiz quadrada \sqrt{n} está, em certo sentido, no meio da entrada.

$O(n)$ Um algoritmo **linear** percorre a entrada um número constante de vezes. Isso muitas vezes é a melhor complexidade de tempo possível, porque geralmente é necessário acessar cada elemento da entrada pelo menos uma vez antes de obter a resposta.

$O(n \log n)$ Essa complexidade de tempo frequentemente indica que o algoritmo ordena a entrada, pois a complexidade de tempo dos eficientes algoritmos de ordenação é $O(n \log n)$. Outra possibilidade é que o algoritmo utilize uma estrutura de dados em que cada operação leva tempo $O(\log n)$.

$O(n^2)$ Um algoritmo **quadrático** muitas vezes contém dois laços aninhados. É possível percorrer todos os pares de elementos da entrada em tempo $O(n^2)$.

$O(n^3)$ Um algoritmo **cúbico** frequentemente contém três laços aninhados. É possível percorrer todos os trios de elementos da entrada em tempo $O(n^3)$.

$O(2^n)$ Esta complexidade de tempo frequentemente indica que o algoritmo itera por todos os subconjuntos dos elementos de entrada. Por exemplo, os subconjuntos de $\{1, 2, 3\}$ são \emptyset , $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ e $\{1, 2, 3\}$.

$O(n!)$ Esta complexidade de tempo frequentemente indica que o algoritmo itera por todas as permutações dos elementos de entrada. Por exemplo, as permutações de $\{1, 2, 3\}$ são $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ e $(3, 2, 1)$.

Um algoritmo é **polinomial** se sua complexidade de tempo for no máximo $O(n^k)$, onde k é uma constante. Todas as complexidades de tempo acima, exceto $O(2^n)$ e $O(n!)$, são polinomiais. Na prática, a constante k geralmente é pequena, e portanto, uma complexidade de tempo polinomial significa que o algoritmo é *eficiente*.

A maioria dos algoritmos neste livro é polinomial. No entanto, existem muitos problemas importantes para os quais nenhum algoritmo polinomial é conhecido, ou seja, ninguém sabe como resolvê-los de forma eficiente. Problemas **NP-difíceis** são um conjunto importante de problemas para os quais nenhum algoritmo polinomial é conhecido¹.

2.3 Estimar a eficiência

Ao calcular a complexidade de tempo de um algoritmo, é possível verificar, antes de implementá-lo, se ele é suficientemente eficiente para o problema. O ponto de partida para as estimativas é o fato de que um computador moderno pode realizar algumas centenas de milhões de operações em um segundo.

Por exemplo, vamos supor que o limite de tempo para um problema seja de um segundo e o tamanho da entrada seja $n = 10^5$. Se a complexidade de tempo for $O(n^2)$, o algoritmo executará cerca de $(10^5)^2 = 10^{10}$ operações. Isso levaria pelo menos algumas dezenas de segundos, então o algoritmo parece ser muito lento para resolver o problema.

Por outro lado, dado o tamanho da entrada, podemos tentar *adivinhar* a complexidade de tempo necessária do algoritmo que resolve o problema. A tabela a seguir contém algumas estimativas úteis, assumindo um limite de tempo de um segundo.

tamanho da entrada	complexidade de tempo necessária
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ ou $O(n)$
n é grande	$O(1)$ ou $O(\log n)$

Por exemplo, se o tamanho da entrada for $n = 10^5$, é provável que se espere que a complexidade de tempo do algoritmo seja $O(n)$ ou $O(n \log n)$. Essa informação facilita o projeto do algoritmo, pois descarta abordagens que resultariam em um algoritmo com uma complexidade de tempo pior.

¹Um livro clássico sobre o assunto é *Computers and Intractability: A Guide to the Theory of NP-Completeness* de M. R. Garey e D. S. Johnson [28].

Ainda assim, é importante lembrar que a complexidade de tempo é apenas uma estimativa de eficiência, pois ela oculta os *fatores constantes*. Por exemplo, um algoritmo que roda em tempo $O(n)$ pode realizar $n/2$ ou $5n$ operações. Isso tem um efeito importante no tempo real de execução do algoritmo.

2.4 Soma máxima de subvetor

Frequentemente, existem vários algoritmos possíveis para resolver um problema, sendo que suas complexidades de tempo são diferentes. Esta seção discute um problema clássico que possui uma solução direta com complexidade de tempo $O(n^3)$. No entanto, ao projetar um algoritmo melhor, é possível resolver o problema em tempo $O(n^2)$ e até mesmo em tempo $O(n)$.

Dado um vetor de n números, nossa tarefa é calcular a **soma máxima de subvetor**, ou seja, a maior soma possível de uma sequência de valores consecutivos no vetor². O problema é interessante quando pode haver valores negativos no vetor. Por exemplo, no vetor

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

o subvetor a seguir produz a soma máxima de 10:

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

Nós assumimos que um subvetor vazio é permitido, então a soma máxima do subvetor é sempre pelo menos 0.

Algoritmo 1

Uma maneira direta de resolver o problema é percorrer todos os subvetores possíveis, calcular a soma dos valores em cada subvetor e manter a soma máxima. O código a seguir implementa esse algoritmo:

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

²O livro *Programming Pearls* de J. Bentley [8] tornou o problema popular.

As variáveis a e b fixam o primeiro e último índice do subvetor, e a soma dos valores é calculada na variável sum . A variável $best$ contém a soma máxima encontrada durante a busca.

A complexidade de tempo do algoritmo é $O(n^3)$, pois consiste em três laços aninhados que percorrem a entrada.

Algoritmo 2

É fácil tornar o Algoritmo 1 mais eficiente removendo um laço dele. Isso é possível calculando a soma ao mesmo tempo em que o final direito do subvetor se move. O resultado é o seguinte código:

```
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";
```

Após essa alteração, a complexidade de tempo é $O(n^2)$.

Algoritmo 3

Surpreendentemente, é possível resolver o problema em tempo $O(n)^3$, o que significa que apenas um loop é necessário. A ideia é calcular, para cada posição do vetor, a soma máxima de um subvetor que termina nessa posição. Em seguida, a resposta para o problema é o máximo dessas somas.

Considere o subproblema de encontrar o subvetor de soma máxima que termina na posição k . Existem duas possibilidades:

1. O subvetor contém apenas o elemento na posição k .
2. O subvetor consiste em um subvetor que termina na posição $k - 1$, seguido pelo elemento na posição k .

No último caso, uma vez que queremos encontrar um subvetor com a soma máxima, o subvetor que termina na posição $k - 1$ também deve ter a soma máxima. Portanto, podemos resolver o problema de forma eficiente calculando a soma máxima do subvetor para cada posição final da esquerda para a direita.

O código a seguir implementa o algoritmo:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
```

³Em [8], este algoritmo de tempo linear é atribuído a J. B. Kadane, e o algoritmo é às vezes chamado de **algoritmo de Kadane**.

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

O algoritmo contém apenas um laço que percorre a entrada, portanto, a complexidade de tempo é $O(n)$. Essa também é a melhor complexidade de tempo possível, porque qualquer algoritmo para o problema precisa examinar todos os elementos do vetor pelo menos uma vez.

Comparação de eficiência

É interessante estudar como os algoritmos são eficientes na prática. A tabela a seguir mostra os tempos de execução dos algoritmos acima para diferentes valores de n em um computador moderno.

Em cada teste, a entrada foi gerada aleatoriamente. O tempo necessário para ler a entrada não foi medido.

tamanho do vetor n	Algoritmo 1	Algoritmo 2	Algoritmo 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

A comparação mostra que todos os algoritmos são eficientes quando o tamanho da entrada é pequeno, mas tamanhos maiores de entrada evidenciam diferenças notáveis nos tempos de execução dos algoritmos. O Algoritmo 1 se torna lento quando $n = 10^4$, e o Algoritmo 2 se torna lento quando $n = 10^5$. Apenas o Algoritmo 3 é capaz de processar até mesmo as maiores entradas instantaneamente.

Capítulo 3

Ordenação

Ordenação é um problema fundamental no design de algoritmos. Muitos algoritmos eficientes utilizam a ordenação como uma sub-rotina, pois frequentemente é mais fácil processar os dados quando os elementos estão ordenados.

Por exemplo, o problema "um array contém dois elementos iguais?" é fácil de resolver usando ordenação. Se o array contiver dois elementos iguais, eles estarão um ao lado do outro após a ordenação, então é fácil encontrá-los. Além disso, o problema "qual é o elemento mais frequente em um array?" pode ser resolvido de forma semelhante.

Existem muitos algoritmos para ordenação, e eles também são bons exemplos de como aplicar diferentes técnicas de design de algoritmos. Os algoritmos de ordenação eficientes funcionam em tempo $O(n \log n)$, e muitos algoritmos que usam a ordenação como sub-rotina também têm essa complexidade de tempo.

3.1 Teoria da ordenação

O problema básico na ordenação é o seguinte:

Dado um array que contém n elementos, sua tarefa é ordenar os elementos em ordem crescente.

Por exemplo, o array

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

ficará da seguinte forma após a ordenação:

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

Algoritmos $O(n^2)$

Algoritmos simples para ordenar um array operam em tempo $O(n^2)$. Tais algoritmos são curtos e geralmente consistem em dois loops aninhados. Um famoso

algoritmo de ordenação em tempo $O(n^2)$ é o **bubble sort** onde os elementos "flutuam" no array de acordo com seus valores.

O Bubble sort consiste em n rodadas. Em cada rodada, o algoritmo percorre os elementos do array. Sempre que dois elementos consecutivos são encontrados que não estão na ordem correta, o algoritmo os troca. O algoritmo pode ser implementado da seguinte forma:

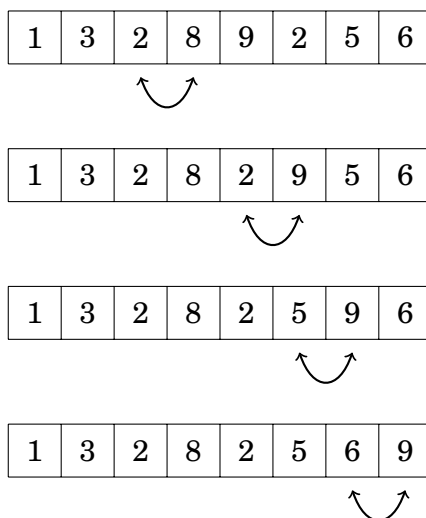
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n-1; j++) {  
        if (array[j] > array[j+1]) {  
            swap(array[j], array[j+1]);  
        }  
    }  
}
```

Após a primeira rodada do algoritmo, o maior elemento estará na posição correta, e em geral, após k rodadas, os k maiores elementos estarão nas posições corretas. Portanto, após n rodadas, o array inteiro estará ordenado.

Por exemplo, no array

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

na primeira rodada do bubble sort, os elementos são trocados da seguinte forma:



Inversões

O Bubble sort é um exemplo de um algoritmo de ordenação que sempre troca elementos *consecutivos* no array. Acontece que a complexidade de tempo de tal algoritmo é *sempre* pelo menos $O(n^2)$, porque no pior caso, são necessárias, $O(n^2)$ trocas para ordenar o array.

Um conceito útil ao analisar algoritmos de ordenação é uma **inversão**: um par de elementos de array ($array[a], array[b]$) tal que $a < b$ and $array[a] > array[b]$, ou seja, os elementos estão na ordem errada. Por exemplo, o array

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

tem três inversões: (6, 3), (6, 5) and (9, 8). O número de inversões indica o quanto de trabalho é necessário para ordenar o array. Um array está completamente ordenado quando não há inversões. Por outro lado, se os elementos do array estiverem em ordem reversa, o número de inversões é o máximo possível:

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

A troca de um par de elementos consecutivos que estão na ordem errada remove exatamente uma inversão do array. Portanto, se um algoritmo de ordenação só pode trocar elementos consecutivos, cada troca remove no máximo uma inversão, e a complexidade de tempo do algoritmo é pelo menos $O(n^2)$.

Algoritmos $O(n \log n)$

É possível ordenar um array de forma eficiente em tempo $O(n \log n)$ usando algoritmos que não estão limitados a trocar elementos consecutivos. Um desses algoritmos é o **merge sort**¹, que é baseado em recursão.

Merge sort ordena um subarray $\text{array}[a \dots b]$ da seguinte forma:

1. Se $a = b$, não faça nada, pois o subarray já está ordenado..
2. Calcule a posição do elemento do meio: $k = \lfloor (a + b)/2 \rfloor$.
3. Ordene recursivamente o subarray $\text{array}[a \dots k]$.
4. Ordene recursivamente o subarray $\text{array}[k + 1 \dots b]$.
5. *Junte* os subarrays ordenados $\text{array}[a \dots k]$ e $\text{array}[k + 1 \dots b]$ em um subarray ordenado $\text{array}[a \dots b]$.

O merge sort é um algoritmo eficiente porque ele reduz pela metade o tamanho do subarray a cada passo. A recursão consiste em $O(\log n)$ níveis, e processar cada nível leva tempo $O(n)$. Juntar os subarrays $\text{array}[a \dots k]$ e $\text{array}[k + 1 \dots b]$ é possível em tempo linear, porque eles já estão ordenados.

Por exemplo, considere ordenar o seguinte array:

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

O array será dividido em dois subarrays da seguinte forma:

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

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

Então, os subarrays serão ordenados recursivamente da seguinte forma:

¹De acordo com [47], o merge sort foi inventado por J. von Neumann em 1945.

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

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

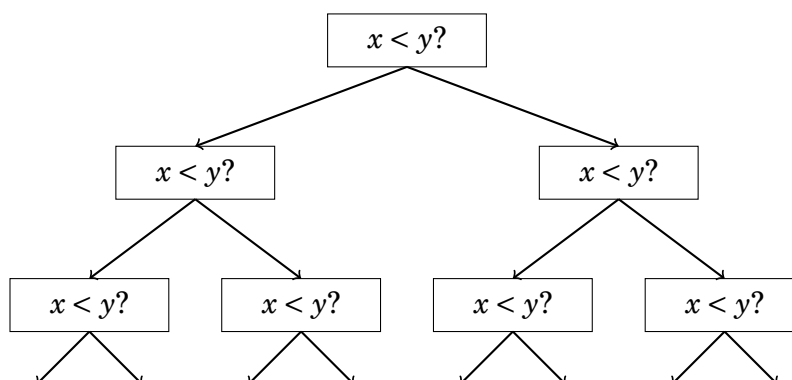
Finalmente, o algoritmo junta os subarrays ordenados e cria o array final ordenado:

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

Limite Inferior de Ordenação

É possível ordenar um array mais rápido do que em tempo $O(n \log n)$? Acontece que isso *não* é possível quando nos limitamos a algoritmos de ordenação baseados na comparação de elementos do array.

O limite inferior para a complexidade temporal pode ser demonstrado considerando a ordenação como um processo no qual cada comparação de dois elementos fornece mais informações sobre o conteúdo do array. O processo cria a seguinte árvore:



Aqui " $x < y$?" significa que alguns elementos x e y são comparados. Se $x < y$, o processo continua para a esquerda e, caso contrário, para a direita. Os resultados do processo são as possíveis maneiras de ordenar o array, um total de $n!$ maneiras. Por essa razão, a altura da árvore deve ser pelo menos

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

Obtemos um limite inferior para esta soma escolhendo os últimos $n/2$ elementos e alterando o valor de cada elemento para $\log_2(n/2)$. Isso nos dá uma estimativa

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

portanto, a altura da árvore e o número mínimo possível de etapas em um algoritmo de ordenação no pior caso é pelo menos $n \log n$.

Counting sort

O limite inferior $n \log n$ não se aplica a algoritmos que não comparam elementos de array, mas usam alguma outra informação. Um exemplo de tal algoritmo

é o **counting sort** que ordena um array em tempo $O(n)$ assumindo que cada elemento no array é um inteiro entre $0 \dots c$ e $c = O(n)$.

O algoritmo cria um *array de contagem*, cujos índices são elementos do array original. O algoritmo itera pelo array original e calcula quantas vezes cada elemento aparece no array.

Por exemplo, o array

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

corresponde ao array de contagem a seguir:

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

Por exemplo, o valor na posição 3 no array de contagem é 2, porque o elemento 3 aparece 2 vezes no array original.

A construção do array de contagem leva tempo $O(n)$. Depois disso, o array ordenado pode ser criado em tempo $O(n)$ porque o número de ocorrências de cada elemento pode ser recuperado do array de contagem. Portanto, a complexidade temporal total do counting sort é $O(n)$.

O counting sort é um algoritmo muito eficiente, mas só pode ser usado quando a constante c é pequena o suficiente, de modo que os elementos do array possam ser usados como índices no array de contagem.

3.2 Ordenação em C++

Quase nunca é uma boa ideia usar um algoritmo de ordenação feito em casa em uma competição, porque existem boas implementações disponíveis em linguagens de programação. Por exemplo, a biblioteca padrão de C++ contém a função `sort` que pode ser facilmente usada para ordenar arrays e outras estruturas de dados.

Há muitos benefícios em usar uma função de biblioteca. Primeiro, isso economiza tempo porque não há necessidade de implementar a função. Segundo, a implementação da biblioteca é certamente correta e eficiente: é improvável que uma função de ordenação feita em casa seja melhor.

Nesta seção, veremos como usar a função `sort` em C++. O código a seguir ordena um vetor em ordem crescente:

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

Após a ordenação, o conteúdo do vetor será: [2,3,3,4,5,5,8]. A ordem de classificação padrão é crescente, mas uma ordem reversa é possível da seguinte forma:

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

Um array comum pode ser ordenado da seguinte forma:

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


O seguinte código ordena a string s:

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

Ordenar uma string significa que os caracteres da string são ordenados. Por exemplo, a string "monkey" se torna "ekmnoy".

Operadores de comparação

A função sort requer que um **operador de comparação** seja definido para o tipo de dados dos elementos a serem ordenados. Ao ordenar, esse operador será usado sempre que for necessário determinar a ordem de dois elementos.

A maioria dos tipos de dados em C++ tem um operador de comparação integrado, e elementos desses tipos podem ser ordenados automaticamente. Por exemplo, números são ordenados de acordo com seus valores e strings são ordenadas em ordem alfabética.

Pares (pair) são ordenados principalmente de acordo com seus primeiros elementos (first). No entanto, se os primeiros elementos de dois pares forem iguais, eles são ordenados de acordo com seus segundos elementos (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());
```

Após isso, a ordem dos pares é: (1,2), (1,5) and (2,3).

De forma semelhante, tuplas (tuple) são ordenadas principalmente pelo primeiro elemento, secundariamente pelo segundo elemento, etc.²:

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

Após isso, a ordem das tuplas é: (1,5,3), (2,1,3) e (2,1,4).

Structs definidas pelo usuário

As structs definidas pelo usuário não possuem um operador de comparação automaticamente. O operador deve ser definido dentro da struct como uma função operator<, cujo parâmetro é outro elemento do mesmo tipo. O operador deve retornar true se o elemento for menor que o parâmetro, e false caso contrário.

²Note que em alguns compiladores mais antigos, a função make_tuple deve ser usada para criar uma tupla em vez de chaves (por exemplo, make_tuple(2,1,4) em vez de {2,1,4}).

Por exemplo, a seguinte struct P contém as coordenadas x e y de um ponto. O operador de comparação é definido de forma que os pontos sejam ordenados principalmente pela coordenada x e secundariamente pela coordenada y.

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

Funções de comparação

Também é possível fornecer uma **função de comparação** externa para a função sort como uma função de callback. Por exemplo, a seguinte função de comparação comp ordena strings principalmente por comprimento e secundariamente por ordem alfabética:

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

Agora um vetor de strings pode ser ordenado da seguinte forma:

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

3.3 Busca binária

Um método geral para buscar um elemento em um array é usar um loop for que itera pelos elementos do array. Por exemplo, o seguinte código busca por um elemento *x* no array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x encontrado no indice i
    }
}
```

A complexidade temporal desta abordagem é $O(n)$, porque no pior caso é necessário verificar todos os elementos do array. Se a ordem dos elementos for arbitrária, esta também é a melhor abordagem possível, pois não há informações adicionais disponíveis sobre onde no array devemos procurar pelo elemento *x*.

No entanto, se o array estiver *ordenado*, a situação é diferente. Neste caso, é possível realizar a busca muito mais rapidamente, porque a ordem dos elementos

no array orienta a busca. O seguinte algoritmo de **busca binária** efetua a busca por um elemento em um array ordenado de forma eficiente em tempo $O(\log n)$.

Método 1

A maneira usual de implementar a busca binária se assemelha a procurar uma palavra em um dicionário. A busca mantém uma região ativa no array, que inicialmente contém todos os elementos do array. Em seguida, um número de passos é executado, cada um dos quais divide pela metade o tamanho da região.

Em cada etapa, a busca verifica o elemento do meio da região ativa. Se o elemento do meio for o elemento alvo, a busca termina. Caso contrário, a busca continua recursivamente para a metade esquerda ou direita da região, dependendo do valor do elemento do meio.

A ideia acima pode ser implementada da seguinte forma:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x encontrado no índice k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Nesta implementação, a região ativa é $a \dots b$, e inicialmente a região é $0 \dots n-1$. O algoritmo divide o tamanho da região pela metade a cada etapa, então a complexidade temporal é $O(\log n)$.

Método 2

Um método alternativo para implementar a busca binária é baseado em uma maneira eficiente de iterar pelos elementos do array. A ideia é fazer saltos e diminuir a velocidade quando estivermos mais perto do elemento alvo.

busca percorre o array da esquerda para a direita, e o comprimento inicial do salto é $n/2$. Em cada etapa, o comprimento do salto será dividido pela metade: primeiro $n/4$, depois $n/8$, $n/16$, etc., até que finalmente o comprimento seja 1. Após os saltos, ou o elemento alvo foi encontrado ou sabemos que ele não aparece no array.

O código a seguir implementa a ideia acima:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x encontrado no índice k
}
```

Durante a busca, a variável b contém o comprimento atual do salto.. A complexidade temporal do algoritmo é $O(\log n)$, porque o código no loop while é executado no máximo duas vezes para cada comprimento de salto.

Funções em C++

A biblioteca padrão de C++ contém as seguintes funções que são baseadas em busca binária e funcionam em tempo logarítmico:

- `lower_bound` retorna um ponteiro para o primeiro elemento do array cujo valor é pelo menos x .
- `upper_bound` retorna um ponteiro para o primeiro elemento do array cujo valor é maior do que x .
- `equal_range` retorna ambos os ponteiros acima.

As funções assumem que o array está ordenado. Se não houver tal elemento, o ponteiro aponta para o elemento após o último elemento do array. Por exemplo, o seguinte código verifica se um array contém um elemento com valor x :

```
auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x encontrado no índice k
}
```

Então, o seguinte código conta o número de elementos cujo valor é x :

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

Usando `equal_range`, o código fica mais curto:

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

Encontrando a menor solução

Um uso importante para a busca binária é encontrar a posição onde o valor de uma *função* muda. Suponha que desejamos encontrar o menor valor k que é uma solução válida para um problema. Temos uma função $ok(x)$ que retorna true se x é uma solução válida e false caso contrário. Além disso, sabemos que $ok(x)$ é false quando $x < k$ e true quando $x \geq k$. A situação é a seguinte:

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

Agora, o valor de k pode ser encontrado usando busca binária

```

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

```

A busca encontra o maior valor de x para o qual $ok(x)$ é false. Assim, o próximo valor $k = x + 1$ é o menor valor possível para o qual $ok(k)$ é true. O comprimento inicial do salto z deve ser grande o suficiente, por exemplo, algum valor para o qual sabemos de antemão que $ok(z)$ é true.

O algoritmo chama a função ok $O(\log z)$ vezes, então a complexidade temporal total depende da função ok . Por exemplo, se a função funciona em tempo $O(n)$, a complexidade temporal total é $O(n \log z)$.

Encontrando o valor máximo

A busca binária também pode ser usada para encontrar o valor máximo de uma função que é primeiro crescente e depois decrescente. Nossa tarefa é encontrar uma posição k tal que

- $f(x) < f(x+1)$ quando $x < k$, e
- $f(x) > f(x+1)$ quando $x \geq k$.

A ideia é usar busca binária para encontrar o maior valor de x para o qual $f(x) < f(x+1)$. Isso implica que $k = x + 1$ porque $f(x+1) > f(x+2)$. O seguinte código implementa a busca:

```

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;

```

Note que, ao contrário da busca binária comum, aqui não é permitido que valores consecutivos da função sejam iguais. Nesse caso, não seria possível saber como continuar a busca.

Capítulo 4

Estruturas de Dados

Uma **estrutura de dados** é uma forma de armazenar dados na memória de um computador. É importante escolher uma estrutura de dados apropriada para um problema, porque cada estrutura de dados tem suas próprias vantagens e desvantagens. A questão crucial é: quais operações são eficientes na estrutura de dados escolhida?

Este capítulo apresenta as estruturas de dados mais importantes na biblioteca padrão do C++. É uma boa ideia usar a biblioteca padrão sempre que possível, porque isso economizará muito tempo. Mais adiante no livro, aprenderemos sobre mais sofisticadas estruturas de dados que não estão disponíveis na biblioteca padrão.

4.1 Vetores Dinâmicos

Um **vetor dinâmico** é um vetor cujo tamanho pode ser alterado durante a execução do programa. O vetor dinâmico mais popular em C++ é a estrutura `vector`, que pode ser usada quase como um vetor comum.

O código a seguir cria um vetor vazio e adiciona três elementos a ele:

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

Depois disso, os elementos podem ser acessados como em um vetor comum:

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

A função `size` retorna o número de elementos no vetor. O código a seguir itera através do vetor e imprime todos os elementos nele:

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

```
}
```

Uma maneira mais curta de iterar através de um vetor é a seguinte:

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

A função `back` retorna o último elemento no vetor, e a função `pop_back` remove o último elemento:

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

O código a seguir cria um vetor com cinco elementos:

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

Outra maneira de criar um vetor é fornecer o número de elementos e o valor inicial para cada elemento:

```
// tamanho 10, valor inicial 0  
vector<int> v(10);
```

```
// tamanho 10, valor inicial 5  
vector<int> v(10, 5);
```

A implementação interna de um vetor usa um vetor comum. Se o tamanho do vetor aumenta e o vetor se torna muito pequeno, um novo vetor é alocado e todos os elementos são movidos para o novo vetor. No entanto, isso não acontece com frequência e a complexidade de tempo média de `push_back` é $O(1)$.

A estrutura `string` também é um vetor dinâmico que pode ser usado quase como um vetor. Além disso, há uma sintaxe especial para strings que não está disponível em outras estruturas de dados. Strings podem ser combinadas usando o símbolo `+`. A função `substr(k , x)` retorna a substring que começa na posição k e tem comprimento x , e a função `find(t)` encontra a posição da primeira ocorrência de uma substring t .

O código a seguir apresenta algumas operações com strings:

```
string a = "hatti";  
string b = a+a;  
cout << b << "\n"; // hattihatti  
b[5] = 'v';  
cout << b << "\n"; // hattivatti
```



```
string c = b.substr(3,4);  
cout << c << "\n"; // tiva
```

4.2 Estruturas de Conjunto

Um **conjunto** é uma estrutura de dados que mantém uma coleção de elementos. As operações básicas de conjuntos são inserção de elemento, pesquisa e remoção.

A biblioteca padrão do C++ contém duas implementações de conjunto: A estrutura `set` é baseada em uma árvore binária balanceada e suas operações funcionam em tempo $O(\log n)$. A estrutura `unordered_set` usa hashing, e suas operações funcionam em tempo $O(1)$ em média.

A escolha de qual implementação de conjunto usar é frequentemente uma questão de gosto. O benefício da estrutura `set` é que ela mantém a ordem dos elementos e fornece funções que não estão disponíveis em `unordered_set`. Por outro lado, `unordered_set` pode ser mais eficiente.

O código a seguir cria um conjunto que contém inteiros, e mostra algumas das operações. A função `insert` adiciona um elemento ao conjunto, a função `count` retorna o número de ocorrências de um elemento no conjunto, e a função `erase` remove um elemento do conjunto.

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

Um conjunto pode ser usado principalmente como um vetor, mas não é possível acessar os elementos usando a notação `[]`. O código a seguir cria um conjunto, imprime o número de elementos nele e então itera por todos os elementos:

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

Uma propriedade importante dos conjuntos é que todos os seus elementos são *distintos*. Assim, a função `count` sempre retorna 0 (o elemento não está no conjunto) ou 1 (o elemento está no conjunto), e a função `insert` nunca adiciona um elemento ao conjunto se ele já estiver lá. O código a seguir ilustra isso:

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

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

C++ também contém as estruturas `multiset` e `unordered_multiset` que, de outra forma, funcionam como `set` e `unordered_set` mas podem conter várias instâncias de um elemento. Por exemplo, no código a seguir, todas as três instâncias do número 5 são adicionadas a um multiconjunto:

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

A função `erase` remove todas as instâncias de um elemento de um multiconjunto:

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

Frequentemente, apenas uma instância deve ser removida, o que pode ser feito da seguinte forma:

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

4.3 Estruturas de Mapa

Um **mapa** é um vetor generalizado que consiste em pares chave-valor. Enquanto as chaves em um vetor comum são sempre os inteiros consecutivos $0, 1, \dots, n-1$, onde n é o tamanho do vetor, as chaves em um mapa podem ser de qualquer tipo de dados e não precisam ser valores consecutivos.

A biblioteca padrão do C++ contém duas implementações de mapa que correspondem às implementações de conjunto: a estrutura `map` é baseada em uma árvore binária balanceada e acessar elementos leva tempo $O(\log n)$, enquanto a estrutura `unordered_map` usa hashing e acessar elementos leva tempo $O(1)$ em média.

O código a seguir cria um mapa onde as chaves são strings e os valores são inteiros:

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

Se o valor de uma chave for solicitado mas o mapa não o contém, a chave é adicionada automaticamente ao mapa com um valor padrão. Por exemplo, no código a seguir, a chave "aybabbtu" com valor 0 é adicionada ao mapa.

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

A função count verifica se uma chave existe em um mapa:

```
if (m.count("aybabbtu")) {  
    // a chave existe  
}
```

O código a seguir imprime todas as chaves e valores em um mapa:

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

4.4 Iteradores e Intervalos

Muitas funções na biblioteca padrão do C++ operam com iteradores. Um **iterador** é uma variável que aponta para um elemento em uma estrutura de dados.

Os iteradores frequentemente usados begin e end definem um intervalo que contém todos os elementos em uma estrutura de dados. O iterador begin aponta para o primeiro elemento na estrutura de dados, e o iterador end aponta para a posição *após* o último elemento. A situação é a seguinte:

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

Observe a assimetria nos iteradores: s.begin() aponta para um elemento na estrutura de dados, enquanto s.end() aponta para fora da estrutura de dados. Assim, o intervalo definido pelos iteradores é *semiaberto*.

Trabalhando com Intervalos

Iteradores são usados em funções da biblioteca padrão do C++ que recebem um intervalo de elementos em uma estrutura de dados. Normalmente, queremos processar todos os elementos em uma estrutura de dados, então os iteradores begin e end são fornecidos para a função.

Por exemplo, o código a seguir ordena um vetor usando a função sort, então inverte a ordem dos elementos usando a função reverse, e finalmente embaralha a ordem de os elementos usando a função random_shuffle.

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

Essas funções também podem ser usadas com um vetor comum. Nesse caso, as funções recebem ponteiros para o vetor em vez de iteradores:

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

Iteradores de Conjunto

Iteradores são frequentemente usados para acessar elementos de um conjunto. O código a seguir cria um iterador `it` que aponta para o menor elemento em um conjunto:

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

Uma maneira mais curta de escrever o código é a seguinte:

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

O elemento para o qual um iterador aponta pode ser acessado usando o símbolo `*`. Por exemplo, o código a seguir imprime o primeiro elemento no conjunto:

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

Iteradores podem ser movidos usando os operadores `++` (para frente) e `--` (para trás), o que significa que o iterador se move para o próximo ou anterior elemento no conjunto.

O código a seguir imprime todos os elementos em ordem crescente:

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

O código a seguir imprime o maior elemento no conjunto:

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

A função `find(x)` retorna um iterador que aponta para um elemento cujo valor é `x`. No entanto, se o conjunto não contém `x`, o iterador será `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x nao foi encontrado
}
```

A função `lower_bound(x)` retorna um iterador para o menor elemento no conjunto cujo valor é *pelo menos* `x`, e a função `upper_bound(x)` retorna um iterador para o menor elemento no conjunto cujo valor é *maior que* `x`. Em ambas as funções, se tal elemento não existe, o valor de retorno é `end`. Essas funções

não são suportadas pela estrutura `unordered_set` que não mantém a ordem dos elementos.

Por exemplo, o código a seguir encontra o elemento mais próximo a x :

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

O código assume que o conjunto não está vazio, e passa por todos os casos possíveis usando um iterador `it`. Primeiro, o iterador aponta para o menor elemento cujo valor é pelo menos x . Se `it` for igual a `begin`, o elemento correspondente está mais próximo de x . Se `it` for igual a `end`, o maior elemento no conjunto está mais próximo de x . Se nenhum dos casos anteriores for válido, o elemento mais próximo a x é o elemento que corresponde a `it` ou o elemento anterior.

4.5 Outras Estruturas

Bitset

Um **bitset** é um vetor cujo cada valor é 0 ou 1. Por exemplo, o código a seguir cria um `bitset` que contém 10 elementos:

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

O benefício de usar bitsets é que eles requerem menos memória do que vetores comuns, porque cada elemento em um `bitset` apenas usa um bit de memória. Por exemplo, se n bits são armazenados em um vetor `int`, $32n$ bits de memória serão usados, mas um `bitset` correspondente requer apenas n bits de memória. Além disso, os valores de um `bitset` podem ser manipulados eficientemente usando operadores de bits, o que torna possível otimizar algoritmos usando conjuntos de bits.

O código a seguir mostra outra maneira de criar o `bitset` acima:

```
bitset<10> s(string("0010011010")); // da direita para a esquerda
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

A função `count` retorna o número de uns no `bitset`:

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

O código a seguir mostra exemplos de uso de operações de bits:

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

Deque

Um **deque** é um vetor dinâmico cujo tamanho pode ser eficientemente alterado em ambas as extremidades do vetor. Como um vetor, um deque fornece as funções `push_back` e `pop_back`, mas também inclui as funções `push_front` e `pop_front` que não estão disponíveis em um vetor.

Um deque pode ser usado da seguinte forma:

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

A implementação interna de um deque é mais complexa do que a de um vetor, e por esta razão, um deque é mais lento que um vetor. Ainda assim, adicionar e remover elementos leva tempo $O(1)$ em média em ambas as extremidades.

Pilha

Uma **pilha** é uma estrutura de dados que fornece duas operações de tempo $O(1)$: adicionar um elemento ao topo, e remover um elemento do topo. Só é possível acessar o topo elemento de uma pilha.

O código a seguir mostra como uma pilha pode ser usada:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
```

```
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Fila

Uma **fila** também fornece duas operações de tempo $O(1)$: adicionar um elemento ao final da fila, e remover o primeiro elemento da fila. Só é possível acessar o primeiro e último elemento de uma fila.

O código a seguir mostra como uma fila pode ser usada:

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

Fila de Prioridade

Uma **fila de prioridade** mantém um conjunto de elementos. As operações suportadas são inserção e, dependendo do tipo de fila, recuperação e remoção de o elemento mínimo ou máximo. A inserção e remoção levam tempo $O(\log n)$, e a recuperação leva tempo $O(1)$.

Enquanto um conjunto ordenado suporta eficientemente todas as operações de uma fila de prioridade, o benefício de usar uma fila de prioridade é que ela tem fatores constantes menores. Uma fila de prioridade é geralmente implementada usando uma estrutura de heap que é muito mais simples do que uma árvore binária balanceada usada em um conjunto ordenado.

Por padrão, os elementos em uma fila de prioridade C++ são classificados em ordem decrescente, e é possível encontrar e remover o maior elemento da fila. O código a seguir ilustra isso:

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


Se quisermos criar uma fila de prioridade que suporte encontrar e remover o menor elemento, podemos fazê-lo da seguinte forma:

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

Estruturas de Dados Baseadas em Políticas

O compilador g++ também suporta algumas estruturas de dados que não fazem parte da biblioteca padrão C++. Tais estruturas são chamadas de estruturas de dados *baseadas em políticas*. Para usar essas estruturas, as seguintes linhas devem ser adicionadas ao código:

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

Depois disso, podemos definir uma estrutura de dados `indexed_set` que é como set mas pode ser indexada como um vetor. A definição para valores `int` é a seguinte:

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

Agora podemos criar um conjunto da seguinte forma:

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

A especialidade deste conjunto é que temos acesso a os índices que os elementos teriam em um vetor ordenado. A função `find_by_order` retorna um iterador para o elemento em uma determinada posição:

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

E a função `order_of_key` retorna a posição de um determinado elemento:

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

Se o elemento não aparecer no conjunto, obtemos a posição que o elemento teria no conjunto:

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

Ambas as funções funcionam em tempo logarítmico.

4.6 Comparação com Ordenação

Muitas vezes é possível resolver um problema usando estruturas de dados ou ordenação. Às vezes, existem diferenças notáveis na eficiência real dessas abordagens, que podem estar ocultas em suas complexidades de tempo.

Vamos considerar um problema onde recebemos duas listas A e B que contêm n elementos. Nossa tarefa é calcular o número de elementos que pertencem a ambas as listas. Por exemplo, para as listas

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

a resposta é 3 porque os números 2, 5 e 9 pertencem a ambas as listas.

Uma solução direta para o problema é percorrer todos os pares de elementos em tempo $O(n^2)$, mas a seguir vamos nos concentrar em algoritmos mais eficientes.

Algoritmo 1

Construímos um conjunto dos elementos que aparecem em A , e depois disso, iteramos pelos elementos de B e verificamos para cada elemento se ele também pertence a A . Isso é eficiente porque os elementos de A estão em um conjunto. Usando a estrutura `set`, a complexidade de tempo do algoritmo é $O(n \log n)$.

Algoritmo 2

Não é necessário manter um conjunto ordenado, então, em vez da estrutura `set` também podemos usar a estrutura `unordered_set`. Esta é uma maneira fácil de tornar o algoritmo mais eficiente, porque só temos que mudar a estrutura de dados subjacente. A complexidade de tempo do novo algoritmo é $O(n)$.

Algoritmo 3

Em vez de estruturas de dados, podemos usar a ordenação. Primeiro, ordenamos as listas A e B . Depois disso, iteramos pelas duas listas ao mesmo tempo e encontramos os elementos comuns. A complexidade de tempo da ordenação é $O(n \log n)$, e o resto do algoritmo funciona em tempo $O(n)$, então a complexidade de tempo total é $O(n \log n)$.

Comparação de Eficiência

A tabela a seguir mostra a eficiência dos algoritmos acima quando n varia e os elementos das listas são inteiros aleatórios entre $1 \dots 10^9$:

n	Algoritmo 1	Algoritmo 2	Algoritmo 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

Os algoritmos 1 e 2 são iguais, exceto que eles usam estruturas de conjunto diferentes. Neste problema, esta escolha tem um efeito importante sobre o tempo de execução, porque o Algoritmo 2 é 4 a 5 vezes mais rápido que o Algoritmo 1.

No entanto, o algoritmo mais eficiente é o Algoritmo 3 que usa ordenação. Ele usa apenas metade do tempo em comparação com o Algoritmo 2. Curiosamente, a complexidade de tempo do Algoritmo 1 e do Algoritmo 3 é $O(n \log n)$, mas apesar disso, o Algoritmo 3 é dez vezes mais rápido. Isso pode ser explicado pelo fato de que a ordenação é um procedimento simples e é feito apenas uma vez no início do Algoritmo 3, e o resto do algoritmo funciona em tempo linear. Por outro lado, o Algoritmo 1 mantém uma árvore binária balanceada complexa durante todo o algoritmo.

Capítulo 5

Busca completa

Busca completa é um método geral que pode ser usado para resolver quase qualquer problema algorítmico. A ideia é gerar todas as soluções possíveis para o problema usando força bruta, e então selecionar a melhor solução ou contar o número de soluções, dependendo do problema.

A busca completa é uma boa técnica se houver tempo suficiente para verificar todas as soluções, porque a busca é geralmente fácil de implementar e sempre fornece a resposta correta. Se a busca completa for muito lenta, outras técnicas, como algoritmos gulosos ou programação dinâmica, podem ser necessárias.

5.1 Gerando subconjuntos

Consideramos primeiro o problema de gerar todos os subconjuntos de um conjunto de n elementos. Por exemplo, os subconjuntos de $\{0, 1, 2\}$ são \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ e $\{0, 1, 2\}$. Existem dois métodos comuns para gerar subconjuntos: podemos realizar uma busca recursiva ou explorar a representação de bits de inteiros.

Método 1

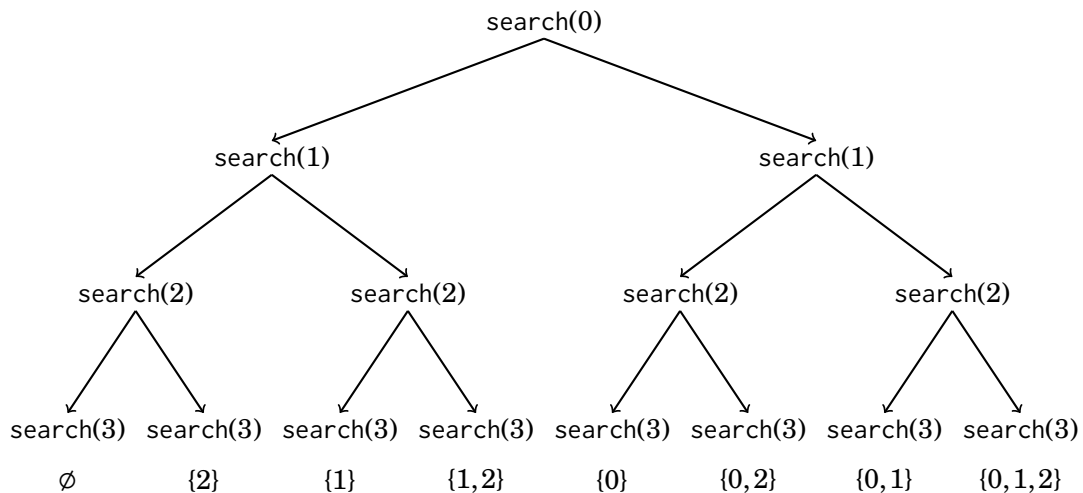
Uma maneira elegante de percorrer todos os subconjuntos de um conjunto é usar recursão. A seguinte função `search` gera os subconjuntos do conjunto $\{0, 1, \dots, n-1\}$. A função mantém um vetor `subset` que conterá os elementos de cada subconjunto. A busca começa quando a função é chamada com o parâmetro 0.

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

```
}
```

Quando a função `search` é chamada com o parâmetro k , ela decide se inclui o elemento k no subconjunto ou não, e em ambos os casos, então chama a si mesma com o parâmetro $k + 1$. No entanto, se $k = n$, a função percebe que todos os elementos foram processados e um subconjunto foi gerado.

A seguinte árvore ilustra as chamadas de função quando $n = 3$. Podemos sempre escolher o ramo esquerdo (k não está incluído no subconjunto) ou o ramo direito (k está incluído no subconjunto).



Método 2

Outra maneira de gerar subconjuntos é baseada na representação de bits de inteiros. Cada subconjunto de um conjunto de n elementos pode ser representado como uma sequência de n bits, que corresponde a um inteiro entre $0 \dots 2^n - 1$. Os uns na sequência de bits indicam quais elementos estão incluídos no subconjunto.

A convenção usual é que o último bit corresponde ao elemento 0, o penúltimo bit corresponde ao elemento 1, e assim por diante. Por exemplo, a representação de bits de 25 é 11001, que corresponde ao subconjunto $\{0, 3, 4\}$.

O seguinte código percorre os subconjuntos de um conjunto de n elementos:

```
for (int b = 0; b < (1<<n); b++) {  
    // processa subconjunto  
}
```

O seguinte código mostra como podemos encontrar os elementos de um subconjunto que corresponde a uma sequência de bits. Ao processar cada subconjunto, o código constrói um vetor que contém os elementos no subconjunto.

```
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 Gerando permutações

A seguir, consideramos o problema de gerar todas as permutações de um conjunto de n elementos. Por exemplo, as permutações de $\{0, 1, 2\}$ são $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ e $(2, 1, 0)$. Novamente, existem duas abordagens: podemos usar recursão ou percorrer as permutações iterativamente.

Método 1

Assim como os subconjuntos, as permutações podem ser geradas usando recursão. A seguinte função `search` percorre as permutações do conjunto $\{0, 1, \dots, n-1\}$. A função constrói um vetor `permutation` que contém a permutação, e a busca começa quando a função é chamada sem parâmetros.

```
void search() {
    if (permutation.size() == n) {
        // processa permutacao
    } 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();
        }
    }
}
```

Cada chamada de função adiciona um novo elemento a `permutation`. O array `chosen` indica quais elementos já estão incluídos na permutação. Se o tamanho de `permutation` for igual ao tamanho do conjunto, uma permutação foi gerada.

Método 2

Outro método para gerar permutações é começar com a permutação $\{0, 1, \dots, n-1\}$ e repetidamente usar uma função que constrói a próxima permutação em ordem crescente. A biblioteca padrão C++ contém a função `next_permutation` que pode ser usada para isso:

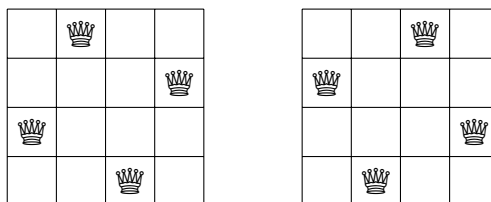
```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
```

```
// processa permutacao
} while (next_permutation(permutation.begin(), permutation.end()));
```

5.3 Backtracking

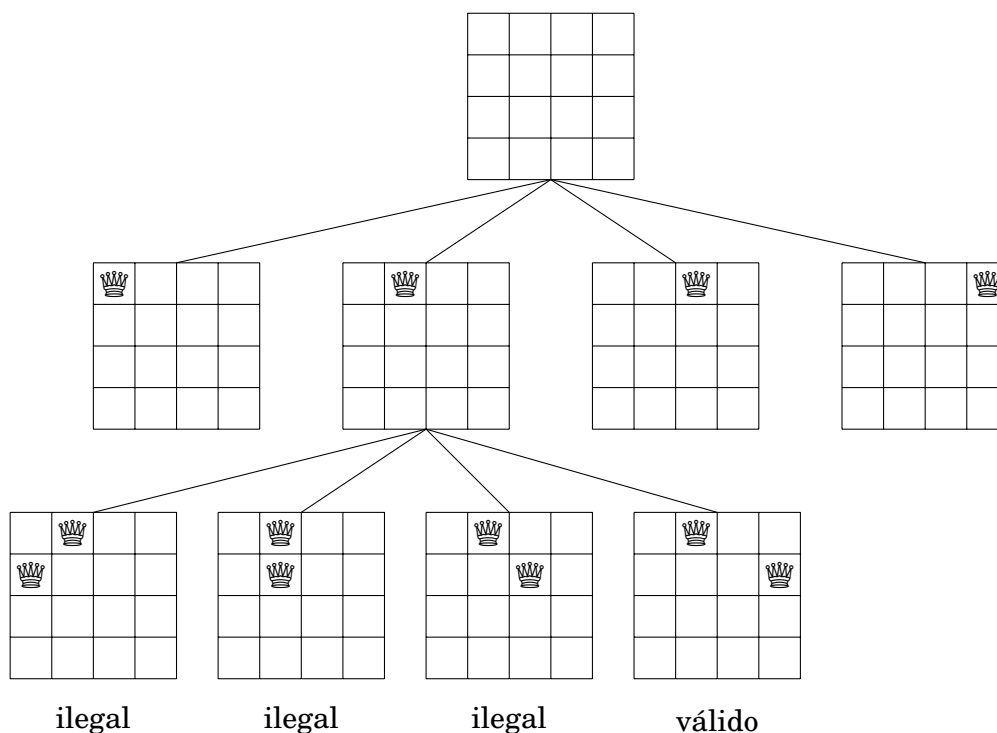
Um algoritmo de **backtracking** começa com uma solução vazia e estende a solução passo a passo. A busca percorre recursivamente todas as maneiras diferentes de como uma solução pode ser construída.

Como exemplo, considere o problema de calcular o número de maneiras pelas quais n rainhas podem ser colocadas em um tabuleiro de xadrez $n \times n$ para que nenhuma rainha ataque a outra. Por exemplo, quando $n = 4$, existem duas soluções possíveis:



O problema pode ser resolvido usando backtracking colocando rainhas no tabuleiro linha por linha. Mais precisamente, exatamente uma rainha será colocada em cada linha para que nenhuma rainha ataque qualquer uma das rainhas colocadas anteriormente. Uma solução foi encontrada quando todas as n rainhas foram colocadas no tabuleiro.

Por exemplo, quando $n = 4$, alguma solução parcial gerada pelo algoritmo de backtracking é a seguinte:



No nível inferior, as três primeiras configurações são ilegais, porque as rainhas se atacam. No entanto, a quarta configuração é válida e pode ser estendida para uma solução completa por colocando mais duas rainhas no tabuleiro. Existe apenas uma maneira de colocar as duas rainhas restantes.

O algoritmo pode ser implementado da seguinte forma:

```
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;
    }
}
```

A busca começa chamando `search(0)`. O tamanho do tabuleiro é $n \times n$, e o código calcula o número de soluções para `count`.

O código assume que as linhas e colunas do tabuleiro são numeradas de 0 a $n - 1$. Quando a função `search` é chamada com o parâmetro y , ela coloca uma rainha na linha y e então chama a si mesma com o parâmetro $y + 1$. Então, se $y = n$, uma solução foi encontrada e a variável `count` é incrementada em um.

O array `column` mantém o controle das colunas que contêm uma rainha, e os arrays `diag1` e `diag2` mantêm o controle das diagonais. Não é permitido adicionar outra rainha a uma coluna ou diagonal que já contém uma rainha. Por exemplo, as colunas e diagonais do tabuleiro 4×4 são numeradas da seguinte forma:

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

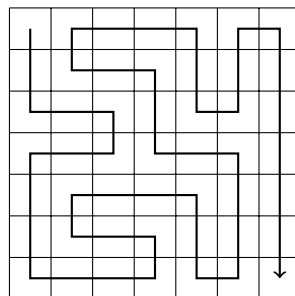
Seja $q(n)$ o número de maneiras de colocar n rainhas em um tabuleiro de xadrez $n \times n$. O algoritmo de backtracking acima nos diz que, por exemplo, $q(8) = 92$. Quando n aumenta, a busca rapidamente se torna lenta, porque o número de soluções aumenta exponencialmente. Por exemplo, calcular $q(16) = 14772512$ usando o algoritmo acima já leva cerca de um minuto em um computador moderno¹.

¹Não há maneira conhecida de calcular com eficiência valores maiores de $q(n)$. O recorde atual é $q(27) = 234907967154122528$, calculado em 2016 [55].

5.4 Podando a busca

Muitas vezes podemos otimizar o backtracking podando a árvore de busca. A ideia é adicionar "inteligência" ao algoritmo para que ele perceba o mais rápido possível se uma solução parcial não pode ser estendida para uma solução completa. Essas otimizações podem ter um tremendo efeito na eficiência da busca.

Vamos considerar o problema de calcular o número de caminhos em uma grade $n \times n$ do canto superior esquerdo para o canto inferior direito, de forma que o caminho visite cada quadrado exatamente uma vez. Por exemplo, em uma grade 7×7 , existem 111712 tais caminhos. Um dos caminhos é o seguinte:



Vamos nos concentrar no caso 7×7 , porque seu nível de dificuldade é apropriado às nossas necessidades. Começamos com um algoritmo de backtracking direto, e então o otimizamos passo a passo usando observações de como a busca pode ser podada. Após cada otimização, medimos o tempo de execução do algoritmo e o número de chamadas recursivas, para que possamos ver claramente o efeito de cada otimização na eficiência da busca.

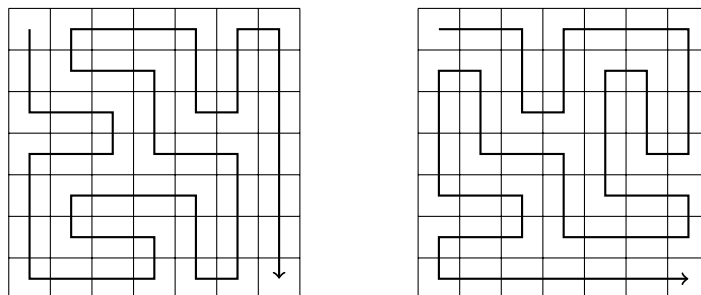
Algoritmo básico

A primeira versão do algoritmo não contém nenhuma otimização. Nós simplesmente usamos backtracking para gerar todos os caminhos possíveis do canto superior esquerdo para o canto inferior direito e contamos o número de tais caminhos.

- tempo de execução: 483 segundos
- número de chamadas recursivas: 76 bilhões

Otimização 1

Em qualquer solução, primeiro nos movemos um passo para baixo ou para a direita. Há sempre dois caminhos que são simétricos sobre a diagonal da grade após o primeiro passo. Por exemplo, os seguintes caminhos são simétricos:

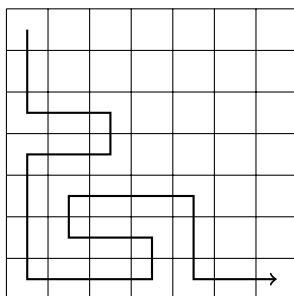


Portanto, podemos decidir que sempre nos movemos um passo para baixo (ou para a direita), e finalmente multiplicamos o número de soluções por dois.

- tempo de execução: 244 segundos
- número de chamadas recursivas: 38 bilhões

Otimização 2

Se o caminho atingir o quadrado inferior direito antes de visitar todos os outros quadrados da grade, é claro que não será possível completar a solução. Um exemplo disso é o seguinte caminho:

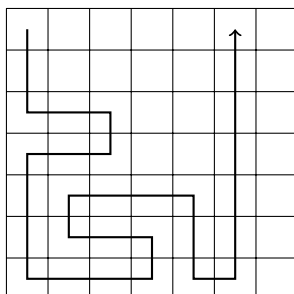


Usando essa observação, podemos encerrar a busca imediatamente se atingirmos o quadrado inferior direito muito cedo.

- tempo de execução: 119 segundos
- número de chamadas recursivas: 20 bilhões

Otimização 3

Se o caminho tocar uma parede e puder virar à esquerda ou à direita, a grade se divide em duas partes que contêm quadrados não visitados. Por exemplo, na seguinte situação, o caminho pode virar à esquerda ou à direita:

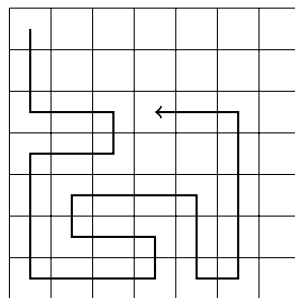


Neste caso, não podemos mais visitar todos os quadrados, então podemos encerrar a busca. Esta otimização é muito útil:

- tempo de execução: 1.8 segundos
- número de chamadas recursivas: 221 milhões

Otimização 4

A ideia da Otimização 3 pode ser generalizada: se o caminho não puder continuar em frente mas pode virar à esquerda ou à direita, a grade se divide em duas partes que contêm quadrados não visitados. Por exemplo, considere o seguinte caminho:



É claro que não podemos mais visitar todos os quadrados, então podemos encerrar a busca. Após esta otimização, a busca é muito eficiente:

- tempo de execução: 0.6 segundos
- número de chamadas recursivas: 69 milhões

Agora é um bom momento para parar de otimizar o algoritmo e ver o que alcançamos. O tempo de execução do algoritmo original foi de 483 segundos, e agora, após as otimizações, o tempo de execução é de apenas 0.6 segundos. Assim, o algoritmo se tornou quase 1000 vezes mais rápido após as otimizações.

Este é um fenômeno usual em backtracking, porque a árvore de busca é geralmente grande e até mesmo observações simples podem efetivamente podar a busca. Especialmente úteis são as otimizações que ocorrem durante as primeiras etapas do algoritmo, ou seja, no topo da árvore de busca.

5.5 Encontro no meio

Encontrar no meio é uma técnica onde o espaço de busca é dividido em duas partes de tamanho aproximadamente igual. Uma busca separada é realizada para ambas as partes, e finalmente os resultados das buscas são combinados.

A técnica pode ser usada se houver uma maneira eficiente de combinar os resultados das buscas. Nessa situação, as duas buscas podem exigir menos tempo

do que uma busca grande. Tipicamente, podemos transformar um fator de 2^n em um fator de $2^{n/2}$ usando a técnica de encontro no meio.

Como exemplo, considere um problema onde recebemos uma lista de n números e um número x , e queremos descobrir se é possível escolher alguns números da lista de modo que sua soma seja x . Por exemplo, dada a lista $[2, 4, 5, 9]$ e $x = 15$, podemos escolher os números $[2, 4, 9]$ para obter $2 + 4 + 9 = 15$. No entanto, se $x = 10$ para a mesma lista, não é possível formar a soma.

Um algoritmo simples para o problema é percorrer todos os subconjuntos dos elementos e verificar se a soma de qualquer um dos subconjuntos é x . O tempo de execução de tal algoritmo é $O(2^n)$, porque existem 2^n subconjuntos. No entanto, usando a técnica de encontro no meio, podemos alcançar um algoritmo de tempo $O(2^{n/2})$ mais eficiente². Observe que $O(2^n)$ e $O(2^{n/2})$ são complexidades diferentes porque $2^{n/2}$ é igual a $\sqrt{2^n}$.

A ideia é dividir a lista em duas listas A e B tais que ambas as listas contêm cerca de metade dos números. A primeira busca gera todos os subconjuntos de A e armazena suas somas em uma lista S_A . Da mesma forma, a segunda busca cria uma lista S_B a partir de B . Depois disso, basta verificar se é possível escolher um elemento de S_A e outro elemento de S_B tal que sua soma seja x . Isso é possível exatamente quando há uma maneira de formar a soma x usando os números da lista original.

Por exemplo, suponha que a lista seja $[2, 4, 5, 9]$ e $x = 15$. Primeiro, dividimos a lista em $A = [2, 4]$ e $B = [5, 9]$. Depois disso, criamos as listas $S_A = [0, 2, 4, 6]$ e $S_B = [0, 5, 9, 14]$. Neste caso, a soma $x = 15$ é possível de formar, porque S_A contém a soma 6, S_B contém a soma 9, e $6 + 9 = 15$. Isso corresponde à solução $[2, 4, 9]$.

Podemos implementar o algoritmo de modo que sua complexidade de tempo seja $O(2^{n/2})$. Primeiro, geramos listas *ordenadas* S_A e S_B , o que pode ser feito em tempo $O(2^{n/2})$ usando uma técnica semelhante à da mesclagem. Depois disso, como as listas estão ordenadas, podemos verificar em tempo $O(2^{n/2})$ se a soma x pode ser criada a partir de S_A e S_B .

²Esta ideia foi introduzida em 1974 por E. Horowitz e S. Sahni [39].

Capítulo 6

Algoritmos gulosos

Um **algoritmo guloso** constrói uma solução para o problema sempre fazendo a escolha que parece ser a melhor no momento. Um algoritmo guloso nunca volta atrás em suas escolhas, mas constrói diretamente a solução final. Por esta razão, os algoritmos gulosos são geralmente muito eficientes.

A dificuldade em projetar algoritmos gulosos está em encontrar uma estratégia gulosa que sempre produza uma solução ótima para o problema. As escolhas localmente ótimas num algoritmo guloso devem ser também globalmente ótimas. Muitas vezes é difícil argumentar que um algoritmo guloso funciona.

6.1 Problema das moedas

Como primeiro exemplo, consideramos um problema onde nos é dado um conjunto de moedas e nossa tarefa é formar uma soma de dinheiro n usando as moedas. Os valores das moedas são $\text{moedas} = \{c_1, c_2, \dots, c_k\}$, e cada moeda pode ser usada quantas vezes quisermos. Qual é o número mínimo de moedas necessárias?

Por exemplo, se as moedas forem as moedas de euro (em centavos)

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

e $n = 520$, precisamos de pelo menos quatro moedas. A solução ótima é selecionar as moedas $200 + 200 + 100 + 20$ cuja soma é 520.

Algoritmo guloso

Um algoritmo guloso simples para o problema seleciona sempre a maior moeda possível, até que a soma de dinheiro necessária seja construída. Este algoritmo funciona no caso de exemplo, porque primeiro selecionamos duas moedas de 200 centavos, depois uma moeda de 100 centavos e finalmente uma moeda de 20 centavos. Mas será que este algoritmo funciona sempre?

Acontece que se as moedas são as moedas de euro, o algoritmo guloso *sempre* funciona, i.e., ele produz sempre uma solução com o menor número possível de moedas. A correção do algoritmo pode ser mostrada da seguinte forma:

Primeiro, cada moeda de 1, 5, 10, 50 e 100 aparece no máximo uma vez numa solução ótima, porque se a solução contivesse duas moedas dessas, poderíamos

substituí-las por uma moeda e obter uma solução melhor. Por exemplo, se a solução contivesse as moedas $5 + 5$, poderíamos substituí-las pela moeda 10.

Da mesma forma, as moedas de 2 e 20 aparecem no máximo duas vezes numa solução ótima, porque poderíamos substituir as moedas $2 + 2 + 2$ pelas moedas $5 + 1$ e as moedas $20 + 20 + 20$ pelas moedas $50 + 10$. Além disso, uma solução ótima não pode conter as moedas $2 + 2 + 1$ ou $20 + 20 + 10$, porque poderíamos substituí-las pelas moedas 5 e 50.

Usando estas observações, podemos mostrar para cada moeda x que não é possível construir de forma ótima uma soma x ou qualquer soma maior usando apenas moedas que são menores que x . Por exemplo, se $x = 100$, a maior soma ótima usando as moedas menores é $50 + 20 + 20 + 5 + 2 + 2 = 99$. Assim, o algoritmo guloso que seleciona sempre a maior moeda produz a solução ótima.

Este exemplo mostra que pode ser difícil argumentar que um algoritmo guloso funciona, mesmo que o próprio algoritmo seja simples.

Caso geral

No caso geral, o conjunto de moedas pode conter quaisquer moedas e o algoritmo guloso *não* produz necessariamente uma solução ótima.

Podemos provar que um algoritmo guloso não funciona mostrando um contraexemplo onde o algoritmo dá uma resposta errada. Neste problema podemos facilmente encontrar um contraexemplo: se as moedas são $\{1, 3, 4\}$ e a soma alvo é 6, o algoritmo guloso produz a solução $4 + 1 + 1$ enquanto que a solução ótima é $3 + 3$.

Não se sabe se o problema geral das moedas pode ser resolvido usando algum algoritmo guloso¹. No entanto, como veremos no Capítulo 7, em alguns casos, o problema geral pode ser eficientemente resolvido usando um algoritmo de programação dinâmica que dá sempre a resposta correta.

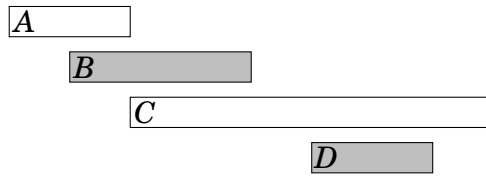
6.2 Escalonamento

Muitos problemas de escalonamento podem ser resolvidos usando algoritmos gulosos. Um problema clássico é o seguinte: Dados n eventos com seus horários de início e fim, encontre um escalonamento que inclua o maior número possível de eventos. Não é possível selecionar um evento parcialmente. Por exemplo, considere os seguintes eventos:

evento	hora de início	hora de fim
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

Neste caso, o número máximo de eventos é dois. Por exemplo, podemos selecionar os eventos *B* e *D* da seguinte forma:

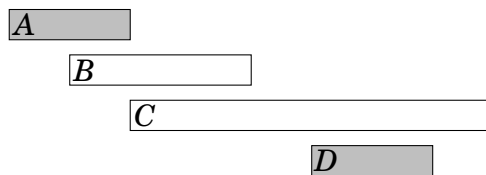
¹No entanto, é possível *verificar* em tempo polinomial se o algoritmo guloso apresentado neste capítulo funciona para um dado conjunto de moedas [53].



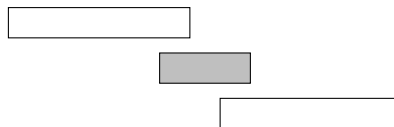
É possível inventar vários algoritmos gulosos para o problema, mas qual deles funciona em todos os casos?

Algoritmo 1

A primeira ideia é selecionar os eventos *mais curtos* possíveis. No caso de exemplo, este algoritmo seleciona os seguintes eventos:



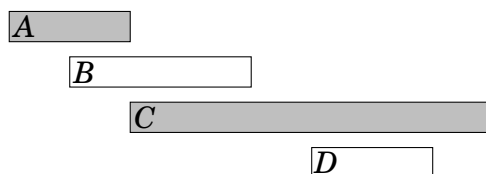
No entanto, selecionar eventos curtos nem sempre é uma estratégia correta. Por exemplo, o algoritmo falha no seguinte caso:



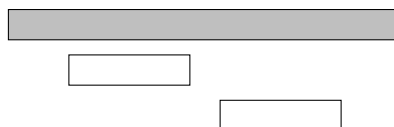
Se selecionarmos o evento curto, só podemos selecionar um evento. No entanto, seria possível selecionar ambos os eventos longos.

Algoritmo 2

Outra ideia é selecionar sempre o próximo evento possível que *começa* o mais cedo possível. Este algoritmo seleciona os seguintes eventos:



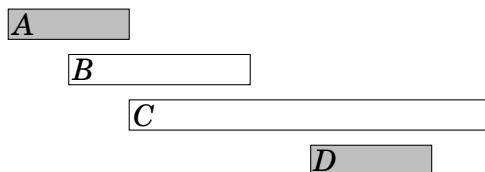
No entanto, podemos encontrar um contraexemplo também para este algoritmo. Por exemplo, no seguinte caso, o algoritmo seleciona apenas um evento:



Se selecionarmos o primeiro evento, não é possível selecionar quaisquer outros eventos. No entanto, seria possível selecionar os outros dois eventos.

Algoritmo 3

A terceira ideia é selecionar sempre o próximo evento possível que *termina* o mais cedo possível. Este algoritmo seleciona os seguintes eventos:



Acontece que este algoritmo *sempre* produz uma solução ótima. A razão para isso é que é sempre uma escolha ótima selecionar primeiro um evento que termina o mais cedo possível. Depois disso, é uma escolha ótima selecionar o próximo evento usando a mesma estratégia, etc., até não podermos selecionar mais eventos.

Uma forma de argumentar que o algoritmo funciona é considerar o que acontece se selecionarmos primeiro um evento que termina mais tarde do que o evento que termina o mais cedo possível. Agora, teremos no máximo um número igual de escolhas para selecionar o próximo evento. Portanto, selecionar um evento que termina mais tarde nunca pode resultar numa solução melhor, e o algoritmo guloso está correto.

6.3 Tarefas e prazos

Vamos agora considerar um problema onde nos são dadas n tarefas com durações e prazos e nossa tarefa é escolher uma ordem para realizar as tarefas. Para cada tarefa, ganhamos $d - x$ pontos onde d é o prazo da tarefa e x é o momento em que terminamos a tarefa. Qual é a maior pontuação total possível que podemos obter?

Por exemplo, suponha que as tarefas são as seguintes:

tarefa	duração	prazo
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

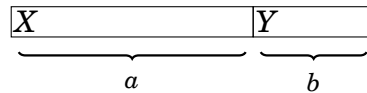
Neste caso, um escalonamento ótimo para as tarefas é o seguinte:



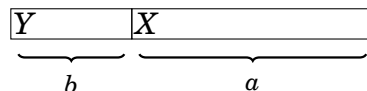
Nesta solução, *C* rende 5 pontos, *B* rende 0 pontos, *A* rende -7 pontos e *D* rende -8 pontos, então a pontuação total é -10 .

Surpreendentemente, a solução ótima para o problema sequer depende dos prazos, uma estratégia gulosa correta é simplesmente executar as tarefas *ordenadas por suas durações* em ordem crescente. A razão para isso é que se alguma vez

executarmos duas tarefas, uma após a outra, de tal forma que a primeira tarefa demore mais tempo do que a segunda tarefa, podemos obter uma solução melhor se trocarmos as tarefas. Por exemplo, considere o seguinte escalonamento:



Aqui $a > b$, então devemos trocar as tarefas:



Agora X dá b pontos a menos e Y dá a pontos a mais, então a pontuação total aumenta em $a - b > 0$. Numa solução ótima, para quaisquer duas tarefas consecutivas, deve verificar-se que a tarefa mais curta vem antes da tarefa mais longa. Assim, as tarefas devem ser executadas ordenadas pelas suas durações.

6.4 Minimizando somas

Consideramos agora um problema onde nos são dados n números a_1, a_2, \dots, a_n e nossa tarefa é encontrar um valor x que minimize a soma

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

Vamos focar nos casos $c = 1$ e $c = 2$.

Caso $c = 1$

Neste caso, devemos minimizar a soma

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

Por exemplo, se os números são $[1, 2, 9, 2, 6]$, a melhor solução é selecionar $x = 2$ o que produz a soma

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

No caso geral, a melhor escolha para x é a *mediana* dos números, i.e., o número do meio após a ordenação. Por exemplo, a lista $[1, 2, 9, 2, 6]$ torna-se $[1, 2, 2, 6, 9]$ após a ordenação, então a mediana é 2.

A mediana é uma escolha ótima, porque se x é menor que a mediana, a soma torna-se menor ao aumentar x , e se x é maior que a mediana, a soma torna-se menor ao diminuir x . Portanto, a solução ótima é que x seja a mediana. Se n é par e existem duas medianas, ambas as medianas e todos os valores entre elas são escolhas ótimas.

Caso $c = 2$

Neste caso, devemos minimizar a soma

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

Por exemplo, se os números são $[1, 2, 9, 2, 6]$, a melhor solução é selecionar $x = 4$ o que produz a soma

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

No caso geral, a melhor escolha para x é a *média* dos números. No exemplo, a média é $(1 + 2 + 9 + 2 + 6)/5 = 4$. Este resultado pode ser derivado apresentando a soma da seguinte forma:

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

A última parte não depende de x , portanto podemos ignorá-la. As partes restantes formam uma função $nx^2 - 2xs$ onde $s = a_1 + a_2 + \cdots + a_n$. Esta é uma parábola com a concavidade voltada para cima com raízes $x = 0$ e $x = 2s/n$, e o valor mínimo é a média das raízes $x = s/n$, i.e., a média dos números a_1, a_2, \dots, a_n .

6.5 Compressão de dados

Um **código binário** atribui para cada caractere de uma string uma **palavra de código** que consiste em bits. Podemos *comprimir* a string usando o código binário substituindo cada caractere pela palavra de código correspondente. Por exemplo, o seguinte código binário atribui palavras de código para os caracteres A–D:

caractere	palavra de código
A	00
B	01
C	10
D	11

Este é um código de **comprimento constante** o que significa que o comprimento de cada palavra de código é o mesmo. Por exemplo, podemos comprimir a string AABACDACA da seguinte forma:

000001001011001000

Usando este código, o comprimento da string comprimida é de 18 bits. No entanto, podemos comprimir a string melhor se usarmos um código de **comprimento variável** onde as palavras de código podem ter comprimentos diferentes. Então podemos dar palavras de código curtas para caracteres que aparecem frequentemente e palavras de código longas para caracteres que aparecem raramente. Acontece que um código **ótimo** para a string acima é o seguinte:

caractere	palavra de código
A	0
B	110
C	10
D	111

Um código ótimo produz uma string comprimida que é o mais curta possível. Neste caso, a string comprimida usando o código ótimo é

001100101110100,

então são necessários apenas 15 bits em vez de 18 bits. Assim, graças a um código melhor, foi possível poupar 3 bits na string comprimida.

Exigimos que nenhuma palavra de código seja um prefixo de outra palavra de código. Por exemplo, não é permitido que um código contenha ambas as palavras de código 10 e 1011. A razão para isso é que queremos ser capazes de gerar a string original a partir da string comprimida. Se uma palavra de código pudesse ser um prefixo de outra palavra de código, isso nem sempre seria possível. Por exemplo, o seguinte código *não* é válido:

caractere	palavra de código
A	10
B	11
C	1011
D	111

Usando este código, não seria possível saber se a string comprimida 1011 corresponde à string AB ou à string C.

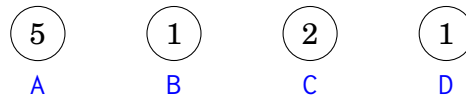
Codificação de Huffman

Codificação de Huffman² é um algoritmo guloso que constrói um código ótimo para comprimir uma dada string. O algoritmo constrói uma árvore binária com base nas frequências dos caracteres na string, e a palavra de código de cada caractere pode ser lida seguindo um caminho desde a raiz até ao nó correspondente. Um movimento para a esquerda corresponde ao bit 0, e um movimento para a direita corresponde ao bit 1.

Inicialmente, cada caractere da string é representado por um nó cujo peso é o número de vezes que o caractere ocorre na string. Então, em cada passo, dois nós com pesos mínimos são combinados criando um novo nó cujo peso é a soma dos pesos dos nós originais. O processo continua até que todos os nós tenham sido combinados.

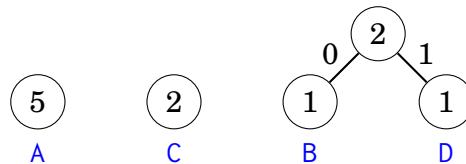
A seguir, veremos como a Codificação de Huffman cria o código ótimo para a string AABACDACA. Inicialmente, existem quatro nós que correspondem aos caracteres da string:

²D. A. Huffman descobriu este método ao resolver um trabalho de um curso universitário e publicou o algoritmo em 1952 [40].

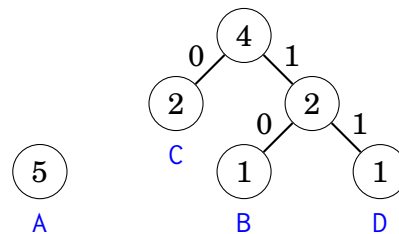


O nó que representa o caractere A tem peso 5 porque o caractere A aparece 5 vezes na string. Os outros pesos foram calculados da mesma forma.

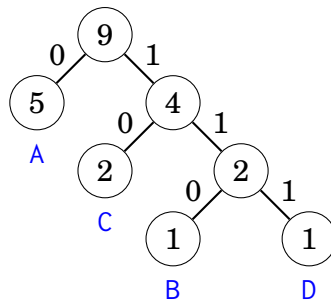
O primeiro passo é combinar os nós que correspondem aos caracteres B e D, ambos com peso 1. O resultado é:



Depois disso, os nós com peso 2 são combinados:



Finalmente, os dois nós restantes são combinados:



Agora todos os nós estão na árvore, então o código está pronto. As seguintes palavras de código podem ser lidas a partir da árvore:

caractere	palavra de código
A	0
B	110
C	10
D	111

Capítulo 7

Programação dinâmica

Programação dinâmica é uma técnica que combina a corretude da busca completa com a eficiência dos algoritmos gulosos. A programação dinâmica pode ser aplicada se o problema puder ser dividido em subproblemas sobrepostos que podem ser resolvidos independentemente.

Existem dois usos para a programação dinâmica:

- **Encontrar uma solução ótima:** Queremos encontrar uma solução que seja a maior possível ou a menor possível.
- **Contar o número de soluções:** Queremos calcular o número total de soluções possíveis.

Veremos primeiro como a programação dinâmica pode ser usada para encontrar uma solução ótima, e então usaremos a mesma ideia para contar as soluções.

Entender a programação dinâmica é um marco na carreira de todo programador competitivo. Enquanto a ideia básica é simples, o desafio é como aplicar a programação dinâmica a diferentes problemas. Este capítulo apresenta um conjunto de problemas clássicos que são um bom ponto de partida.

7.1 Problema das moedas

Vamos primeiro nos concentrar em um problema que já vimos no Capítulo 6: Dado um conjunto de valores de moedas $\text{moedas} = \{c_1, c_2, \dots, c_k\}$ e uma soma alvo de dinheiro n , nossa tarefa é formar a soma n usando o menor número possível de moedas.

No Capítulo 6, resolvemos o problema usando um algoritmo guloso que sempre escolhe a maior moeda possível. O algoritmo guloso funciona, por exemplo, quando as moedas são as moedas de euro, mas no caso geral, o algoritmo guloso não produz necessariamente uma solução ótima.

Agora é hora de resolver o problema de forma eficiente usando programação dinâmica, para que o algoritmo funcione para qualquer conjunto de moedas. O algoritmo de programação dinâmica é baseado em uma função recursiva que percorre todas as possibilidades de como formar a soma, como um algoritmo de

força bruta. No entanto, o algoritmo de programação dinâmica é eficiente porque usa *memoização* e calcula a resposta para cada subproblema apenas uma vez.

Formulação recursiva

A ideia da programação dinâmica é formular o problema recursivamente para que a solução do problema possa ser calculada a partir de soluções para subproblemas menores. No problema das moedas, um problema recursivo natural é o seguinte: qual é o menor número de moedas necessário para formar uma soma x ?

Seja $\text{resolver}(x)$ o número mínimo de moedas necessárias para uma soma x . Os valores da função dependem dos valores das moedas. Por exemplo, se $\text{moedas} = \{1, 3, 4\}$, os primeiros valores da função são os seguintes:

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

Por exemplo, $\text{resolver}(10) = 3$, porque pelo menos 3 moedas são necessárias para formar a soma 10. A solução ótima é $3 + 3 + 4 = 10$.

A propriedade essencial de resolver é que seus valores podem ser calculados recursivamente a partir de seus valores menores. A ideia é focar na *primeira* moeda que escolhemos para a soma. Por exemplo, no cenário acima, a primeira moeda pode ser 1, 3 ou 4. Se escolhermos primeiro a moeda 1, a tarefa restante é formar a soma 9 usando o número mínimo de moedas, que é um subproblema do problema original. Claro, o mesmo se aplica às moedas 3 e 4. Assim, podemos usar a seguinte fórmula recursiva para calcular o número mínimo de moedas:

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

O caso base da recursão é $\text{resolver}(0) = 0$, porque nenhuma moeda é necessária para formar uma soma vazia. Por exemplo,

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

Agora estamos prontos para fornecer uma função recursiva geral que calcula o número mínimo de moedas necessárias para formar uma soma x :

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

Primeiro, se $x < 0$, o valor é ∞ , porque é impossível formar uma soma negativa de dinheiro. Então, se $x = 0$, o valor é 0, porque nenhuma moeda é necessária para formar uma soma vazia. Finalmente, se $x > 0$, a variável c percorre todas as possibilidades de como escolher a primeira moeda da soma.

Uma vez encontrada uma função recursiva que resolve o problema, podemos implementar diretamente uma solução em C++ (a constante INF denota infinito):

```
int resolver(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int melhor = INF;
    for (auto c : moedas) {
        melhor = min(melhor, resolver(x-c)+1);
    }
    return melhor;
}
```

Ainda assim, esta função não é eficiente, porque pode haver um número exponencial de maneiras de construir a soma. No entanto, a seguir veremos como tornar a função eficiente usando uma técnica chamada memoização.

Usando memoização

A ideia da programação dinâmica é usar **memoização** para calcular eficientemente valores de uma função recursiva. Isso significa que os valores da função são armazenados em um array após o cálculo. Para cada parâmetro, o valor da função é calculado recursivamente apenas uma vez e, depois disso, o valor pode ser recuperado diretamente do array.

Neste problema, usamos arrays

```
bool pronto[N];
int valor[N];
```

onde `pronto[x]` indica se o valor de `resolver(x)` foi calculado, e se foi, `valor[x]` contém esse valor. A constante N foi escolhida de forma que todos os valores necessários caibam nos arrays.

Agora a função pode ser eficientemente implementada da seguinte forma:

```
int resolver(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (pronto[x]) return valor[x];
    int melhor = INF;
    for (auto c : moedas) {
        melhor = min(melhor, resolver(x-c)+1);
    }
    valor[x] = melhor;
    pronto[x] = true;
}
```

```
    return melhor;
}
```

A função lida com os casos base $x < 0$ e $x = 0$ como anteriormente. Então a função verifica em `pronto[x]` se `resolver(x)` já foi armazenado em `valor[x]`, e se foi, a função o retorna diretamente. Caso contrário, a função calcula o valor de `resolver(x)` recursivamente e o armazena em `valor[x]`.

Esta função funciona de forma eficiente, porque a resposta para cada parâmetro x é calculada recursivamente apenas uma vez. Depois que um valor de `resolver(x)` é armazenado em `valor[x]`, ele pode ser recuperado de forma eficiente sempre que a função for chamada novamente com o parâmetro x . A complexidade de tempo do algoritmo é $O(nk)$, onde n é a soma alvo e k é o número de moedas.

Observe que também podemos construir o array `valor` *iterativamente* usando um loop que simplesmente calcula todos os valores de `resolver` para os parâmetros $0 \dots n$:

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

Na verdade, a maioria dos programadores competitivos prefere esta implementação, porque é mais curta e tem menores fatores constantes. De agora em diante, também usaremos implementações iterativas em nossos exemplos. Ainda assim, geralmente é mais fácil pensar em soluções de programação dinâmica em termos de funções recursivas.

Construindo uma solução

Às vezes, somos solicitados a encontrar o valor de uma solução ótima e dar um exemplo de como essa solução pode ser construída. No problema das moedas, por exemplo, podemos declarar outro array que indica para cada soma de dinheiro a primeira moeda em uma solução ótima:

```
int primeiro[N];
```

Então, podemos modificar o algoritmo da seguinte forma:

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

```

        valor[x] = valor[x-c]+1;
        primeiro[x] = c;
    }
}
}

```

Depois disso, o seguinte código pode ser usado para imprimir as moedas que aparecem em uma solução ótima para a soma n :

```

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

```

Contando o número de soluções

Vamos agora considerar outra versão do problema das moedas, onde nossa tarefa é calcular o número total de maneiras de produzir uma soma x usando as moedas. Por exemplo, se moedas = {1,3,4} e $x = 5$, existem um total de 6 maneiras:

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

Novamente, podemos resolver o problema recursivamente. Seja $\text{resolver}(x)$ o número de maneiras de formarmos a soma x . Por exemplo, se moedas = {1,3,4}, então $\text{resolver}(5) = 6$ e a fórmula recursiva é

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

Então, a função recursiva geral é a seguinte:

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

Se $x < 0$, o valor é 0, porque não há soluções. Se $x = 0$, o valor é 1, porque só há uma maneira de formar uma soma vazia. Caso contrário, calculamos a soma de todos os valores da forma $\text{resolver}(x-c)$ onde c está em moedas.

O código a seguir constrói um array contagem tal que $\text{contagem}[x]$ é igual ao valor de $\text{resolver}(x)$ para $0 \leq x \leq n$:

```

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

```

```

for (auto c : moedas) {
    if (x-c >= 0) {
        contagem[x] += contagem[x-c];
    }
}

```

Muitas vezes, o número de soluções é tão grande que não é necessário calcular o número exato, mas é suficiente fornecer a resposta módulo m onde, por exemplo, $m = 10^9 + 7$. Isso pode ser feito alterando o código para que todos os cálculos sejam feitos módulo m . No código acima, basta adicionar a linha

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

após a linha

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

Agora discutimos todas as ideias básicas da programação dinâmica. Como a programação dinâmica pode ser usada em muitas situações diferentes, vamos agora passar por um conjunto de problemas que mostram outros exemplos sobre as possibilidades da programação dinâmica.


7.2 Maior subsequência crescente

Nosso primeiro problema é encontrar a **maior subsequência crescente** em um array de n elementos. Esta é uma sequência de comprimento máximo de elementos do array que vai da esquerda para a direita, e cada elemento na sequência é maior que o elemento anterior. Por exemplo, no array

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

a maior subsequência crescente contém 4 elementos:

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



Seja $\text{tamanho}(k)$ o comprimento da maior subsequência crescente que termina na posição k . Assim, se calcularmos todos os valores de $\text{tamanho}(k)$ onde $0 \leq k \leq n - 1$, descobriremos o comprimento da maior subsequência crescente. Por

exemplo, os valores da função para o array acima são os seguintes:

tamanho(0)	=	1
tamanho(1)	=	1
tamanho(2)	=	2
tamanho(3)	=	1
tamanho(4)	=	3
tamanho(5)	=	2
tamanho(6)	=	4
tamanho(7)	=	2

Por exemplo, $\text{tamanho}(6) = 4$, porque a maior subsequência crescente que termina na posição 6 consiste em 4 elementos.

Para calcular um valor de $\text{tamanho}(k)$, devemos encontrar uma posição $i < k$ para a qual $\text{array}[i] < \text{array}[k]$ e $\text{tamanho}(i)$ seja o maior possível. Então sabemos que $\text{tamanho}(k) = \text{tamanho}(i) + 1$, porque esta é uma maneira ótima de adicionar $\text{array}[k]$ a uma subsequência. No entanto, se não houver tal posição i , então $\text{tamanho}(k) = 1$, o que significa que a subsequência contém apenas $\text{array}[k]$.

Como todos os valores da função podem ser calculados a partir de seus valores menores, podemos usar programação dinâmica. No código a seguir, os valores da função serão armazenados em um array `tamanho`.

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

Este código funciona em tempo $O(n^2)$, porque consiste em dois loops aninhados. No entanto, também é possível implementar o cálculo de programação dinâmica de forma mais eficiente em tempo $O(n \log n)$. Você consegue encontrar uma maneira de fazer isso?

7.3 Caminhos em uma grade

Nosso próximo problema é encontrar um caminho do canto superior esquerdo para o canto inferior direito de uma grade $n \times n$, de forma que só nos movamos para baixo e para a direita. Cada quadrado contém um inteiro positivo, e o caminho deve ser construído de forma que a soma dos valores ao longo do caminho seja a maior possível.

A figura a seguir mostra um caminho ideal em uma grade:

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

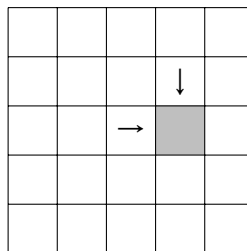
A soma dos valores no caminho é 67, e esta é a maior soma possível em um caminho do canto superior esquerdo para o canto inferior direito.

Assuma que as linhas e colunas da grade são numeradas de 1 a n , e $\text{valor}[y][x]$ é igual ao valor do quadrado (y, x) . Seja $\text{soma}(y, x)$ a soma máxima em um caminho do canto superior esquerdo para o quadrado (y, x) . Agora $\text{soma}(n, n)$ nos diz a soma máxima do canto superior esquerdo para o canto inferior direito. Por exemplo, na grade acima, $\text{soma}(5, 5) = 67$.

Podemos calcular recursivamente as somas da seguinte forma:

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

A fórmula recursiva é baseada na observação de que um caminho que termina no quadrado (y, x) pode vir do quadrado $(y, x-1)$ ou do quadrado $(y-1, x)$:



Assim, selecionamos a direção que maximiza a soma. Assumimos que $\text{soma}(y, x) = 0$ se $y = 0$ ou $x = 0$ (porque tais caminhos não existem), então a fórmula recursiva também funciona quando $y = 1$ ou $x = 1$.

Como a função soma possui dois parâmetros, o array de programação dinâmica também possui duas dimensões. Por exemplo, podemos usar um array

```
int soma[N][N];
```

e calcular as somas da seguinte forma:

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

A complexidade de tempo do algoritmo é $O(n^2)$.

7.4 Problemas da mochila

O termo **mochila** refere-se a problemas onde um conjunto de objetos é dado, e subconjuntos com algumas propriedades precisam ser encontrados. Os problemas da mochila podem ser resolvidos usando programação dinâmica.

Nesta seção, vamos focar no seguinte problema: Dada uma lista de pesos $[w_1, w_2, \dots, w_n]$, determine todas as somas que podem ser construídas usando os pesos. Por exemplo, se os pesos forem $[1, 3, 3, 5]$, as seguintes somas são possíveis:

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

Neste caso, todas as somas entre $0 \dots 12$ são possíveis, exceto 2 e 10. Por exemplo, a soma 7 é possível porque nós podemos selecionar os pesos $[1, 3, 3]$.

Para resolver o problema, vamos focar em subproblemas onde usamos apenas os primeiros k pesos para construir somas. Seja $\text{possivel}(x, k) = \text{verdadeiro}$ se podemos construir uma soma x usando os primeiros k pesos, e caso contrário $\text{possivel}(x, k) = \text{falso}$. Os valores da função podem ser calculados recursivamente da seguinte forma:

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

A fórmula é baseada no fato de que podemos usar ou não o peso w_k na soma. Se usarmos w_k , a tarefa restante é formar a soma $x - w_k$ usando os primeiros $k - 1$ pesos, e se não usarmos w_k , a tarefa restante é formar a soma x usando os primeiros $k - 1$ pesos. Como casos base,

$$\text{possivel}(x, 0) = \begin{cases} \text{verdadeiro} & x = 0 \\ \text{falso} & x \neq 0 \end{cases}$$

porque se nenhum peso for usado, só podemos formar a soma 0.

A tabela a seguir mostra todos os valores da função para os pesos $[1, 3, 3, 5]$ (o símbolo "X" indica os valores verdadeiros):

$k \backslash x$	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

Após calcular esses valores, $\text{possivel}(x, n)$ nos diz se podemos construir uma soma x usando *todos* os pesos.

Seja W a soma total dos pesos. A seguinte solução de programação dinâmica em tempo $O(nW)$ corresponde à função recursiva:

```
possivel[0][0] = true;
for (int k = 1; k <= n; k++) {
```

```

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

```

No entanto, aqui está uma implementação melhor que usa apenas um array unidimensional `possivel[x]` que indica se podemos construir um subconjunto com soma x . O truque é atualizar o array da direita para a esquerda para cada novo peso:

```

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

```

Observe que a ideia geral apresentada aqui pode ser usada em muitos problemas de mochila. Por exemplo, se recebermos objetos com pesos e valores, podemos determinar para cada soma de peso o valor máximo soma de um subconjunto.

7.5 Distância de edição

A **distância de edição** ou **distância de Levenshtein**¹ é o número mínimo de operações de edição necessárias para transformar uma string em outra. As operações de edição permitidas são as seguintes:

- inserir um caractere (por exemplo, $ABC \rightarrow ABCA$)
- remover um caractere (por exemplo, $ABC \rightarrow AC$)
- modificar um caractere (por exemplo, $ABC \rightarrow ADC$)

Por exemplo, a distância de edição entre LOVE e MOVIE é 2, porque podemos primeiro realizar a operação $LOVE \rightarrow MOVE$ (modificar) e então a operação $MOVE \rightarrow MOVIE$ (inserir). Este é o menor número possível de operações, porque é claro que apenas uma operação não é suficiente.

Suponha que recebemos uma string x de comprimento n e uma string y de comprimento m , e queremos calcular a distância de edição entre x e y . Para resolver o problema, definimos uma função $distancia(a, b)$ que retorna a distância de edição entre os prefixos $x[0 \dots a]$ e $y[0 \dots b]$. Assim, usando esta função, a distância de edição entre x e y é igual a $distancia(n-1, m-1)$.

¹A distância recebe o nome de V. I. Levenshtein, que a estudou em conexão com códigos binários [49].

Podemos calcular os valores de distancia da seguinte forma:

$$\begin{aligned} \text{distancia}(a, b) = \min(\text{distancia}(a, b-1) + 1, \\ \text{distancia}(a-1, b) + 1, \\ \text{distancia}(a-1, b-1) + \text{custo}(a, b)). \end{aligned}$$

Aqui $\text{custo}(a, b) = 0$ se $x[a] = y[b]$, e caso contrário $\text{custo}(a, b) = 1$. A fórmula considera as seguintes maneiras de editar a string x:

- $\text{distancia}(a, b-1)$: inserir um caractere no final de x
- $\text{distancia}(a-1, b)$: remover o último caractere de x
- $\text{distancia}(a-1, b-1)$: combinar ou modificar o último caractere de x

Nos dois primeiros casos, uma operação de edição é necessária (inserir ou remover). No último caso, se $x[a] = y[b]$, podemos combinar os últimos caracteres sem editar, e caso contrário, uma operação de edição é necessária (modificar).

A tabela a seguir mostra os valores de distancia no caso de exemplo:

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

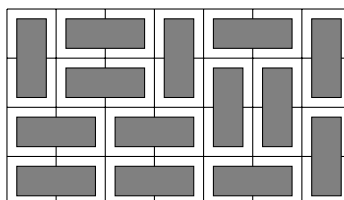
O canto inferior direito da tabela nos diz que a distância de edição entre LOVE e MOVIE é 2. A tabela também mostra como construir a menor sequência de operações de edição. Neste caso, o caminho é o seguinte:

		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

Os últimos caracteres de LOVE e MOVIE são iguais, então a distância de edição entre eles é igual à distância de edição entre LOV e MOVI. Podemos usar uma operação de edição para remover o caractere I de MOVI. Assim, a distância de edição é um a mais que a distância de edição entre LOV e MOV, etc.

7.6 Contando os ladrilhos

Às vezes, os estados de uma solução de programação dinâmica são mais complexos do que combinações fixas de números. Como exemplo, considere o problema de calcular o número de maneiras distintas de preencher uma grade $n \times m$ usando ladrilhos de tamanho 1×2 e 2×1 . Por exemplo, uma solução válida para a grade 4×7 é



e o número total de soluções é 781.

O problema pode ser resolvido usando programação dinâmica percorrendo a grade linha por linha. Cada linha em uma solução pode ser representada como uma string que contém m caracteres do conjunto $\{\sqcup, \sqcup, \sqcup, \sqcup\}$. Por exemplo, a solução acima consiste em quatro linhas que correspondem às seguintes strings:

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

Seja $\text{contar}(k, x)$ o número de maneiras de construir uma solução para as linhas $1 \dots k$ da grade, de modo que a string x corresponda à linha k . É possível usar a programação dinâmica aqui, porque o estado de uma linha é restringido apenas pelo estado da linha anterior.

Uma solução é válida se a linha 1 não contiver o caractere \sqcup , a linha n não contiver o caractere \sqcup , e todas as linhas consecutivas forem *compatíveis*. Por exemplo, as linhas $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ e $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ são compatíveis, enquanto as linhas $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ e $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ não são compatíveis.

Como uma linha consiste em m caracteres e existem quatro opções para cada caractere, o número de linhas distintas é no máximo 4^m . Assim, a complexidade de tempo da solução é $O(n4^{2m})$ porque podemos percorrer os $O(4^m)$ estados possíveis para cada linha, e para cada estado, existem $O(4^m)$ estados possíveis para a linha anterior. Na prática, é uma boa ideia girar a grade para que o lado mais curto tenha comprimento m , porque o fator 4^{2m} domina a complexidade de tempo.

É possível tornar a solução mais eficiente usando uma representação mais compacta para as linhas. Acontece que é suficiente saber quais colunas da linha anterior contêm o quadrado superior de um ladrilho vertical. Assim, podemos representar uma linha usando apenas caracteres \sqcup e \sqcup , onde \sqcup é uma combinação de caracteres \sqcup , \sqcup e \sqcup . Usando esta representação, existem apenas 2^m linhas distintas e a complexidade de tempo é $O(n2^{2m})$.

Como nota final, há também uma fórmula direta surpreendente para calcular o número de ladrilhos²:

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

Esta fórmula é muito eficiente, pois calcula o número de ladrilhos em tempo $O(nm)$, mas como a resposta é um produto de números reais, um problema ao usar a fórmula é como armazenar os resultados intermediários com precisão.

²Surpreendentemente, esta fórmula foi descoberta em 1961 por duas equipes de pesquisa [43, 67] que trabalharam de forma independente.

Capítulo 8

Análise amortizada

A complexidade de tempo de um algoritmo é, muitas vezes, fácil de analisar apenas examinando a estrutura do algoritmo: quais laços o algoritmo contém e quantas vezes os laços são executados. No entanto, às vezes, uma análise direta não fornece uma imagem verdadeira da eficiência do algoritmo.

A **análise amortizada** pode ser usada para analisar algoritmos que contêm operações cuja complexidade de tempo varia. A ideia é estimar o tempo total usado para todas essas operações durante a execução do algoritmo, em vez de focar em operações individuais.

8.1 Método dos dois ponteiros

No **método dos dois ponteiros**, dois ponteiros são usadas para iterar pelos valores do vetor. Ambos os ponteiros podem se mover para uma única direção, o que garante que o algoritmo funcione de forma eficiente. A seguir, discutiremos dois problemas que podem ser resolvidos usando o método de dois ponteiros.

Soma de subvetor

Como primeiro exemplo, considere um problema em que recebemos um vetor de n inteiros positivos e uma soma alvo x , e queremos encontrar um subvetor cuja soma seja x ou relatar que não existe tal subvetor.

Por exemplo, o vetor

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

contém um subvetor cuja soma é 8:

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

Este problema pode ser resolvido em tempo $O(n)$ usando o método de dois ponteiros. A ideia é manter ponteiros que apontam para o primeiro e o último valor de um subvetor. A cada turno, o ponteiro esquerdo se move um passo para a direita e o ponteiro direito se move para a direita enquanto a soma do subvetor

resultante for no máximo x . Se a soma se tornar exatamente x , uma solução foi encontrada.

Como exemplo, considere o seguinte vetor e uma soma alvo $x = 8$:

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

O subvetor inicial contém os valores 1, 3 e 2, cuja soma é 6:

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

↑ ↑

Então, o ponteiro esquerdo se move um passo para a direita. O ponteiro direito não se move, porque, caso contrário, a soma do subvetor excederia x .

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

↑ ↑

Novamente, o ponteiro esquerdo se move um passo para a direita, e desta vez o ponteiro direito se move três passos para a direita. A soma do subvetor é $2 + 5 + 1 = 8$, então um subvetor cuja soma é x foi encontrado.

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

 ↑ ↑

O tempo de execução do algoritmo depende de o número de passos que o ponteiro direito se move. Embora não haja um limite superior útil sobre quantos passos o ponteiro pode se mover em um *único* turno, sabemos que o ponteiro se move *um total de* $O(n)$ passos durante o algoritmo, porque ela só se move para a direita.

Como o ponteiro esquerdo e o direito se movem $O(n)$ passos durante o algoritmo, o algoritmo funciona em tempo $O(n)$.

Problema 2SUM

Outro problema que pode ser resolvido usando o método de dois ponteiros é o seguinte problema, também conhecido como o **problema 2SUM**: dado um vetor de n números e uma soma alvo x , encontre dois valores do vetor de forma que sua soma seja x , ou relate que tais valores não existem.

Para resolver o problema, primeiro ordenamos os valores do vetor em ordem crescente. Depois disso, iteramos pelo vetor usando dois ponteiros. O ponteiro esquerdo começa no primeiro valor e se move um passo para a direita a cada turno. O ponteiro direito começa no último valor e sempre se move para a esquerda até que a soma do ponteiro esquerdo e direito seja no máximo x . Se a soma for exatamente x , uma solução foi encontrada.

Por exemplo, considere o seguinte vetor e uma soma alvo $x = 12$:

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

As posições iniciais dos ponteiros são como se segue. A soma dos valores é $1 + 10 = 11$ que é menor que x .

1	4	5	6	7	9	9	10
↑							↑

Então o ponteiro esquerdo se move um passo para a direita. O ponteiro direito se move três passos para a esquerda, e a soma se torna $4 + 7 = 11$.

1	4	5	6	7	9	9	10
	↑			↑			

Depois disso, o ponteiro esquerdo se move um passo para a direita novamente. O ponteiro direito não se move e uma solução $5 + 7 = 12$ foi encontrada.

1	4	5	6	7	9	9	10
		↑		↑			

O tempo de execução do algoritmo é $O(n \log n)$, pois primeiro ele ordena o vetor em tempo $O(n \log n)$, e então ambos os ponteiros movem $O(n)$ passos.

Observe que é possível resolver o problema de outra forma em tempo $O(n \log n)$ usando a busca binária. Nessa solução, iteramos pelo vetor e, para cada valor do vetor, tentamos encontrar outro valor que produza a soma x . Isso pode ser feito realizando n buscas binárias, cada uma das quais leva tempo $O(\log n)$.

Um problema mais difícil é o **problema 3SUM** que pede para encontrar *três* valores de vetor cuja soma seja x . Usando a ideia do algoritmo acima, este problema pode ser resolvido em tempo $O(n^2)$ ¹. Você consegue ver como?

8.2 Elementos menores mais próximos

A análise amortizada é frequentemente usada para estimar o número de operações realizadas em uma estrutura de dados. As operações podem ser distribuídas de forma desigual, de modo que a maioria das operações ocorra durante uma determinada fase do algoritmo, mas o número total de operações é limitado.

Como exemplo, considere o problema de encontrar para cada elemento do vetor o **elemento menor mais próximo**, ou seja, o primeiro elemento menor que precede o elemento no vetor. É possível que tal elemento não exista, caso em que o algoritmo deve relatar isso. A seguir, veremos como o problema pode ser resolvido de forma eficiente usando uma estrutura de pilha.

Percorremos o vetor da esquerda para a direita e mantemos uma pilha de elementos do vetor. Em cada posição do vetor, removemos elementos da pilha até

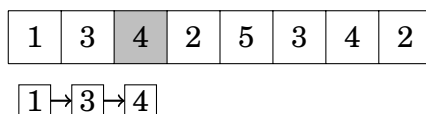
¹Por muito tempo, pensou-se que resolver o problema 3SUM de forma mais eficiente do que em tempo $O(n^2)$ não seria possível. No entanto, em 2014, descobriu-se [30] que este não é o caso.

que o elemento superior seja menor que o elemento atual, ou a pilha esteja vazia. Então, relatamos que o elemento superior é o elemento menor mais próximo do elemento atual, ou se a pilha estiver vazia, não existe tal elemento. Finalmente, adicionamos o elemento atual à pilha.

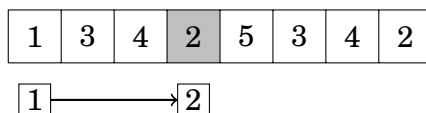
Como exemplo, considere o seguinte vetor:

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

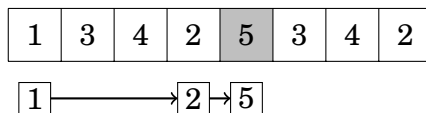
Primeiro, os elementos 1, 3 e 4 são adicionados à pilha, porque cada elemento é maior que o elemento anterior. Assim, o elemento menor mais próximo de 4 é 3, e o elemento menor mais próximo de 3 é 1.



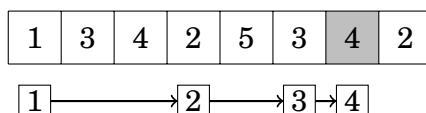
O próximo elemento 2 é menor que os dois principais elementos na pilha. Assim, os elementos 3 e 4 são removidos da pilha, e então o elemento 2 é adicionado à pilha. Seu elemento menor mais próximo é 1:



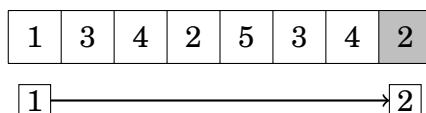
Então, o elemento 5 é maior que o elemento 2, então ele será adicionado à pilha, e seu elemento menor mais próximo é 2:



Depois disso, o elemento 5 é removido da pilha e os elementos 3 e 4 são adicionados à pilha:



Finalmente, todos os elementos, exceto 1, são removidos da pilha e o último elemento 2 é adicionado à pilha:



A eficiência do algoritmo depende de o número total de operações de pilha. Se o elemento atual for maior que o elemento superior na pilha, ele é diretamente adicionado à pilha, o que é eficiente. No entanto, às vezes a pilha pode conter vários elementos maiores e leva tempo para removê-los. Ainda assim, cada elemento é adicionado *exatamente uma vez* à pilha e removido *no máximo uma vez* da pilha. Assim, cada elemento causa $O(1)$ operações de pilha, e o algoritmo funciona em tempo $O(n)$.

8.3 Mínimo da janela deslizante

Uma **janela deslizante** é um subvetor de tamanho constante que se move da esquerda para a direita através do vetor. Em cada posição da janela, queremos calcular alguma informação sobre os elementos dentro da janela. Nesta seção, vamos focar no problema de manter o **mínimo da janela deslizante**, o que significa que devemos reportar o menor valor dentro de cada janela.

O mínimo da janela deslizante pode ser calculado usando uma ideia semelhante à que usamos para calcular os elementos menores mais próximos. Mantemos uma fila onde cada elemento é maior que o elemento anterior, e o primeiro elemento sempre corresponde ao elemento mínimo dentro da janela. Após cada movimento da janela, removemos elementos do final da fila até que o último elemento da fila seja menor que o novo elemento da janela, ou a fila fique vazia. Também removemos o primeiro elemento da fila se ele não estiver mais dentro da janela. Finalmente, adicionamos o novo elemento da janela ao final da fila.

Como exemplo, considere o seguinte vetor:

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

Suponha que o tamanho da janela deslizante seja 4. Na primeira posição da janela, o menor valor é 1:

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

1	→	4	→	5
---	---	---	---	---

Então a janela se move um passo para a direita. O novo elemento 3 é menor que os elementos 4 e 5 na fila, então os elementos 4 e 5 são removidos da fila e o elemento 3 é adicionado à fila. O menor valor ainda é 1.

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

1	→	3
---	---	---

Depois disso, a janela se move novamente, e o menor elemento 1 não pertence mais à janela. Assim, ele é removido da fila e o menor valor agora é 3. Além disso, o novo elemento 4 é adicionado à fila.

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

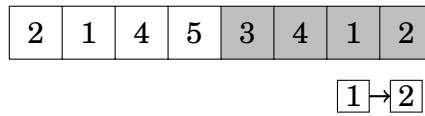
3	→	4
---	---	---

O próximo novo elemento 1 é menor que todos os elementos na fila. Assim, todos os elementos são removidos da fila e ela conterá apenas o elemento 1:

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

1

Finalmente, a janela atinge sua última posição. O elemento 2 é adicionado à fila, mas o menor valor dentro da janela ainda é 1.



Como cada elemento do vetor é adicionado à fila exatamente uma vez e removido da fila no máximo uma vez, o algoritmo funciona em tempo $O(n)$.

Capítulo 9

Consultas de intervalo

Neste capítulo, discutimos estruturas de dados que nos permitem processar eficientemente consultas de intervalo. Em uma **consulta de intervalo**, nossa tarefa é calcular um valor com base em um subvetor de um vetor. Consultas de intervalo típicas são:

- $\text{soma}_q(a, b)$: calcular a soma dos valores no intervalo $[a, b]$
- $\text{min}_q(a, b)$: encontrar o valor mínimo no intervalo $[a, b]$
- $\text{max}_q(a, b)$: encontrar o valor máximo no intervalo $[a, b]$

Por exemplo, considere o intervalo $[3, 6]$ no seguinte vetor:

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

Neste caso, $\text{soma}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ e $\text{max}_q(3, 6) = 6$.

Uma maneira simples de processar consultas de intervalo é usar um loop que percorre todos os valores do vetor no intervalo. Por exemplo, a seguinte função pode ser usada para processar consultas de soma em um vetor:

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

Esta função funciona em tempo $O(n)$, onde n é o tamanho do vetor. Assim, podemos processar q consultas em $O(nq)$ tempo usando a função. No entanto, se n e q forem grandes, essa abordagem é lenta. Felizmente, verifica-se que existem maneiras de processar consultas de intervalo com muito mais eficiência.

9.1 Consultas de vetor estático

Primeiro, vamos nos concentrar em uma situação em que o vetor é *estático*, ou seja, os valores do vetor nunca são atualizados entre as consultas. Nesse caso, é suficiente construir uma estrutura de dados estática que nos diga a resposta para qualquer consulta possível.

Consultas de soma

Podemos processar facilmente consultas de soma em um vetor estático construindo um **vetor de soma de prefixos**. Cada valor no vetor de soma de prefixos é igual à soma dos valores no vetor original até aquela posição, ou seja, o valor na posição k é $\text{soma}_q(0, k)$. O vetor de soma de prefixos pode ser construído em tempo $O(n)$.

Por exemplo, considere o seguinte vetor:

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

O vetor de soma de prefixos correspondente é o seguinte:

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

Como o vetor de soma de prefixos contém todos os valores de $\text{soma}_q(0, k)$, podemos calcular qualquer valor de $\text{soma}_q(a, b)$ em tempo $O(1)$ da seguinte forma:

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

Ao definir $\text{soma}_q(0, -1) = 0$, a fórmula acima também é válida quando $a = 0$.

Por exemplo, considere o intervalo $[3, 6]$:

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

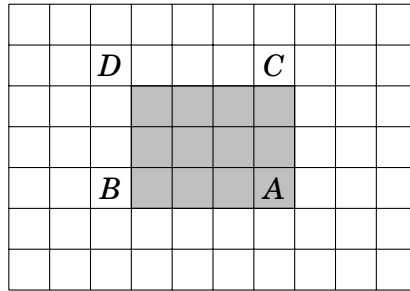
Nesse caso $\text{soma}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Essa soma pode ser calculada a partir de dois valores do vetor de soma de prefixos:

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

Assim, $\text{soma}_q(3, 6) = \text{soma}_q(0, 6) - \text{soma}_q(0, 2) = 27 - 8 = 19$.

Também é possível generalizar essa ideia para dimensões superiores. Por exemplo, podemos construir um vetor de soma de prefixos bidimensional que pode ser usado para calcular a soma de qualquer subvetor retangular em tempo $O(1)$. Cada soma em tal vetor corresponde a um subvetor que começa no canto superior esquerdo do vetor.

A imagem a seguir ilustra a ideia:



A soma do subvetor cinza pode ser calculada usando a fórmula

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

onde $S(X)$ denota a soma dos valores em um subvetor retangular do canto superior esquerdo até a posição de X .

Consultas de mínimo

Consultas de mínimo são mais difíceis de processar do que consultas de soma. Ainda assim, existe um método de pré-processamento de $O(n \log n)$ bastante simples, após o qual podemos responder a qualquer consulta de mínimo em tempo $O(1)$ ¹. Observe que, como as consultas de mínimo e máximo podem ser processadas de forma semelhante, podemos nos concentrar nas consultas de mínimo.

A ideia é pré-calcular todos os valores de $\min_q(a, b)$ onde $b - a + 1$ (o comprimento do intervalo) é uma potência de dois. Por exemplo, para o vetor

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

os seguintes valores são calculados:

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						

¹Esta técnica foi introduzida em [7] e às vezes chamada de método **sparse table**. Existem também técnicas mais sofisticadas [22] onde o tempo de pré-processamento é apenas $O(n)$, mas tais algoritmos não são necessários em programação competitiva.

O número de valores pré-calculados é $O(n \log n)$, porque existem comprimentos de intervalo $O(\log n)$ que são potências de dois. Os valores podem ser calculados de forma eficiente usando a fórmula recursiva

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

onde $b - a + 1$ é uma potência de dois e $w = (b - a + 1)/2$. Calcular todos esses valores leva tempo $O(n \log n)$.

Depois disso, qualquer valor de $\min_q(a, b)$ pode ser calculado em tempo $O(1)$ como um mínimo de dois valores pré-calculados. Seja k a maior potência de dois que não exceda $b - a + 1$. Podemos calcular o valor de $\min_q(a, b)$ usando a fórmula

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

Na fórmula acima, o intervalo $[a, b]$ é representado como a união dos intervalos $[a, a + k - 1]$ e $[b - k + 1, b]$, ambos de comprimento k .

Como exemplo, considere o intervalo $[1, 6]$:

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

O comprimento do intervalo é 6, e a maior potência de dois que não excede 6 é 4. Assim, o intervalo $[1, 6]$ é a união dos intervalos $[1, 4]$ e $[3, 6]$:

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

Como $\min_q(1, 4) = 3$ e $\min_q(3, 6) = 1$, concluímos que $\min_q(1, 6) = 1$.

9.2 Árvore binária indexada

Uma **árvore binária indexada** ou uma **árvore de Fenwick**² pode ser vista como uma variante dinâmica de um vetor de soma de prefixos. Ela suporta duas operações de tempo $O(\log n)$ em um vetor: processar uma consulta de soma de intervalo e atualizar um valor.

A vantagem de uma árvore binária indexada é que ela nos permite atualizar de forma eficiente os valores do vetor entre as consultas de soma. Isso não seria possível usando um vetor de soma de prefixos, porque após cada atualização, seria necessário construir todo o vetor de soma de prefixos novamente em tempo $O(n)$.

²A estrutura de árvore binária indexada foi apresentada por P. M. Fenwick em 1994 [21].

Estrutura

Mesmo que o nome da estrutura seja *árvore* binária indexada, ela é geralmente representada como um vetor. Nesta seção, assumimos que todos os vetores são indexados a partir de um, porque isso facilita a implementação.

Seja $p(k)$ a maior potência de dois que divide k . Armazenamos uma árvore binária indexada como um vetor $tree$ tal que

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

ou seja, cada posição k contém a soma dos valores em um intervalo do vetor original cujo comprimento é $p(k)$ e que termina na posição k . Por exemplo, como $p(6) = 2$, $tree[6]$ contém o valor de $\text{sum}_q(5, 6)$.

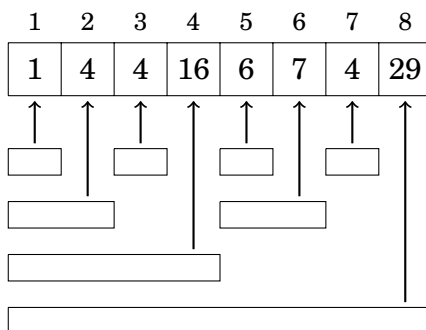
Por exemplo, considere o seguinte vetor:

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

A árvore binária indexada correspondente é a seguinte:

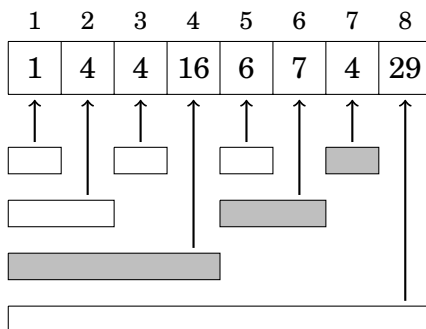
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

A figura a seguir mostra mais claramente como cada valor na árvore binária indexada corresponde a um intervalo no vetor original:



Usando uma árvore binária indexada, qualquer valor de $\text{sum}_q(1, k)$ pode ser calculado em tempo $O(\log n)$, porque um intervalo $[1, k]$ sempre pode ser dividido em $O(\log n)$ intervalos cujas somas são armazenadas na árvore.

Por exemplo, o intervalo $[1, 7]$ consiste nos seguintes intervalos:



Assim, podemos calcular a soma correspondente da seguinte forma:

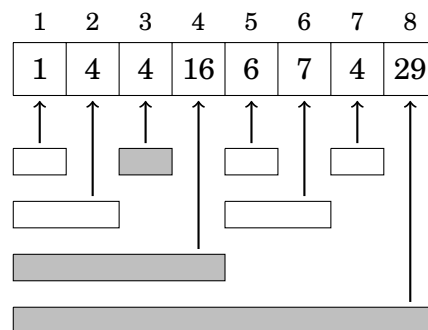
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

Para calcular o valor de $\text{sum}_q(a, b)$ onde $a > 1$, podemos usar o mesmo truque que usamos com vetores de soma de prefixos:

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

Como podemos calcular $\text{sum}_q(1, b)$ e $\text{sum}_q(1, a - 1)$ em tempo $O(\log n)$, a complexidade de tempo total é $O(\log n)$.

Então, após atualizar um valor no vetor original, vários valores na árvore binária indexada devem ser atualizados. Por exemplo, se o valor na posição 3 mudar, as somas dos seguintes intervalos mudam:



Como cada elemento do vetor pertence a $O(\log n)$ intervalos na árvore binária indexada, é suficiente atualizar $O(\log n)$ valores na árvore.

Implementação

As operações de uma árvore binária indexada podem ser implementadas eficientemente usando operações de bits. O fato chave necessário é que podemos calcular qualquer valor de $p(k)$ usando a fórmula

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

A seguinte função calcula o valor de $\text{sum}_q(1, k)$:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k & -k;
    }
    return s;
}
```

A seguinte função aumenta o valor do vetor na posição k em x (x pode ser positivo ou negativo):


```

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

```

A complexidade de tempo de ambas as funções é $O(\log n)$, porque as funções acessam $O(\log n)$ valores na árvore binária indexada, e cada movimento para a próxima posição leva tempo $O(1)$.

9.3 Árvore de segmentos

Uma **árvore de segmentos**³ é uma estrutura de dados que suporta duas operações: processar uma consulta de intervalo e atualizar um valor do vetor. As árvores de segmentos podem suportar consultas de soma, consultas de mínimo e máximo e muitas outras consultas para que ambas as operações funcionem em tempo $O(\log n)$.

Comparada a uma árvore binária indexada, a vantagem de uma árvore de segmentos é que ela é uma estrutura de dados mais geral. Enquanto as árvores binárias indexadas suportam apenas consultas de soma⁴, as árvores de segmentos também suportam outras consultas. Por outro lado, uma árvore de segmentos requer mais memória e é um pouco mais difícil de implementar.

Estrutura

Uma árvore de segmentos é uma árvore binária tal que os nós no nível inferior da árvore correspondem aos elementos do vetor, e os outros nós contêm informações necessárias para processar consultas de intervalo.

Nesta seção, assumimos que o tamanho do vetor é uma potência de dois e a indexação baseada em zero é usada, porque é conveniente construir uma árvore de segmentos para tal vetor. Se o tamanho do vetor não for uma potência de dois, podemos sempre anexar elementos extras a ele.

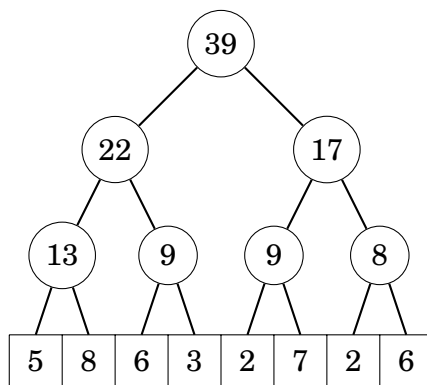
Vamos primeiro discutir árvores de segmentos que suportam consultas de soma. Como exemplo, considere o seguinte vetor:

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

A árvore de segmentos correspondente é a seguinte:

³A implementação bottom-up neste capítulo corresponde àquela em [62]. Estruturas semelhantes foram usadas no final dos anos 1970 para resolver problemas geométricos [9].

⁴Na verdade, usando *duas* árvores binárias indexadas, é possível suportar consultas de mínimo [16], mas isso é mais complicado do que usar uma árvore de segmentos.

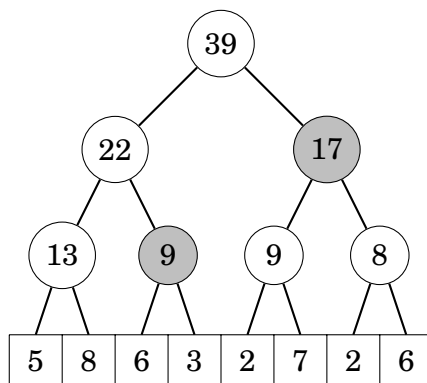


Cada nó interno da árvore corresponde a um intervalo do vetor cujo tamanho é uma potência de dois. Na árvore acima, o valor de cada nó interno é a soma dos valores correspondentes do vetor, e pode ser calculado como a soma dos valores de seu nó filho esquerdo e direito.

Acontece que qualquer intervalo $[a, b]$ pode ser dividido em $O(\log n)$ intervalos cujos valores são armazenados nos nós da árvore. Por exemplo, considere o intervalo $[2, 7]$:

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

Aqui $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. Neste caso, os dois nós da árvore a seguir correspondem ao intervalo:

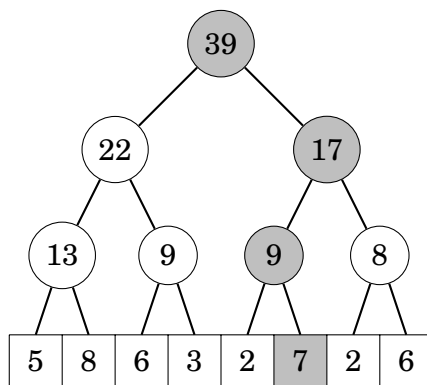


Assim, outra maneira de calcular a soma é $9 + 17 = 26$.

Quando a soma é calculada usando nós localizados o mais alto possível na árvore, no máximo dois nós em cada nível da árvore são necessários. Portanto, o número total de nós é $O(\log n)$.

Após uma atualização do vetor, devemos atualizar todos os nós cujo valor depende do valor atualizado. Isso pode ser feito percorrendo o caminho do elemento do vetor atualizado até o nó superior e atualizando os nós ao longo do caminho.

A figura a seguir mostra quais nós da árvore mudam se o valor do vetor 7 mudar:

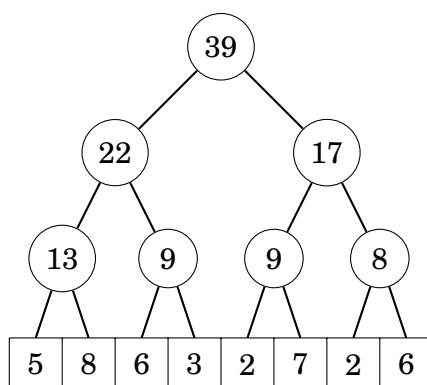


O caminho de baixo para cima sempre consiste em $O(\log n)$ nós, então cada atualização muda $O(\log n)$ nós na árvore.

Implementação

Armazenamos uma árvore de segmentos como um vetor de $2n$ elementos, onde n é o tamanho do vetor original e uma potência de dois. Os nós da árvore são armazenados de cima para baixo: $tree[1]$ é o nó superior, $tree[2]$ e $tree[3]$ são seus filhos, e assim por diante. Finalmente, os valores de $tree[n]$ a $tree[2n - 1]$ correspondem aos valores do vetor original no nível inferior da árvore.

Por exemplo, a árvore de segmentos



é armazenada da seguinte forma:

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

Usando essa representação, o pai de $tree[k]$ é $tree[\lfloor k/2 \rfloor]$, e seus filhos são $tree[2k]$ e $tree[2k + 1]$. Note que isso implica que a posição de um nó é par se for um filho esquerdo e ímpar se for um filho direito.

A seguinte função calcula o valor de $sum_q(a, b)$:

```
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;
}

```

A função mantém um intervalo que é inicialmente $[a + n, b + n]$. Então, a cada passo, o intervalo é movido um nível mais alto na árvore, e antes disso, os valores dos nós que não pertencem ao intervalo superior são adicionados à soma.

A seguinte função aumenta o valor do vetor na posição k por 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];
    }
}

```

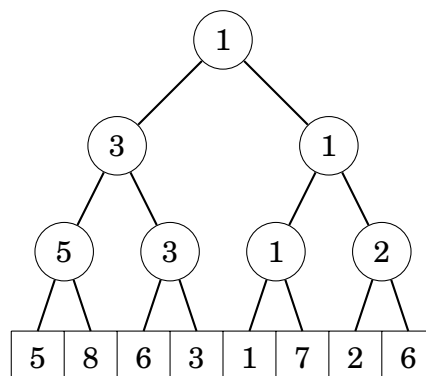
Primeiro a função atualiza o valor no nível inferior da árvore. Depois disso, a função atualiza os valores de todos os nós internos da árvore, até que alcance o nó superior da árvore.

Ambas as funções acima funcionam em tempo $O(\log n)$, pois uma árvore de segmentos de n elementos consiste em $O(\log n)$ níveis, e as funções movem um nível mais alto na árvore a cada passo.

Outras consultas

Árvores de segmentos podem suportar todas as consultas de intervalo onde é possível dividir um intervalo em duas partes, calcular a resposta separadamente para ambas as partes e então combinar as respostas de forma eficiente. Exemplos de tais consultas são mínimo e máximo, maior divisor comum, e operações de bits and, or e xor.

Por exemplo, a seguinte árvore de segmentos suporta consultas de mínimo:

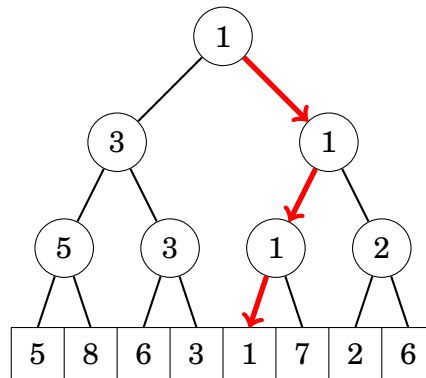


Neste caso, cada nó da árvore contém o menor valor no intervalo do vetor correspondente. O nó superior da árvore contém o menor valor em todo o vetor.

As operações podem ser implementadas como anteriormente, mas em vez de somas, os mínimos são calculados.

A estrutura de uma árvore de segmentos também nos permite usar a busca binária para localizar elementos do vetor. Por exemplo, se a árvore suporta consultas de mínimo, podemos encontrar a posição de um elemento com o menor valor em $O(\log n)$ tempo.

Por exemplo, na árvore acima, um elemento com o menor valor 1 pode ser encontrado traversando um caminho para baixo a partir do nó superior:



9.4 Técnicas adicionais

Compressão de índices

Uma limitação em estruturas de dados que são construídas sobre um vetor é que os elementos são indexados usando inteiros consecutivos. Dificuldades surgem quando índices grandes são necessários. Por exemplo, se desejarmos usar o índice 10^9 , o vetor deve conter 10^9 elementos, o que exigiria muita memória.

No entanto, geralmente podemos contornar essa limitação usando **compressão de índices**, onde os índices originais são substituídos por índices 1, 2, 3, etc. Isso pode ser feito se soubermos todos os índices necessários durante o algoritmo antecipadamente.

A ideia é substituir cada índice original x por $c(x)$, onde c é uma função que comprime os índices. Exigimos que a ordem dos índices não mude, então se $a < b$, então $c(a) < c(b)$. Isso nos permite realizar consultas convenientemente mesmo que os índices sejam comprimidos.

Por exemplo, se os índices originais são 555, 10^9 e 8, os novos índices são:

$$\begin{aligned} c(8) &= 1 \\ c(555) &= 2 \\ c(10^9) &= 3 \end{aligned}$$

Atualizações de intervalo

Até agora, implementamos estruturas de dados que suportam consultas de intervalo e atualizações de valores únicos. Vamos agora considerar uma situação

oposta, onde devemos atualizar intervalos e recuperar valores únicos. Vamos focar numa operação que aumenta todos os elementos num intervalo $[a, b]$ por x .

Surpreendentemente, podemos usar as estruturas de dados apresentadas neste capítulo também nesta situação. Para fazer isso, construímos um **vetor de diferenças** cujos valores indicam as diferenças entre valores consecutivos no vetor original. Assim, o vetor original é o vetor de soma de prefixo do vetor de diferenças. Por exemplo, considere o seguinte vetor:

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

O vetor de diferenças para o vetor acima é o seguinte:

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

Por exemplo, o valor 2 na posição 6 no vetor original corresponde à soma $3 - 2 + 4 - 3 = 2$ no vetor de diferenças.

A vantagem do vetor de diferenças é que podemos atualizar um intervalo no vetor original mudando apenas dois elementos no vetor de diferenças. Por exemplo, se quisermos aumentar o vetor original valores entre as posições 1 e 4 por 5, basta aumentar o valor do vetor de diferenças na posição 1 por 5 e diminuir o valor na posição 5 por 5. O resultado é o seguinte:

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

De forma mais geral, para aumentar os valores no intervalo $[a, b]$ por x , aumentamos o valor na posição a por x e diminuimos o valor na posição $b + 1$ por x . Assim, é só necessário atualizar valores únicos e processar consultas de soma, para que possamos usar uma árvore binária indexada ou uma árvore de segmentos.

Um problema mais difícil é suportar tanto consultas de intervalo quanto atualizações de intervalo. No Capítulo 28 veremos que mesmo isso é possível.

Manipulação de bits

10.1 Representação de bits

Aqui está a representação de bits do número int 43:

Os bits na representação são indexados da direita para a esquerda. Para converter uma representação de bits $b_k \cdots b_2 b_1 b_0$ em um número, podemos usar a fórmula

$$b_k 2^k + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

A representação de bits de um número é **assinada** ou **não assinada**. Geralmente uma representação assinada é usada, o que significa que números negativos e positivos podem ser representados. Uma variável assinada de n bits pode conter qualquer inteiro entre -2^{n-1} e $2^{n-1} - 1$. Por exemplo, o tipo `int` em C++ é um tipo assinado, então uma variável `int` pode conter qualquer inteiro entre -2^{31} e $2^{31} - 1$.

O primeiro bit em uma representação assinada é o sinal do número (0 para números não negativos e 1 para números negativos), e os $n - 1$ bits restantes contêm a magnitude do número. **Complemento de dois** é usado, o que significa que o número oposto de um número é calculado primeiro invertendo todos os bits no número, e depois aumentando o número por um.

Por exemplo, a representação de bits de o número int `-43` é

11111111111111111111111111010101.

Em uma representação não assinada, apenas os números não negativos podem ser usados, mas o limite superior para os valores é maior. Uma variável não assinada de n bits pode conter qualquer inteiro entre 0 e $2^n - 1$. Por exemplo, em C++, uma variável `unsigned int` pode conter qualquer inteiro entre 0 e $2^{32} - 1$.

Existe uma conexão entre as representações: um número assinado $-x$ é igual a um número não assinado $2^n - x$. Por exemplo, o código a seguir mostra que o número assinado $x = -43$ é igual ao número não assinado $y = 2^{32} - 43$:

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

Se um número for maior que o limite superior da representação de bits, o número sofrerá overflow. Em uma representação assinada, o próximo número após $2^{n-1} - 1$ é -2^{n-1} , e em uma representação não assinada, o próximo número após $2^n - 1$ é 0. Por exemplo, considere o código a seguir:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Inicialmente, o valor de x é $2^{31} - 1$. Este é o maior valor que pode ser armazenado em uma variável `int`, então o próximo número após $2^{31} - 1$ é -2^{31} .

10.2 Operações de bits

Operação E

A operação **E** $x \& y$ produz um número que tem bits iguais a um nas posições onde ambos x e y têm bits iguais a um. Por exemplo, $22 \& 26 = 18$, porque

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

Usando a operação E, podemos verificar se um número x é par porque $x \& 1 = 0$ se x é par, e $x \& 1 = 1$ se x é ímpar. De forma mais geral, x é divisível por 2^k exatamente quando $x \& (2^k - 1) = 0$.

Operação OU

A operação **OU** $x \mid y$ produz um número que tem bits iguais a um nas posições onde pelo menos um de x e y tem bits iguais a um. Por exemplo, $22 \mid 26 = 30$, porque

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

Operação XOR

A operação **XOR** $x \wedge y$ produz um número que tem bits iguais a um nas posições onde exatamente um de x e y tem bits iguais a um. Por exemplo, $22 \wedge 26 = 12$, porque

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

Operação NÃO

A operação **NÃO** $\sim x$ produz um número onde todos os bits de x foram invertidos. A fórmula $\sim x = -x - 1$ é válida, por exemplo, $\sim 29 = -30$.

O resultado da operação NÃO no nível de bits depende do comprimento da representação de bits, porque a operação inverte todos os bits. Por exemplo, se os números são de 32 bits números int, o resultado é o seguinte:

$$\begin{array}{rcl}
 x & = & 29 \quad 000000000000000000000000011101 \\
 \sim x & = & -30 \quad 1111111111111111111111111100010
 \end{array}$$

Deslocamentos de bits

O deslocamento de bits para a esquerda $x \ll k$ anexa k bits zero ao número, e o deslocamento de bits para a direita $x \gg k$ remove os k últimos bits do número. Por exemplo, $14 \ll 2 = 56$, porque 14 e 56 correspondem a 1110 e 111000. Da mesma forma, $49 \gg 3 = 6$, porque 49 e 6 correspondem a 110001 e 110.

Observe que $x \ll k$ corresponde a multiplicar x por 2^k , e $x \gg k$ corresponde a dividir x por 2^k arredondado para baixo para um inteiro.

Aplicações

Um número da forma $1 \ll k$ tem um bit na posição k e todos os outros bits são zero, então podemos usar esses números para acessar bits únicos de números. Em particular, o k -ésimo bit de um número é igual a um exatamente quando $x \& (1 \ll k)$ não é zero. O código a seguir imprime a representação de bits de um número int x :

```

for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}

```

Também é possível modificar bits únicos de números usando ideias semelhantes. Por exemplo, a fórmula $x \mid (1 \ll k)$ define o k -ésimo bit de x como igual a um, a fórmula $x \& \sim(1 \ll k)$ define o k -ésimo bit de x como igual a zero, e a fórmula $x \wedge (1 \ll k)$ inverte o k -ésimo bit de x .

A fórmula $x \& (x - 1)$ define o último bit igual a um de x como igual a zero, e a fórmula $x \& -x$ define todos os bits iguais a um como iguais a zero, exceto pelo último bit igual a um. A fórmula $x \mid (x - 1)$ inverte todos os bits após o último bit igual a um. Observe também que um número positivo x é uma potência de dois exatamente quando $x \& (x - 1) = 0$.

Funções adicionais

O compilador g++ fornece o seguinte funções para contar bits:

- `__builtin_clz(x)`: o número de zeros no início do número
- `__builtin_ctz(x)`: o número de zeros no final do número
- `__builtin_popcount(x)`: o número de uns no número
- `__builtin_parity(x)`: a paridade (par ou ímpar) do número de uns

As funções podem ser usadas da seguinte forma:

```
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
```

Apesar das funções acima suportarem apenas o tipo `int`, também existem versões para `long` `long`, disponíveis com o sufixo `ll`.

10.3 Representando conjuntos

Cada subconjunto de um conjunto $\{0, 1, 2, \dots, n - 1\}$ pode ser representado como um inteiro de n bits cujos bits iguais a um indicam quais elementos pertencem ao subconjunto. Esta é uma maneira eficiente de representar conjuntos, porque cada elemento requer apenas um bit de memória, e as operações de conjunto podem ser implementadas como operações de bits.

Por exemplo, como `int` é um tipo de 32 bits, um número `int` pode representar qualquer subconjunto do conjunto $\{0, 1, 2, \dots, 31\}$. A representação de bits do conjunto $\{1, 3, 4, 8\}$ é

000000000000000000000000100011010,

que corresponde ao número $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Implementação de conjunto

O código a seguir declara uma variável `int` chamada `x` que pode conter um subconjunto de $\{0, 1, 2, \dots, 31\}$. Depois disso, o código adiciona os elementos 1, 3, 4 e 8 ao conjunto e imprime o tamanho do conjunto.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Então, o código a seguir imprime todos elementos que pertencem ao conjunto:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// saída: 1 3 4 8
```

Operações de conjunto

As operações de conjunto podem ser implementadas da seguinte forma como operações de bits:

	sintaxe de conjunto	sintaxe de bits
interseção	$a \cap b$	$a \& b$
união	$a \cup b$	$a \mid b$
complemento	\bar{a}	$\sim a$
diferença	$a \setminus b$	$a \& (\sim b)$

Por exemplo, o código a seguir primeiro constrói os conjuntos $x = \{1, 3, 4, 8\}$ e $y = \{3, 6, 8, 9\}$, e então constrói o conjunto $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```
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
```

Iterando por subconjuntos

O código a seguir percorre os subconjuntos de $\{0, 1, \dots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // processar subconjunto b
}
```

O código a seguir percorre os subconjuntos com exatamente k elementos:

```

for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // processar subconjunto b
    }
}

```

O código a seguir percorre os subconjuntos de um conjunto x :

```

int b = 0;
do {
    // processar subconjunto b
} while (b=(b-x)&x);

```

10.4 Otimizações de bits

Muitos algoritmos podem ser otimizados usando operações de bits. Essas otimizações não alteram a complexidade de tempo do algoritmo, mas podem ter um grande impacto no tempo de execução real do código. Nesta seção, discutimos exemplos de tais situações.

Distâncias de Hamming

A **distância de Hamming** $\text{hamming}(a, b)$ entre duas cadeias de caracteres a e b de comprimento igual é o número de posições onde as cadeias de caracteres diferem. Por exemplo,

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

Considere o seguinte problema: Dado uma lista de n cadeias de caracteres de bits, cada uma de comprimento k , calcule a distância de Hamming mínima entre duas cadeias de caracteres na lista. Por exemplo, a resposta para $[00111, 01101, 11110]$ é 2, porque

- $\text{hamming}(00111, 01101) = 2$,
- $\text{hamming}(00111, 11110) = 3$, e
- $\text{hamming}(01101, 11110) = 3$.

Uma maneira direta de resolver o problema é percorrer todos os pares de cadeias de caracteres e calcular suas distâncias de Hamming, o que gera um algoritmo de tempo $O(n^2k)$. A função a seguir pode ser usada para calcular distâncias:

```

int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}

```

```
}
```

No entanto, se k for pequeno, podemos otimizar o código armazenando as cadeias de caracteres de bits como inteiros e calculando as distâncias de Hamming usando operações de bits. Em particular, se $k \leq 32$, podemos simplesmente armazenar as cadeias de caracteres como valores `int` e usar a seguinte função para calcular distâncias:

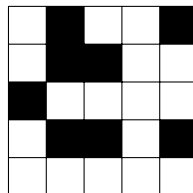
```
int hamming(int a, int b) {  
    return __builtin_popcount(a^b);  
}
```

Na função acima, a operação XOR constrói uma cadeia de caracteres de bits que tem bits iguais a um nas posições onde a e b diferem. Então, o número de bits é calculado usando a função `__builtin_popcount`.

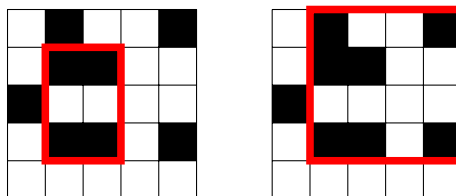
Para comparar as implementações, geramos uma lista de 10000 cadeias de caracteres de bits aleatórias de comprimento 30. Usando a primeira abordagem, a pesquisa levou 13,5 segundos, e depois da otimização de bits, levou apenas 0,5 segundos. Portanto, o código otimizado de bits era quase 30 vezes mais rápido que o código original.

Contando subgrades

Como outro exemplo, considere o seguinte problema: Dada uma grade $n \times n$ em que cada quadrado é preto (1) ou branco (0), calcule o número de subgrades em que todos os cantos são pretos. Por exemplo, a grade



contém duas dessas subgrades:



Existe um algoritmo de tempo $O(n^3)$ para resolver o problema: percorrer todos os $O(n^2)$ pares de linhas e para cada par (a, b) calcule o número de colunas que contêm um preto quadrado em ambas as linhas em $O(n)$ tempo. O código a seguir assume que `color[y][x]` denota a cor na linha y e coluna x :

```
int count = 0;  
for (int i = 0; i < n; i++) {  
    if (color[a][i] == 1 && color[b][i] == 1) count++;  
}
```

```
}
```

Então, essas colunas contam para $\text{count}(\text{count} - 1)/2$ subgrades com cantos pretos, porque podemos escolher quaisquer dois deles para formar uma subgrade.

Para otimizar esse algoritmo, dividimos a grade em blocos de colunas de forma que cada bloco consista em N colunas consecutivas. Então, cada linha é armazenada como uma lista de números de N bits que descrevem as cores dos quadrados. Agora podemos processar N colunas ao mesmo tempo usando operações de bits. No código a seguir, $\text{color}[y][k]$ representa um bloco de N cores como bits.

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

O algoritmo resultante funciona em $O(n^3/N)$ tempo.

Geramos uma grade aleatória de tamanho 2500×2500 e comparamos a implementação original e otimizada de bits. Enquanto o código original levou 29,6 segundos, a versão otimizada de bits levou apenas 3,1 segundos com $N = 32$ (números int) e 1,7 segundos com $N = 64$ (números long long).

10.5 Programação dinâmica

As operações de bits fornecem uma maneira eficiente e conveniente de implementar algoritmos de programação dinâmica cujos estados contêm subconjuntos de elementos, porque esses estados podem ser armazenados como inteiros. A seguir, discutimos exemplos de combinação de operações de bits e programação dinâmica.

Seleção ótima

Como primeiro exemplo, considere o seguinte problema: Somos dados os preços de k produtos ao longo de n dias, e queremos comprar cada produto exatamente uma vez. No entanto, podemos comprar no máximo um produto em um dia. Qual é o preço total mínimo? Por exemplo, considere o seguinte cenário ($k = 3$ e $n = 8$):

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

Neste cenário, o preço total mínimo é 5:

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

Seja $\text{price}[x][d]$ o preço do produto x no dia d . Por exemplo, no cenário acima $\text{price}[2][3] = 7$. Então, seja $\text{total}(S, d)$ o preço total mínimo para comprar um subconjunto S de produtos até o dia d . Usando essa função, a solução para o problema é $\text{total}(\{0 \dots k-1\}, n-1)$.

Primeiro, $\text{total}(\emptyset, d) = 0$, porque não custa nada comprar um conjunto vazio, e $\text{total}(\{x\}, 0) = \text{price}[x][0]$, porque existe uma maneira de comprar um produto no primeiro dia. Então, a seguinte recorrência pode ser usada:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

Isso significa que nós ou não compramos nenhum produto no dia d ou compramos um produto x que pertence a S . Neste último caso, removemos x de S e adicionamos o preço de x ao preço total.

A próxima etapa é calcular os valores da função usando programação dinâmica. Para armazenar os valores da função, declaramos um array

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

onde K e N são constantes suficientemente grandes. A primeira dimensão do array corresponde a uma representação de bits de um subconjunto.

Primeiro, os casos onde $d = 0$ podem ser processados da seguinte forma:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Então, a recorrência se traduz no seguinte código:

```
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]);
            }
        }
    }
}
```

A complexidade de tempo do algoritmo é $O(n2^k k)$.

De permutações para subconjuntos

Usando programação dinâmica, é frequentemente possível transformar uma iteração sobre permutações em uma iteração sobre subconjuntos¹. A vantagem disso é que $n!$, o número de permutações, é muito maior que 2^n , o número de subconjuntos. Por exemplo, se $n = 20$, então $n! \approx 2.4 \cdot 10^{18}$ e $2^n \approx 10^6$. Assim, para certos valores de n , podemos iterar eficientemente pelos subconjuntos, mas não pelas permutações.

Como exemplo, considere o seguinte problema: Há um elevador com peso máximo x , e n pessoas com pesos conhecidos que desejam ir do térreo para o último andar. Qual é o número mínimo de viagens necessárias se as pessoas entrarem no elevador em uma ordem ótima?

Por exemplo, suponha que $x = 10$, $n = 5$ e os pesos são os seguintes:

pessoa	peso
0	2
1	3
2	3
3	5
4	6

Nesse caso, o número mínimo de viagens é 2. Uma ordem ótima é $\{0, 2, 3, 1, 4\}$, que divide as pessoas em duas viagens: primeiro $\{0, 2, 3\}$ (peso total 10), e então $\{1, 4\}$ (peso total 9).

O problema pode ser facilmente resolvido em $O(n!n)$ tempo testando todas as possíveis permutações de n pessoas. No entanto, podemos usar programação dinâmica para obter um algoritmo mais eficiente em tempo $O(2^n n)$. A ideia é calcular para cada subconjunto de pessoas dois valores: o número mínimo de viagens necessárias e o peso mínimo das pessoas que viajam no último grupo.

Seja $\text{peso}[p]$ o peso da pessoa p . Definimos duas funções: $\text{viagens}(S)$ é o número mínimo de viagens para um subconjunto S , e $\text{ultima}(S)$ é o peso mínimo da última viagem. Por exemplo, no cenário acima

$$\text{viagens}(\{1, 3, 4\}) = 2 \quad \text{e} \quad \text{ultima}(\{1, 3, 4\}) = 5,$$

porque as viagens ótimas são $\{1, 4\}$ e $\{3\}$, e a segunda viagem tem peso 5. Claro, nosso objetivo final é calcular o valor de $\text{viagens}(\{0 \dots n-1\})$.

Podemos calcular os valores das funções recursivamente e então aplicar programação dinâmica. A ideia é percorrer todas as pessoas que pertencem a S e escolher de forma ótima a última pessoa p que entra no elevador. Cada escolha desse tipo gera um subproblema para um subconjunto menor de pessoas. Se $\text{ultima}(S \setminus p) + \text{peso}[p] \leq x$, podemos adicionar p à última viagem. Caso contrário, precisamos reservar uma nova viagem que inicialmente contém apenas p .

Para implementar a programação dinâmica, declaramos um array

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

¹Essa técnica foi introduzida em 1962 por M. Held e R. M. Karp [34].

que contém para cada subconjunto S um par $(viagens(S), ultima(S))$. Definimos o valor para um grupo vazio da seguinte forma:

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

Então, podemos preencher o array da seguinte forma:

```
for (int s = 1; s < (1<<n); s++) {
    // valor inicial: n+1 viagens sao necessarias
    melhor[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto opcao = melhor[s^(1<<p)];
            if (opcao.second+peso[p] <= x) {
                // adicionar p a uma viagem existente
                opcao.second += peso[p];
            } else {
                // reservar uma nova viagem para p
                opcao.first++;
                opcao.second = peso[p];
            }
            melhor[s] = min(melhor[s], opcao);
        }
    }
}
```

Observe que o loop acima garante que para quaisquer dois subconjuntos S_1 e S_2 tais que $S_1 \subset S_2$, processamos S_1 antes de S_2 . Assim, os valores de programação dinâmica são calculados na ordem correta.

Contando subconjuntos

Nosso último problema neste capítulo é o seguinte: Seja $X = \{0 \dots n-1\}$, e cada subconjunto $S \subset X$ recebe um inteiro $valor[S]$. Nossa tarefa é calcular para cada S

$$soma(S) = \sum_{A \subset S} valor[A],$$

ou seja, a soma dos valores dos subconjuntos de S .

Por exemplo, suponha que $n = 3$ e os valores são os seguintes:

- $valor[\emptyset] = 3$
- $valor[\{2\}] = 5$
- $valor[\{0\}] = 1$
- $valor[\{0,2\}] = 1$
- $valor[\{1\}] = 4$
- $valor[\{1,2\}] = 3$
- $valor[\{0,1\}] = 5$
- $valor[\{0,1,2\}] = 3$

Nesse caso, por exemplo,”

$$\begin{aligned}\text{soma}(\{0,2\}) &= \text{valor}[\emptyset] + \text{valor}[\{0\}] + \text{valor}[\{2\}] + \text{valor}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

Como há um total de 2^n subconjuntos, uma possível solução é percorrer todos os pares de subconjuntos em tempo $O(2^{2n})$. No entanto, usando programação dinâmica, podemos resolver o problema em tempo $O(2^n n)$. A ideia é focar em somas onde os elementos que podem ser removidos de S são restritos.

Seja $\text{parcial}(S, k)$ a soma dos valores dos subconjuntos de S com a restrição de que apenas os elementos $0 \dots k$ podem ser removidos de S . Por exemplo,

$$\text{parcial}(\{0,2\}, 1) = \text{valor}[\{2\}] + \text{valor}[\{0,2\}],$$

porque podemos remover apenas os elementos $0 \dots 1$. Podemos calcular os valores de soma usando os valores de parcial , porque

$$\text{soma}(S) = \text{parcial}(S, n-1).$$

Os casos base para a função são

$$\text{parcial}(S, -1) = \text{valor}[S],$$

porque nesse caso nenhum elemento pode ser removido de S . Então, no caso geral, podemos usar a seguinte recorrência:

$$\text{parcial}(S, k) = \begin{cases} \text{parcial}(S, k-1) & k \notin S \\ \text{parcial}(S, k-1) + \text{parcial}(S \setminus \{k\}, k-1) & k \in S \end{cases}$$

Aqui focamos no elemento k . Se $k \in S$, temos duas opções: podemos manter k em S ou removê-lo de S .

Há uma maneira particularmente inteligente de implementar o cálculo das somas. Podemos declarar um array

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

que conterá a soma de cada subconjunto. O array é inicializado da seguinte forma:

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

Então, podemos preencher o array da seguinte forma:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s & (1<<k)) soma[s] += soma[s ^ (1<<k)];
    }
}
```

Esse código calcula os valores de $\text{parcial}(S, k)$ para $k = 0 \dots n-1$ no array soma . Como $\text{parcial}(S, k)$ é sempre baseado em $\text{parcial}(S, k-1)$, podemos reutilizar o array soma , o que gera uma implementação muito eficiente.

Parte II

Algoritmos de grafos

Capítulo 11

Conceitos básicos sobre grafos

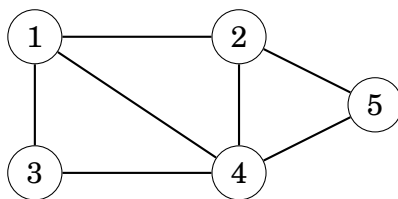
Muitos problemas de programação podem ser resolvidos modelando o problema como um problema de grafo e usando um algoritmo de grafo apropriado. Um exemplo típico de grafo é uma rede de estradas e cidades em um país. Às vezes, porém, o grafo está oculto no problema e pode ser difícil detectá-lo.

Esta parte do livro discute algoritmos de grafos, com foco especial em tópicos que são importantes na programação competitiva. Neste capítulo, abordaremos conceitos relacionados a grafos e estudaremos diferentes maneiras de representar grafos em algoritmos.

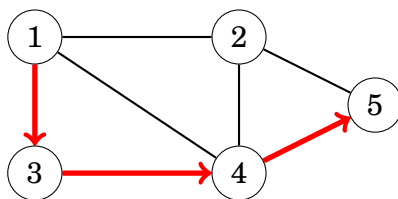
11.1 Terminologia de grafos

Um **grafo** consiste em **nós** e **arestas**. Neste livro, a variável n denota o número de nós em um grafo, e a variável m denota o número de arestas. Os nós são numerados usando inteiros $1, 2, \dots, n$.

Por exemplo, o grafo a seguir consiste em 5 nós e 7 arestas:



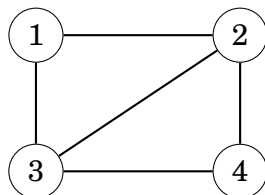
Um **caminho** leva do nó a ao nó b através das arestas do grafo. O **comprimento** de um caminho é o número de arestas nele. Por exemplo, o grafo acima contém um caminho $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ de comprimento 3 do nó 1 ao nó 5:



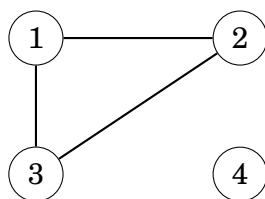
Um caminho é um **ciclo** se o primeiro e o último nó forem os mesmos. Por exemplo, o grafo acima contém um ciclo $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. Um caminho é **simples** se cada nó aparecer no máximo uma vez no caminho.

Conectividade

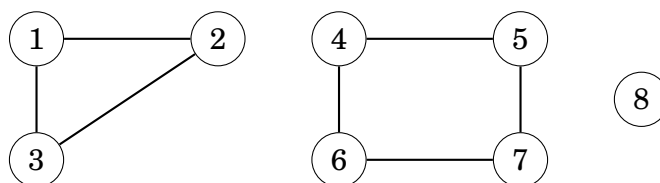
Um grafo é **conexo** se houver um caminho entre quaisquer dois nós. Por exemplo, o seguinte grafo é conexo:



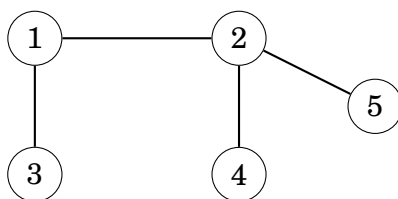
O seguinte grafo não é conexo, porque não é possível ir do nó 4 a nenhum outro nó:



As partes conectadas de um grafo são chamadas de seus **componentes**. Por exemplo, o seguinte grafo contém três componentes: {1, 2, 3}, {4, 5, 6, 7} e {8}.

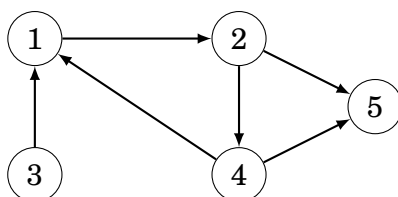


Uma **árvore** é um grafo conexo que consiste em n nós e $n - 1$ arestas. Existe um único caminho entre quaisquer dois nós de uma árvore. Por exemplo, o seguinte grafo é uma árvore:



Direções das arestas

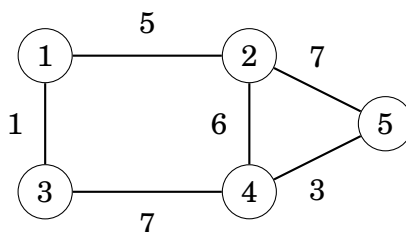
Um grafo é **direcionado** se as arestas puderem ser percorridas em apenas uma direção. Por exemplo, o seguinte grafo é direcionado:



O grafo acima contém um caminho $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ do nó 3 ao nó 5, mas não há caminho do nó 5 ao nó 3.

Pesos das arestas

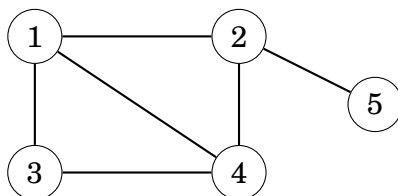
Em um grafo **ponderado**, cada aresta recebe um **peso**. Os pesos são frequentemente interpretados como comprimentos de aresta. Por exemplo, o seguinte grafo é ponderado:



O comprimento de um caminho em um grafo ponderado é a soma dos pesos das arestas no caminho. Por exemplo, no grafo acima, o comprimento do caminho $1 \rightarrow 2 \rightarrow 5$ é 12, e o comprimento do caminho $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ é 11. O último caminho é o **caminho mais curto** do nó 1 ao nó 5.

Vizinhos e graus

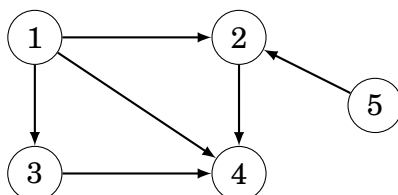
Dois nós são **vizinhos** ou **adjacentes** se houver uma aresta entre eles. O **grau** de um nó é o número de seus vizinhos. Por exemplo, no seguinte grafo, os vizinhos do nó 2 são 1, 4 e 5, então seu grau é 3.



A soma dos graus em um grafo é sempre $2m$, onde m é o número de arestas, porque cada aresta aumenta em um o grau de exatamente dois nós. Por esta razão, a soma dos graus é sempre par.

Um grafo é **regular** se o grau de cada nó for uma constante d . Um grafo é **completo** se o grau de cada nó for $n - 1$, ou seja, o grafo contém todas as arestas possíveis entre os nós.

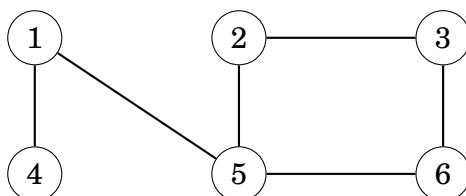
Em um grafo direcionado, o **grau de entrada** de um nó é o número de arestas que terminam no nó, e o **grau de saída** de um nó é o número de arestas que começam no nó. Por exemplo, no seguinte grafo, o grau de entrada do nó 2 é 2, e o grau de saída do nó 2 é 1.



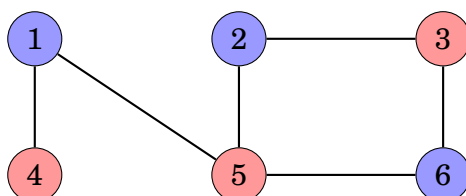
Colorações

Em uma **coloração** de um grafo, cada nó recebe uma cor de forma que nenhum nó adjacente tenha a mesma cor.

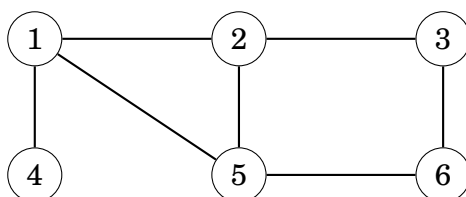
Um grafo é **bipartido** se for possível colori-lo usando duas cores. Acontece que um grafo é bipartido exatamente quando não contém um ciclo com um número ímpar de arestas. Por exemplo, o grafo



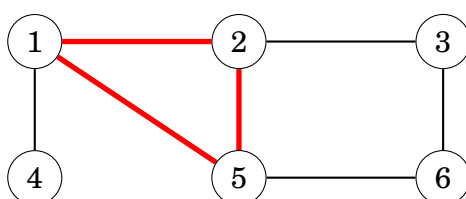
é bipartido, pois pode ser colorido da seguinte forma:



No entanto, o grafo

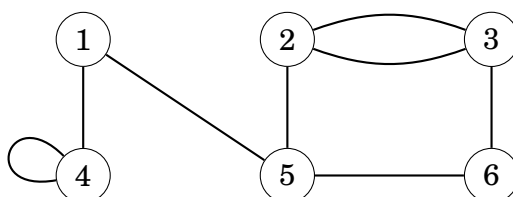


não é bipartido, porque não é possível colorir o seguinte ciclo de três nós usando duas cores:



Simplicidade

Um grafo é **simples** se nenhuma aresta começa e termina no mesmo nó, e não há múltiplas arestas entre dois nós. Muitas vezes, assumimos que os grafos são simples. Por exemplo, o seguinte grafo *não* é simples:



11.2 Representação de grafos

Existem várias maneiras de representar grafos em algoritmos. A escolha de uma estrutura de dados depende do tamanho do grafo e da maneira como o algoritmo o processa. A seguir, veremos três representações comuns.

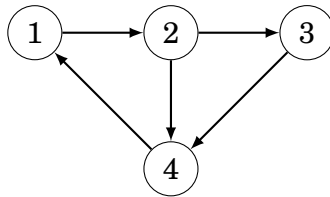
Representação de lista de adjacência

Na representação de lista de adjacência, cada nó x no grafo recebe uma **lista de adjacência** que consiste em nós para os quais há uma aresta de x . Listas de adjacência são a maneira mais popular de representar grafos, e a maioria dos algoritmos pode ser implementada de forma eficiente usando-as.

Uma maneira conveniente de armazenar as listas de adjacência é declarar um array de vetores da seguinte forma:

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

A constante N é escolhida de forma que todas as listas de adjacência possam ser armazenadas. Por exemplo, o grafo



pode ser armazenado da seguinte forma:

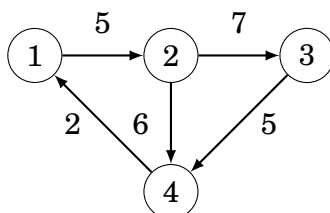
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Se o grafo for não direcionado, ele pode ser armazenado de forma semelhante, mas cada aresta é adicionada em ambas as direções.

Para um grafo ponderado, a estrutura pode ser estendida da seguinte forma:

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

Nesse caso, a lista de adjacência do nó a contém o par (b, w) sempre que houver uma aresta do nó a ao nó b com peso w . Por exemplo, o grafo



pode ser armazenado da seguinte forma:

```
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});
```

O benefício de usar listas de adjacência é que podemos encontrar os nós para os quais podemos mover de um determinado nó através de uma aresta de forma eficiente. Por exemplo, o seguinte loop percorre todos os nós para os quais podemos mover do nó s :

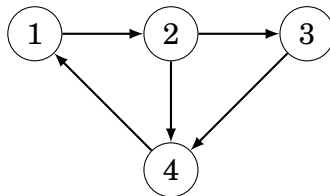
```
for (auto u : adj[s]) {  
    // processa o nó u  
}
```

Representação de matriz de adjacência

Uma **matriz de adjacência** é um array bidimensional que indica quais arestas o grafo contém. Podemos verificar eficientemente a partir de uma matriz de adjacência se existe uma aresta entre dois nós. A matriz pode ser armazenada como um array

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

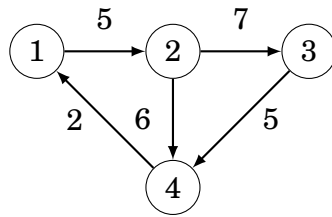
onde cada valor $adj[a][b]$ indica se o grafo contém uma aresta do nó a ao nó b . Se a aresta estiver incluída no grafo, então $adj[a][b] = 1$, e caso contrário $adj[a][b] = 0$. Por exemplo, o grafo



pode ser representado da seguinte forma:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Se o grafo for ponderado, a representação da matriz de adjacência pode ser estendida para que a matriz contenha o peso da aresta, se a aresta existir. Usando essa representação, o grafo



corresponde à seguinte matriz:

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

A desvantagem da representação da matriz de adjacência é que a matriz contém n^2 elementos e, geralmente, a maioria deles é zero. Por esse motivo, a representação não pode ser usada se o grafo for grande.

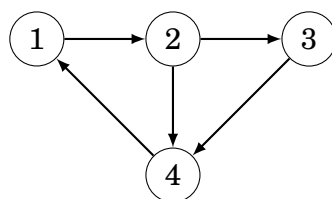
Representação de lista de arestas

Uma **lista de arestas** contém todas as arestas de um grafo em alguma ordem. Esta é uma maneira conveniente de representar um grafo se o algoritmo processa todas as arestas do grafo e não é necessário encontrar arestas que comecem em um determinado nó.

A lista de arestas pode ser armazenada em um vetor

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

onde cada par (a, b) denota que há uma aresta do nó a ao nó b . Assim, o grafo



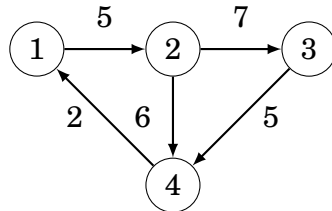
pode ser representado da seguinte forma:

```
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});
```

Se o grafo for ponderado, a estrutura pode ser estendida da seguinte forma:

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

Cada elemento desta lista é da forma (a,b,w) , o que significa que há uma aresta do nó a ao nó b com peso w . Por exemplo, o grafo



pode ser representado da seguinte forma¹:

```
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});
```

¹Em alguns compiladores mais antigos, a função `make_tuple` deve ser usada no lugar das chaves (por exemplo, `make_tuple(1,2,5)` em vez de `{1,2,5}`).

Capítulo 12

Travessia de Grafos

Este capítulo discute dois algoritmos fundamentais de grafos: busca em profundidade e busca em largura. Ambos os algoritmos recebem um nó inicial no grafo e visitam todos os nós que podem ser alcançados a partir do nó inicial. A diferença entre os algoritmos está na ordem em que eles visitam os nós.

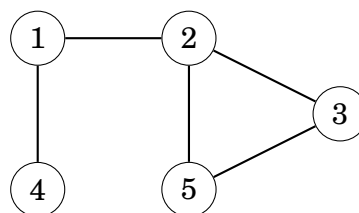
12.1 Busca em Profundidade

Busca em profundidade (DFS) é uma técnica simples de travessia de grafos. O algoritmo começa em um nó inicial e prossegue para todos os outros nós que são alcançáveis a partir do nó inicial usando as arestas do grafo.

A busca em profundidade sempre segue um único caminho no grafo enquanto encontra novos nós. Depois disso, ele retorna aos nós anteriores e começa a explorar outras partes do grafo. O algoritmo mantém o controle dos nós visitados, para que processe cada nó apenas uma vez.

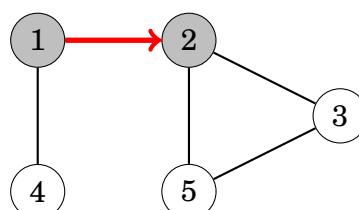
Exemplo

Vamos considerar como a busca em profundidade processa o seguinte grafo:

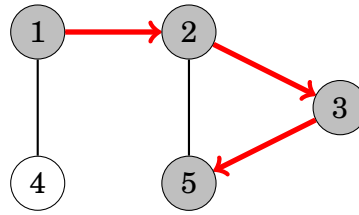


Podemos começar a busca em qualquer nó do grafo; agora vamos começar a busca no nó 1.

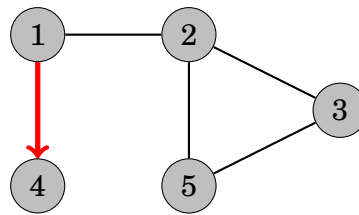
A busca primeiro prossegue para o nó 2:



Depois disso, os nós 3 e 5 serão visitados:



Os vizinhos do nó 5 são 2 e 3, mas a busca já visitou ambos, então é hora de retornar aos nós anteriores. Os vizinhos dos nós 3 e 2 também já foram visitados, então passamos do nó 1 para o nó 4:



Depois disso, a busca termina porque visitou todos os nós.

A complexidade de tempo da busca em profundidade é $O(n + m)$ onde n é o número de nós e m é o número de arestas, porque o algoritmo processa cada nó e aresta uma vez.

Implementação

A busca em profundidade pode ser convenientemente implementada usando recursão. A seguinte função `dfs` inicia uma busca em profundidade em um determinado nó. A função assume que o grafo é armazenado como listas de adjacência em um array:

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

e também mantém um array:

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

que mantém o controle dos nós visitados. Inicialmente, cada valor do array é `false`, e quando a busca chega ao nó s , o valor de `visited[s]` se torna `true`. A função pode ser implementada da seguinte forma:

```
void dfs(int s) {  
    if (visited[s]) return;  
    visited[s] = true;  
    // processa o nó s  
    for (auto u: adj[s]) {  
        dfs(u);  
    }  
}
```

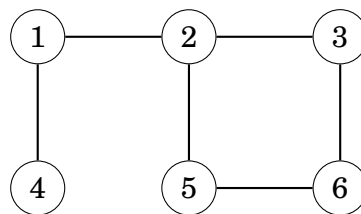
12.2 Busca em Largura

Busca em largura (BFS) visita os nós em ordem crescente de sua distância do nó inicial. Assim, podemos calcular a distância do nó inicial a todos os outros nós usando a busca em largura. No entanto, a busca em largura é mais difícil de implementar do que a busca em profundidade.

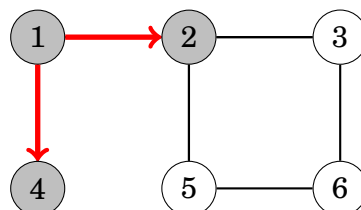
A busca em largura percorre os nós um nível após o outro. Primeiro, a busca explora os nós cuja distância do nó inicial é 1, depois os nós cuja distância é 2 e assim por diante. Este processo continua até que todos os nós tenham sido visitados.

Exemplo

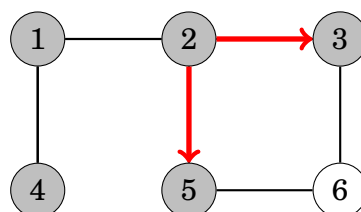
Vamos considerar como a busca em largura processa o seguinte grafo:



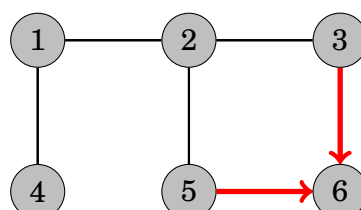
Suponha que a busca comece no nó 1. Primeiro, processamos todos os nós que podem ser alcançados a partir do nó 1 usando uma única aresta:



Depois disso, prosseguimos para os nós 3 e 5:



Finalmente, visitamos o nó 6:



Agora, calculamos as distâncias do nó inicial a todos os nós do grafo. As distâncias são as seguintes:

nó	distância
1	0
2	1
3	2
4	1
5	2
6	3

Como na busca em profundidade, a complexidade de tempo da busca em largura é $O(n + m)$, onde n é o número de nós e m é o número de arestas.

Implementação

A busca em largura é mais difícil de implementar do que a busca em profundidade, porque o algoritmo visita nós em diferentes partes do grafo. Uma implementação típica é baseada em uma fila que contém nós. A cada etapa, o próximo nó na fila será processado.

O código a seguir assume que o grafo é armazenado como listas de adjacência e mantém as seguintes estruturas de dados:

```
queue<int> q;  
bool visited[N];  
int distance[N];
```

A fila q contém nós a serem processados em ordem crescente de sua distância. Novos nós são sempre adicionados ao final da fila, e o nó no início da fila é o próximo nó a ser processado. O array `visited` indica quais nós a busca já visitou e o array `distance` conterá as distâncias do nó inicial a todos os nós do grafo.

A busca pode ser implementada da seguinte forma, começando no nó x :

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

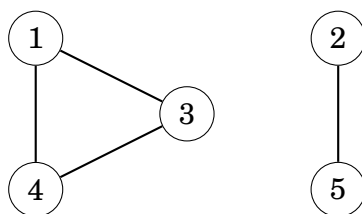

12.3 Aplicações

Usando os algoritmos de travessia de grafos, podemos verificar muitas propriedades dos grafos. Normalmente, tanto a busca em profundidade quanto a busca em largura podem ser usadas, mas na prática, a busca em profundidade é uma escolha melhor, porque é mais fácil de implementar. Nas seguintes aplicações, assumiremos que o grafo é não direcionado.

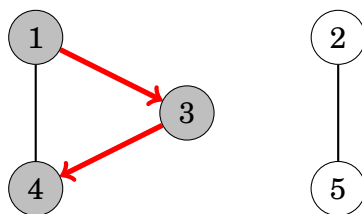
Verificação de Conectividade

Um grafo é conectado se houver um caminho entre quaisquer dois nós do grafo. Assim, podemos verificar se um grafo é conectado começando em um nó arbitrário e descobrindo se podemos alcançar todos os outros nós.

Por exemplo, no grafo:



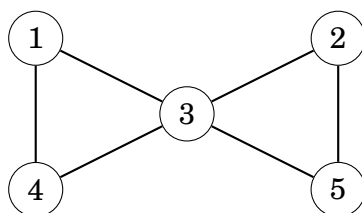
uma busca em profundidade a partir do nó 1 visita os seguintes nós:



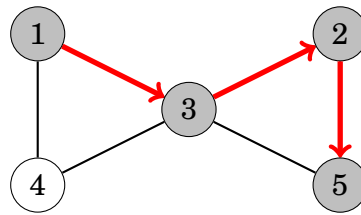
Como a busca não visitou todos os nós, podemos concluir que o grafo não é conectado. De forma semelhante, também podemos encontrar todos os componentes conectados de um grafo iterando pelos nós e sempre iniciando uma nova busca em profundidade se o nó atual ainda não pertence a nenhum componente.

Encontrando Ciclos

Um grafo contém um ciclo se, durante uma travessia de grafo, encontrarmos um nó cujo vizinho (diferente do nó anterior no caminho atual) já foi visitado. Por exemplo, o grafo:



contém dois ciclos e podemos encontrar um deles da seguinte forma:



Depois de passar do nó 2 para o nó 5, notamos que o vizinho 3 do nó 5 já foi visitado. Assim, o grafo contém um ciclo que passa pelo nó 3, por exemplo, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

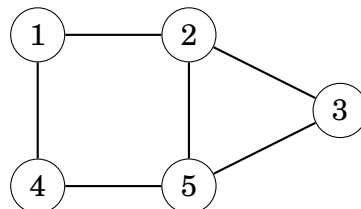
Outra forma de descobrir se um grafo contém um ciclo é simplesmente calcular o número de nós e arestas em cada componente. Se um componente contém c nós e nenhum ciclo, ele deve conter exatamente $c - 1$ arestas (portanto, deve ser uma árvore). Se houver c ou mais arestas, o componente certamente contém um ciclo.

Verificação de Bipartição

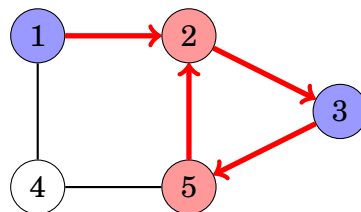
Um grafo é bipartido se seus nós podem ser coloridos usando duas cores, de modo que não haja nós adjacentes com a mesma cor. É surpreendentemente fácil verificar se um grafo é bipartido usando algoritmos de travessia de grafo.

A ideia é colorir o nó inicial de azul, todos os seus vizinhos de vermelho, todos os seus vizinhos de azul e assim por diante. Se em algum ponto da busca notarmos que dois nós adjacentes têm a mesma cor, isso significa que o grafo não é bipartido. Caso contrário, o grafo é bipartido e uma coloração foi encontrada.

Por exemplo, o grafo:



não é bipartido, porque uma busca a partir do nó 1 prossegue da seguinte forma:



Notamos que a cor de ambos os nós 2 e 5 é vermelha, sendo que eles são nós adjacentes no grafo. Assim, o grafo não é bipartido.

Este algoritmo sempre funciona, porque quando há apenas duas cores disponíveis, a cor do nó inicial em um componente determina as cores de todos os outros nós no componente. Não faz diferença se o nó inicial é vermelho ou azul.

Observe que, no caso geral, é difícil descobrir se os nós em um grafo podem ser coloridos usando k cores para que nenhum nó adjacente tenha a mesma cor. Mesmo quando $k = 3$, nenhum algoritmo eficiente é conhecido e o problema é NP-difícil.

Capítulo 13

Caminhos mínimos

Encontrar um caminho mínimo entre dois nós de um grafo é um problema importante que possui muitas aplicações práticas. Por exemplo, um problema natural relacionado a uma rede rodoviária é calcular o menor comprimento possível de uma rota entre duas cidades, dados os comprimentos das estradas.

Em um grafo não ponderado, o comprimento de um caminho é igual ao número de suas arestas, e podemos simplesmente usar a busca em largura para encontrar um caminho mínimo. No entanto, neste capítulo, vamos nos concentrar em grafos ponderados, onde algoritmos mais sofisticados são necessários para encontrar caminhos mínimos.

13.1 Algoritmo de Bellman–Ford

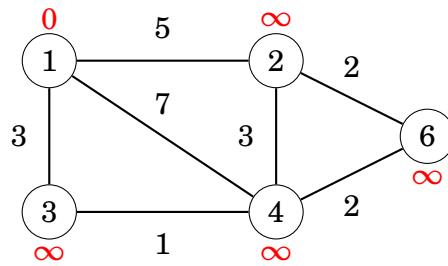
O **algoritmo de Bellman–Ford**¹ encontra caminhos mínimos de um nó inicial até todos os nós do grafo. O algoritmo pode processar todos os tipos de grafos, desde que o grafo não contenha um ciclo com comprimento negativo. Se o grafo contiver um ciclo negativo, o algoritmo pode detectar isso.

O algoritmo mantém o controle das distâncias do nó inicial até todos os nós do grafo. Inicialmente, a distância até o nó inicial é 0 e a distância até todos os outros nós é infinita. O algoritmo reduz as distâncias encontrando arestas que encurtam os caminhos até que não seja possível reduzir nenhuma distância.

Exemplo

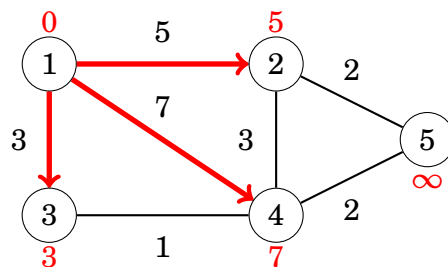
Vamos considerar como o algoritmo de Bellman–Ford funciona no seguinte grafo:

¹O algoritmo recebeu o nome de R. E. Bellman e L. R. Ford, que o publicaram independentemente em 1958 e 1956, respectivamente [5, 24].

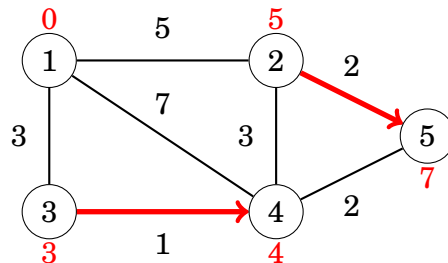


Cada nó do grafo recebe uma distância. Inicialmente, a distância até o nó inicial é 0, e a distância até todos os outros nós é infinita.

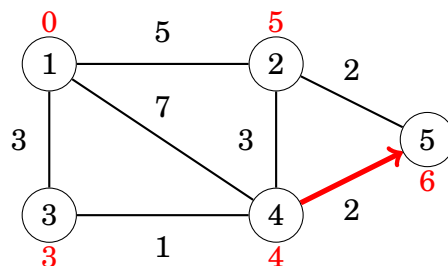
O algoritmo procura por arestas que reduzam as distâncias. Primeiro, todas as arestas do nó 1 reduzem as distâncias:



Depois disso, as arestas $2 \rightarrow 5$ e $3 \rightarrow 4$ reduzem as distâncias:

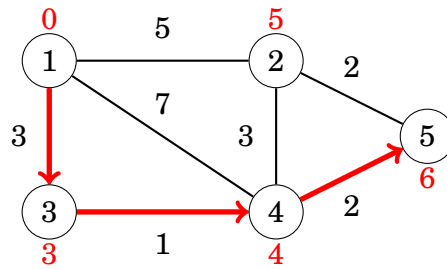


Finalmente, há mais uma mudança:



Depois disso, nenhuma aresta pode reduzir nenhuma distância. Isso significa que as distâncias são finais, e calculamos com sucesso as menores distâncias do nó inicial até todos os nós do grafo.

Por exemplo, a menor distância 3 do nó 1 ao nó 5 corresponde ao seguinte caminho:



Implementação

A seguinte implementação do algoritmo de Bellman–Ford determina as menores distâncias de um nó x até todos os nós do grafo. O código assume que o grafo é armazenado como uma lista de arestas `edges` que consiste em tuplas da forma (a, b, w) , significando que há uma aresta do nó a ao nó b com peso w .

O algoritmo consiste em $n - 1$ rodadas, e em cada rodada o algoritmo percorre todas as arestas do grafo e tenta reduzir as distâncias. O algoritmo constrói um vetor `distance` que conterá as distâncias de x até todos os nós do grafo. A constante `INF` denota uma distância infinita.

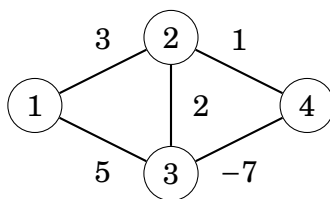
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

A complexidade de tempo do algoritmo é $O(nm)$, pois o algoritmo consiste em $n - 1$ rodadas e itera por todas as m arestas durante uma rodada. Se não houver ciclos negativos no grafo, todas as distâncias serão finais após $n - 1$ rodadas, pois cada caminho mínimo pode conter no máximo $n - 1$ arestas.

Na prática, as distâncias finais geralmente podem ser encontradas mais rapidamente do que em $n - 1$ rodadas. Assim, uma maneira possível de tornar o algoritmo mais eficiente é pará-lo se nenhuma distância puder ser reduzida durante uma rodada.

Ciclos negativos

O algoritmo de Bellman–Ford também pode ser usado para verificar se o grafo contém um ciclo com comprimento negativo. Por exemplo, o grafo



contém um ciclo negativo $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ com comprimento -4 .

Se o grafo contiver um ciclo negativo, podemos encurtar infinitamente qualquer caminho que contenha o ciclo repetindo o ciclo repetidamente. Assim, o conceito de um caminho mínimo não faz sentido nesta situação.

Um ciclo negativo pode ser detectado usando o algoritmo de Bellman–Ford executando o algoritmo por n rodadas. Se a última rodada reduzir alguma distância, o grafo contém um ciclo negativo. Observe que este algoritmo pode ser usado para pesquisar por um ciclo negativo em todo o grafo, independentemente do nó inicial.

Algoritmo SPFA

O **algoritmo SPFA** (“Shortest Path Faster Algorithm”) [20] é uma variante do algoritmo de Bellman–Ford, que geralmente é mais eficiente do que o algoritmo original. O algoritmo SPFA não percorre todas as arestas em cada rodada, mas, em vez disso, escolhe as arestas a serem examinadas de forma mais inteligente.

O algoritmo mantém uma fila de nós que podem ser usados para reduzir as distâncias. Primeiro, o algoritmo adiciona o nó inicial x à fila. Então, o algoritmo sempre processa o primeiro nó na fila, e quando uma aresta $a \rightarrow b$ reduz uma distância, o nó b é adicionado à fila.

A eficiência do algoritmo SPFA depende da estrutura do grafo: o algoritmo costuma ser eficiente, mas sua complexidade de tempo no pior caso ainda é $O(nm)$ e é possível criar entradas que tornam o algoritmo tão lento quanto o algoritmo de Bellman–Ford original.

13.2 Algoritmo de Dijkstra

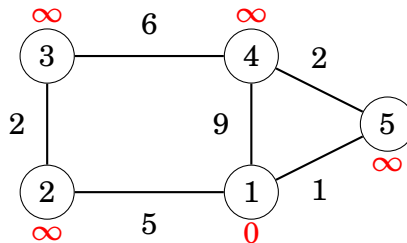
O **algoritmo de Dijkstra**² encontra os caminhos mínimos do nó inicial até todos os nós do grafo, assim como o algoritmo de Bellman–Ford. O benefício do algoritmo de Dijkstra é que ele é mais eficiente e pode ser usado para processar grafos grandes. No entanto, o algoritmo requer que não haja arestas com peso negativo no grafo.

Assim como o algoritmo de Bellman–Ford, o algoritmo de Dijkstra mantém distâncias para os nós e as reduz durante a pesquisa. O algoritmo de Dijkstra é eficiente porque ele só processa cada aresta do grafo uma vez, usando o fato de que não há arestas negativas.

²E. W. Dijkstra publicou o algoritmo em 1959 [14]; no entanto, seu artigo original não menciona como implementar o algoritmo de forma eficiente.

Exemplo

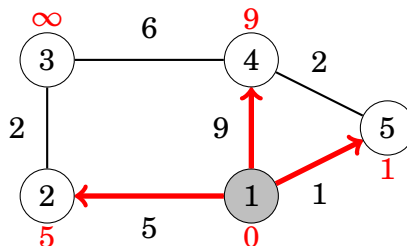
Vamos considerar como o algoritmo de Dijkstra funciona no seguinte grafo quando o nó inicial é o nó 1:



Assim como no algoritmo de Bellman–Ford, inicialmente a distância até o nó inicial é 0 e a distância até todos os outros nós é infinita.

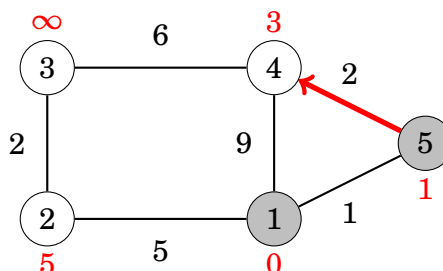
A cada etapa, o algoritmo de Dijkstra seleciona um nó que ainda não foi processado e cuja distância é a menor possível. O primeiro nó é o nó 1 com distância 0.

Quando um nó é selecionado, o algoritmo percorre todas as arestas que começam nesse nó e reduz as distâncias usando-as:

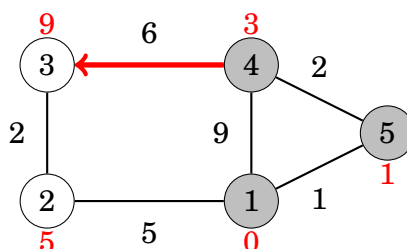


Neste caso, as arestas do nó 1 reduziram as distâncias dos nós 2, 4 e 5, cujas distâncias agora são 5, 9 e 1.

O próximo nó a ser processado é o nó 5 com distância 1. Isso reduz a distância até o nó 4 de 9 para 3:

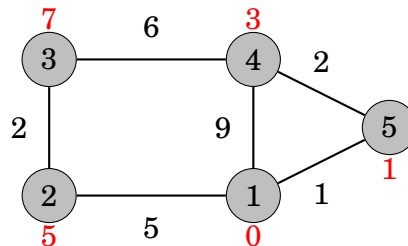


Depois disso, o próximo nó é o nó 4, que reduz a distância até o nó 3 para 9:



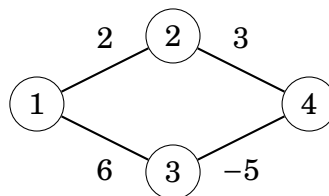
Uma propriedade notável do algoritmo de Dijkstra é que sempre que um nó é selecionado, sua distância é final. Por exemplo, neste ponto do algoritmo, as distâncias 0, 1 e 3 são as distâncias finais até os nós 1, 5 e 4.

Depois disso, o algoritmo processa os dois nós restantes, e as distâncias finais são as seguintes:



Arestas negativas

A eficiência do algoritmo de Dijkstra é baseada no fato de que o grafo não contém arestas negativas. Se houver uma aresta negativa, o algoritmo pode dar resultados incorretos. Como exemplo, considere o seguinte grafo:



O caminho mínimo do nó 1 ao nó 4 é $1 \rightarrow 3 \rightarrow 4$ e seu comprimento é 1. No entanto, o algoritmo de Dijkstra encontra o caminho $1 \rightarrow 2 \rightarrow 4$ seguindo as arestas de menor peso. O algoritmo não leva em consideração que no outro caminho, o peso -5 compensa o peso grande anterior 6.

Implementação

A seguinte implementação do algoritmo de Dijkstra calcula as distâncias mínimas de um nó x para outros nós do grafo. O grafo é armazenado como listas de adjacência de modo que $\text{adj}[a]$ contém um par (b, w) sempre que houver uma aresta do nó a ao nó b com peso w .

Uma implementação eficiente do algoritmo de Dijkstra exige que seja possível encontrar eficientemente o nó de distância mínima que ainda não foi processado. Uma estrutura de dados apropriada para isso é uma fila de prioridade que contém os nós ordenados por suas distâncias. Usando uma fila de prioridade, o próximo nó a ser processado pode ser recuperado em tempo logarítmico.

No código a seguir, a fila de prioridade q contém pares da forma $(-d, x)$, o que significa que a distância atual até o nó x é d . O vetor distance contém a distância até cada nó, e o vetor processed indica se um nó já foi processado. Inicialmente, a distância é 0 para x e ∞ para todos os outros nós.

```

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});
        }
    }
}
}

```

Observe que a fila de prioridade contém distâncias *negativas* até os nós. A razão para isso é que a versão padrão da fila de prioridade em C++ encontra elementos máximos, enquanto queremos encontrar elementos mínimos. Usando distâncias negativas, podemos usar diretamente a fila de prioridade padrão³. Observe também que pode haver várias instâncias do mesmo nó na fila de prioridade; no entanto, apenas a instância com a distância mínima será processada.

A complexidade de tempo da implementação acima é $O(n + m \log m)$, pois o algoritmo percorre todos os nós do grafo e adiciona para cada aresta no máximo uma distância à fila de prioridade.

13.3 Algoritmo de Floyd–Warshall

O **algoritmo de Floyd–Warshall**⁴ fornece uma maneira alternativa de abordar o problema de encontrar caminhos mínimos. Ao contrário dos outros algoritmos deste capítulo, ele encontra todos os caminhos mínimos entre os nós em uma única execução.

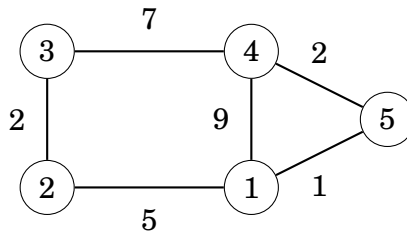
O algoritmo mantém uma matriz bidimensional que contém as distâncias entre os nós. Primeiro, as distâncias são calculadas usando apenas arestas diretas entre os nós, e depois disso, o algoritmo reduz as distâncias usando nós intermediários nos caminhos.

Exemplo

Vamos considerar como o algoritmo de Floyd–Warshall funciona no seguinte grafo:

³Claro, também poderíamos declarar a fila de prioridade como no Capítulo 4.5 e usar distâncias positivas, mas a implementação seria um pouco mais longa.

⁴O algoritmo recebeu o nome de R. W. Floyd e S. Warshall, que o publicaram independentemente em 1962 [23, 70].



Inicialmente, a distância de cada nó para si mesmo é 0, e a distância entre os nós a e b é x se houver uma aresta entre os nós a e b com peso x . Todas as outras distâncias são infinitas.

Neste grafo, a matriz inicial é a seguinte:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

O algoritmo consiste em rodadas consecutivas. Em cada rodada, o algoritmo seleciona um novo nó que pode atuar como um nó intermediário nos caminhos a partir de agora, e as distâncias são reduzidas usando este nó.

Na primeira rodada, o nó 1 é o novo nó intermediário. Há um novo caminho entre os nós 2 e 4 com comprimento 14, pois o nó 1 os conecta. Há também um novo caminho entre os nós 2 e 5 com comprimento 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

Na segunda rodada, o nó 2 é o novo nó intermediário. Isso cria novos caminhos entre os nós 1 e 3 e entre os nós 3 e 5:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

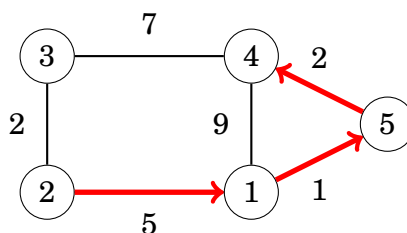
Na terceira rodada, o nó 3 é o novo nó intermediário. Há um novo caminho entre os nós 2 e 4:

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

O algoritmo continua assim, até que todos os nós tenham sido designados nós intermediários. Depois que o algoritmo termina, a matriz contém as distâncias mínimas entre quaisquer dois nós:

	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

Por exemplo, a matriz nos diz que a menor distância entre os nós 2 e 4 é 8. Isso corresponde ao seguinte caminho:



Implementação

A vantagem do algoritmo de Floyd–Warshall é que ele é fácil de implementar. O código a seguir constrói uma matriz de distância onde $\text{distance}[a][b]$ é a menor distância entre os nós a e b . Primeiro, o algoritmo inicializa distance usando a matriz de adjacência adj do grafo:

```

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;
    }
}

```

Depois disso, as menores distâncias podem ser encontradas da seguinte forma:

```

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]);  
}  
}
```

A complexidade de tempo do algoritmo é $O(n^3)$, pois ele contém três loops aninhados que percorrem os nós do grafo.

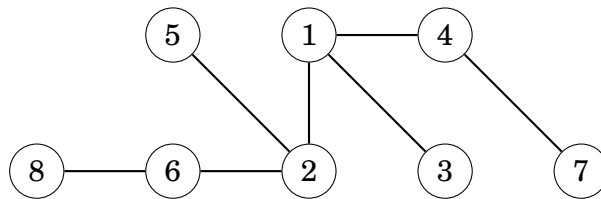
Como a implementação do algoritmo de Floyd–Warshall é simples, o algoritmo pode ser uma boa escolha mesmo que seja necessário encontrar apenas um único caminho mínimo no grafo. No entanto, o algoritmo só pode ser usado quando o grafo é tão pequeno que uma complexidade de tempo cúbica é rápida o suficiente.

Capítulo 14

Algoritmos em Árvores

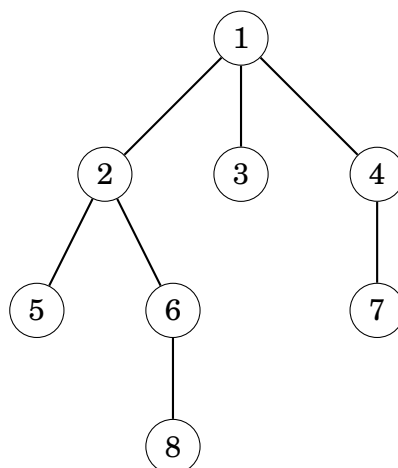
Uma **árvore** é um grafo conexo e acíclico que consiste em n nós e $n - 1$ arestas. Remover qualquer aresta de uma árvore a divide em dois componentes, e adicionar qualquer aresta a uma árvore cria um ciclo. Além disso, há sempre um caminho único entre quaisquer dois nós de uma árvore.

Por exemplo, a seguinte árvore consiste em 8 nós e 7 arestas:



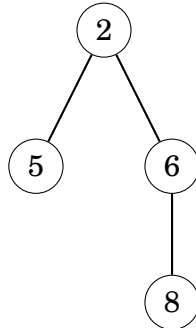
As **folhas** de uma árvore são os nós com grau 1, ou seja, com apenas um vizinho. Por exemplo, as folhas da árvore acima são os nós 3, 5, 7 e 8.

Em uma árvore **enraizada**, um dos nós é designado como a **raiz** da árvore, e todos os outros nós são colocados abaixo da raiz. Por exemplo, na árvore a seguir, o nó 1 é o nó raiz.



Em uma árvore enraizada, os **filhos** de um nó são seus vizinhos inferiores e o **pai** de um nó é seu vizinho superior. Cada nó tem exatamente um pai, exceto pela raiz que não possui pai. Por exemplo, na árvore acima, os filhos do nó 2 são os nós 5 e 6, e seu pai é o nó 1.

A estrutura de uma árvore enraizada é *recursiva*: cada nó da árvore atua como a raiz de uma **subárvore** que contém o próprio nó e todos os nós que estão nas subárvores de seus filhos. Por exemplo, na árvore acima, a subárvore do nó 2 consiste nos nós 2, 5, 6 e 8:



14.1 Travessia de Árvore

Algoritmos gerais de travessia de grafos podem ser usados para percorrer os nós de uma árvore. No entanto, a travessia de uma árvore é mais fácil de implementar do que a de um grafo geral, pois não há ciclos na árvore e não é possível alcançar um nó por múltiplas direções.

A maneira típica de percorrer uma árvore é iniciar uma busca em profundidade em um nó arbitrário. A seguinte função recursiva pode ser usada:

```
void dfs(int s, int e) {  
    // processa o nó s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

A função recebe dois parâmetros: o nó atual s e o nó anterior e . O objetivo do parâmetro e é garantir que a busca se mova apenas para nós que ainda não foram visitados.

A seguinte chamada de função inicia a busca no nó x :

```
dfs(x, 0);
```

Na primeira chamada $e = 0$, pois não há nó anterior, e é permitido prosseguir em qualquer direção na árvore.

Programação Dinâmica

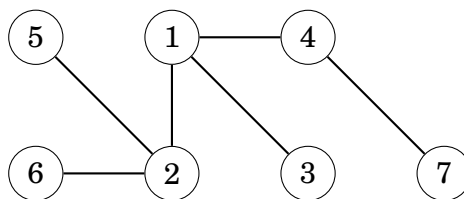
A programação dinâmica pode ser usada para calcular algumas informações durante a travessia de uma árvore. Usando programação dinâmica, podemos, por exemplo, calcular em tempo $O(n)$ para cada nó de uma árvore enraizada o número de nós em sua subárvore ou o comprimento do caminho mais longo do nó até uma folha.

Como exemplo, vamos calcular para cada nó s um valor $\text{count}[s]$: o número de nós em sua subárvore. A subárvore contém o próprio nó e todos os nós nas subárvores de seus filhos, então podemos calcular o número de nós recursivamente usando o seguinte código:

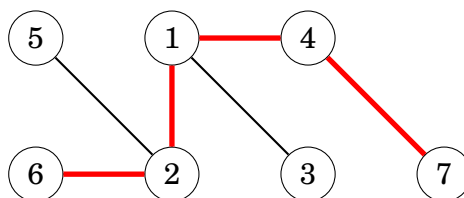
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Diâmetro

O **diâmetro** de uma árvore é o comprimento máximo de um caminho entre dois nós. Por exemplo, considere a seguinte árvore:



O diâmetro desta árvore é 4, o que corresponde ao seguinte caminho:



Observe que pode haver vários caminhos de comprimento máximo. No caminho acima, poderíamos substituir o nó 6 pelo nó 5 para obter outro caminho com comprimento 4.

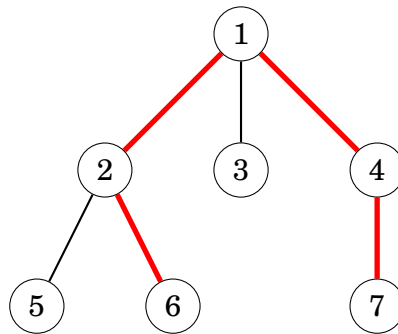
A seguir, discutiremos dois algoritmos de tempo $O(n)$ para calcular o diâmetro de uma árvore. O primeiro algoritmo é baseado em programação dinâmica e o segundo algoritmo usa duas buscas em profundidade.

Algoritmo 1

Uma maneira geral de abordar muitos problemas de árvore é primeiro enraizar a árvore arbitrariamente. Depois disso, podemos tentar resolver o problema separadamente para cada subárvore. Nosso primeiro algoritmo para calcular o diâmetro é baseado nessa ideia.

Uma observação importante é que todo caminho em uma árvore enraizada tem um *ponto mais alto*: o nó mais alto que pertence ao caminho. Assim, podemos calcular para cada nó o comprimento do caminho mais longo cujo ponto mais alto é o nó. Um desses caminhos corresponde ao diâmetro da árvore.

Por exemplo, na árvore a seguir, o nó 1 é o ponto mais alto no caminho que corresponde ao diâmetro:



Calculamos para cada nó x dois valores:

- $\text{toLeaf}(x)$: o comprimento máximo de um caminho de x até qualquer folha
- $\text{maxLength}(x)$: o comprimento máximo de um caminho cujo ponto mais alto é x

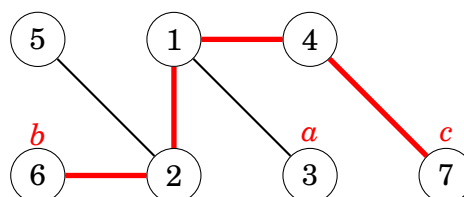
Por exemplo, na árvore acima, $\text{toLeaf}(1) = 2$, pois há um caminho $1 \rightarrow 2 \rightarrow 6$, e $\text{maxLength}(1) = 4$, pois há um caminho $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$. Neste caso, $\text{maxLength}(1)$ é igual ao diâmetro.

A programação dinâmica pode ser usada para calcular os valores acima para todos os nós em tempo $O(n)$. Primeiro, para calcular $\text{toLeaf}(x)$, percorremos os filhos de x , escolhemos um filho c com o máximo $\text{toLeaf}(c)$ e adicionamos um a esse valor. Então, para calcular $\text{maxLength}(x)$, escolhemos dois filhos distintos a e b tais que a soma $\text{toLeaf}(a) + \text{toLeaf}(b)$ seja máxima e adicionamos dois a essa soma.

Algoritmo 2

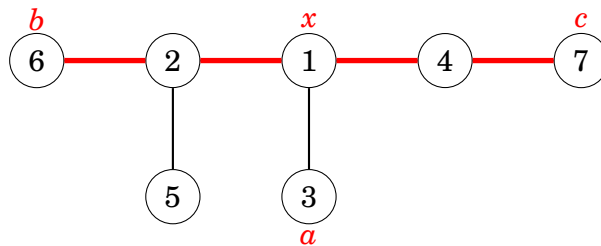
Outra maneira eficiente de calcular o diâmetro de uma árvore é baseada em duas buscas em profundidade. Primeiro, escolhemos um nó arbitrário a na árvore e encontramos o nó b mais distante de a . Então, encontramos o nó c mais distante de b . O diâmetro da árvore é a distância entre b e c .

No seguinte grafo, a , b e c podem ser:



Este é um método elegante, mas por que ele funciona?

Ajuda desenhar a árvore de forma diferente para que o caminho que corresponde ao diâmetro seja horizontal e todos os outros nós pendurados nele:

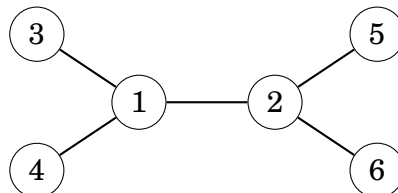


O nó x indica o local onde o caminho do nó a se junta ao caminho que corresponde ao diâmetro. O nó mais distante de a é o nó b , o nó c ou algum outro nó que esteja pelo menos tão distante do nó x . Assim, este nó é sempre uma escolha válida para um ponto final de um caminho que corresponde ao diâmetro.

14.3 Todos os Caminhos Mais Longos

Nosso próximo problema é calcular para cada nó na árvore o comprimento máximo de um caminho que começa naquele nó. Isso pode ser visto como uma generalização do problema do diâmetro da árvore, porque o maior desses comprimentos é igual ao diâmetro da árvore. Este problema também pode ser resolvido em tempo $O(n)$.

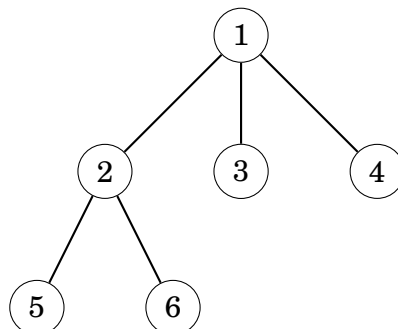
Como exemplo, considere a seguinte árvore:



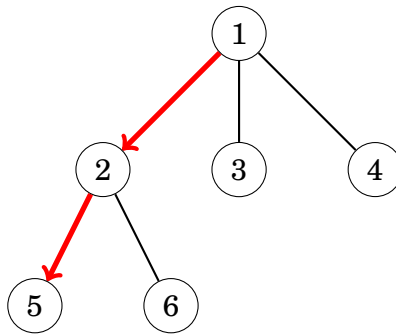
Seja $\text{maxLength}(x)$ o comprimento máximo de um caminho que começa no nó x . Por exemplo, na árvore acima, $\text{maxLength}(4) = 3$, porque há um caminho $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Aqui está uma tabela completa dos valores:

nó x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Também neste problema, um bom ponto de partida para resolvê-lo é enraizar a árvore arbitrariamente:

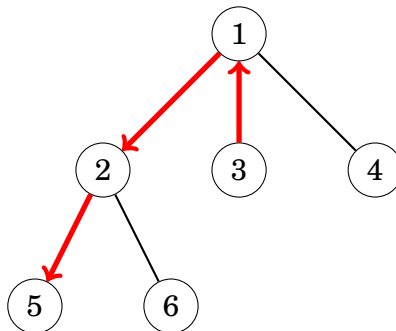


A primeira parte do problema é calcular para cada nó x o comprimento máximo de um caminho que passa por um filho de x . Por exemplo, o caminho mais longo do nó 1 passa por seu filho 2:

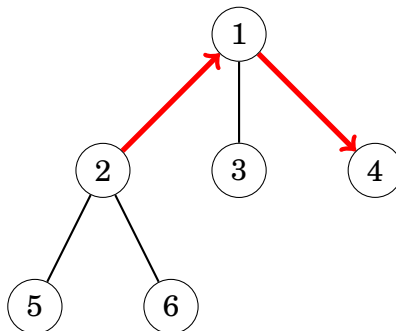


Esta parte é fácil de resolver em tempo $O(n)$, pois podemos usar a programação dinâmica como fizemos anteriormente.

Então, a segunda parte do problema é calcular para cada nó x o comprimento máximo de um caminho através de seu pai p . Por exemplo, o caminho mais longo do nó 3 passa por seu pai 1:



À primeira vista, parece que devemos escolher o caminho mais longo de p . No entanto, isso *não* funciona sempre, porque o caminho mais longo de p pode passar por x . Aqui está um exemplo desta situação:



Ainda assim, podemos resolver a segunda parte em tempo $O(n)$ armazenando *dois* comprimentos máximos para cada nó x :

- $\text{maxLength}_1(x)$: o comprimento máximo de um caminho de x
- $\text{maxLength}_2(x)$: o comprimento máximo de um caminho de x em outra direção que não o primeiro caminho

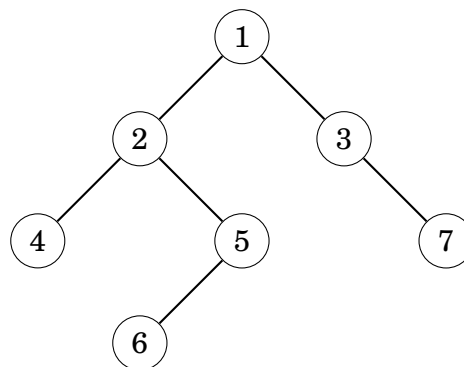
Por exemplo, no grafo acima, $\text{maxLength}_1(1) = 2$ usando o caminho $1 \rightarrow 2 \rightarrow 5$ e $\text{maxLength}_2(1) = 1$ usando o caminho $1 \rightarrow 3$.

Finalmente, se o caminho que corresponde a $\text{maxLength}_1(p)$ passa por x , concluímos que o comprimento máximo é $\text{maxLength}_2(p) + 1$, caso contrário, o comprimento máximo é $\text{maxLength}_1(p) + 1$.

14.4 Árvores Binárias

Uma **árvore binária** é uma árvore enraizada onde cada nó possui uma subárvore esquerda e uma subárvore direita. É possível que uma subárvore de um nó esteja vazia. Assim, todo nó em uma árvore binária tem zero, um ou dois filhos.

Por exemplo, a seguinte árvore é uma árvore binária:



Os nós de uma árvore binária possuem três ordenações naturais que correspondem a diferentes maneiras de percorrer recursivamente a árvore:

- **pré-ordem:** primeiro processa a raiz, depois percorre a subárvore esquerda e, em seguida, percorre a subárvore direita
- **em-ordem:** primeiro percorre a subárvore esquerda, depois processa a raiz e, em seguida, percorre a subárvore direita
- **pós-ordem:** primeiro percorre a subárvore esquerda, depois percorre a subárvore direita e, em seguida, processa a raiz

Para a árvore acima, os nós em pré-ordem são $[1, 2, 4, 5, 6, 3, 7]$, em ordem $[4, 2, 6, 5, 1, 3, 7]$ e em pós-ordem $[4, 6, 5, 2, 7, 3, 1]$.

Se conhecermos a pré-ordem e a ordem de uma árvore, podemos reconstruir a estrutura exata da árvore. Por exemplo, a árvore acima é a única árvore possível com pré-ordem $[1, 2, 4, 5, 6, 3, 7]$ e ordem $[4, 2, 6, 5, 1, 3, 7]$. De forma semelhante, a pós-ordem e a ordem também determinam a estrutura de uma árvore.

No entanto, a situação é diferente se conhecermos apenas a pré-ordem e a pós-ordem de uma árvore. Nesse caso, pode haver mais de uma árvore que corresponda às ordenações. Por exemplo, em ambas as árvores



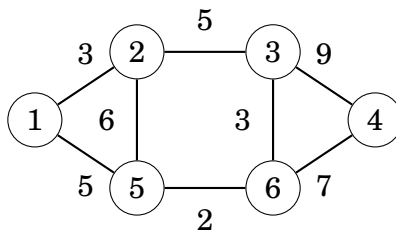
a pré-ordem é $[1,2]$ e a pós-ordem é $[2,1]$, mas as estruturas das árvores são diferentes.

Capítulo 15

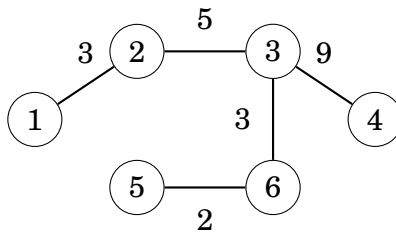
Árvores geradoras

Uma **árvore geradora** de um grafo é composta por todos os nós do grafo e por algumas de suas arestas, de forma que haja um caminho entre quaisquer dois nós. Assim como as árvores em geral, as árvores geradoras são conectadas e acíclicas. Normalmente, existem várias maneiras de construir uma árvore geradora.

Por exemplo, considere o seguinte grafo:

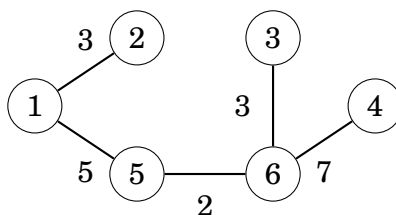


Uma árvore geradora para o grafo é a seguinte:

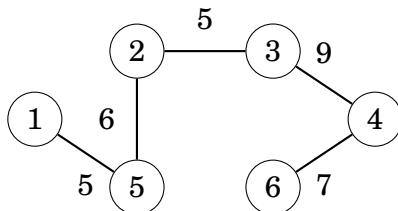


O peso de uma árvore geradora é a soma dos pesos de suas arestas. Por exemplo, o peso da árvore geradora acima é $3 + 5 + 9 + 3 + 2 = 22$.

Uma **Árvore geradora mínima** é uma árvore geradora cujo peso é o menor possível. O peso de uma árvore geradora mínima para o grafo de exemplo é 20, e tal árvore pode ser construída da seguinte forma:



De maneira semelhante, uma **Árvore geradora máxima** é uma árvore geradora cujo peso é o maior possível. O peso de uma árvore geradora máxima para o grafo de exemplo é 32:



Note que um grafo pode ter várias árvores geradoras mínimas e máximas, então as árvores não são únicas.

Acontece que vários métodos gulosos podem ser usados para construir árvores geradoras mínimas e máximas. Neste capítulo, discutimos dois algoritmos que processam as arestas do grafo ordenadas pelos seus pesos. Nosso foco é encontrar árvores geradoras mínimas, mas os mesmos algoritmos podem encontrar árvores geradoras máximas processando as arestas em ordem inversa.

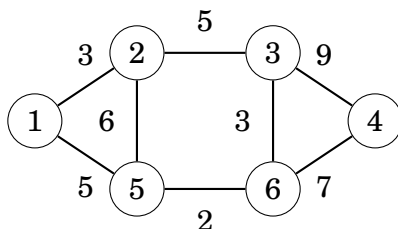
15.1 Algoritmo de Kruskal

No **Algoritmo de Kruskal**¹, a árvore geradora inicial contém apenas os nós do grafo e não contém nenhuma aresta. Em seguida, o algoritmo percorre as arestas ordenadas pelos seus pesos, e sempre adiciona uma aresta à árvore se ela não criar um ciclo.

O algoritmo mantém os componentes da árvore. Inicialmente, cada nó do grafo pertence a um componente separado. Sempre que uma aresta é adicionada à árvore, dois componentes são unidos. Finalmente, todos os nós pertencem ao mesmo componente, e uma árvore geradora mínima foi encontrada.

Exemplo

Vamos considerar como o algoritmo de Kruskal processa o seguinte grafo:



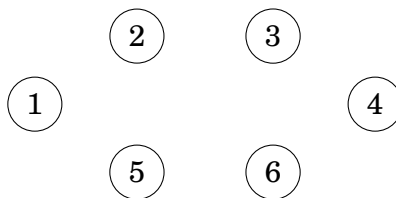
O primeiro passo do algoritmo é ordenar as arestas em ordem crescente de seus pesos. O resultado é a seguinte lista:

¹O algoritmo foi publicado em 1956 por J. B. Kruskal [48].

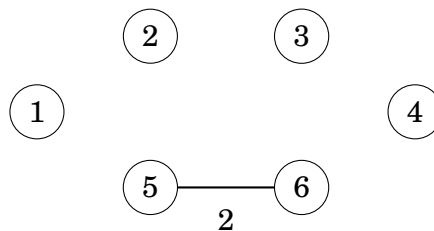
aresta	peso
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Depois disso, o algoritmo percorre a lista e adiciona cada aresta à árvore se ela unir dois componentes separados.

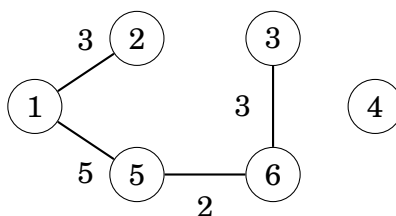
Inicialmente, cada nó está em seu próprio componente:



A primeira aresta a ser adicionada à árvore é a aresta 5-6 que cria um componente {5, 6} ao unir os componentes {5} e {6}:



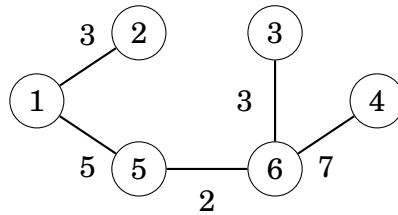
Após isso, as arestas 1-2, 3-6 e 1-5 são adicionadas de maneira similar:



Após esses passos, a maioria dos componentes foi unida e há dois componentes na árvore: {1, 2, 3, 5, 6} e {4}.

A próxima aresta na lista é a aresta 2-3, mas ela não será incluída na árvore, pois os nós 2 e 3 já estão no mesmo componente. Pelo mesmo motivo, a aresta 2-5 também não será incluída na árvore.

Finalmente, a aresta 4–6 será incluída na árvore:

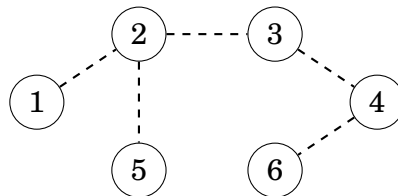


Após isso, o algoritmo não adicionará mais arestas, porque o grafo está conectado e há um caminho entre quaisquer dois nós. O grafo resultante é uma árvore geradora mínima com peso $2 + 3 + 3 + 5 + 7 = 20$.

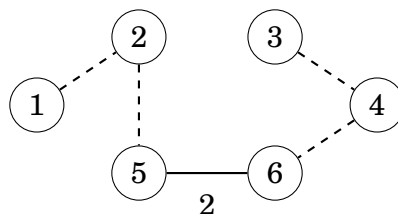
Por que isso funciona?

É uma boa pergunta por que o algoritmo de Kruskal funciona. Por que a estratégia gulosa garante que encontraremos uma árvore geradora mínima?

Vamos ver o que acontece se a aresta de menor peso do grafo *não* estiver incluída na árvore geradora. Por exemplo, suponha que uma árvore geradora para o grafo anterior não contivesse a aresta de menor peso 5–6. Não sabemos a estrutura exata de tal árvore geradora, mas em qualquer caso ela tem que conter algumas arestas. Assuma que a árvore seria como a seguinte:



Entretanto, não é possível que a árvore acima seja uma árvore geradora mínima para o grafo. O motivo disso é que podemos remover uma aresta da árvore e substituí-la pela aresta de peso mínimo 5–6. Isso produz uma árvore geradora cujo peso é *menor*:



Por essa razão, é sempre ideal incluir a aresta de menor peso na árvore para produzir uma árvore geradora mínima. Usando um argumento semelhante, podemos mostrar que também é ideal adicionar a próxima aresta em ordem de peso à árvore, e assim por diante. Portanto, o algoritmo de Kruskal funciona corretamente e sempre produz uma árvore geradora mínima.

Implementação

Ao implementar o algoritmo de Kruskal, é conveniente usar a representação de lista de arestas do grafo. A primeira fase do algoritmo ordena as arestas na lista em tempo $O(m \log m)$. Após isso, a segunda fase do algoritmo constrói a árvore geradora mínima da seguinte forma:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

O laço percorre as arestas na lista e sempre processa uma aresta $a-b$ onde a e b são dois nós. Duas funções são necessárias: a função `same` determina se a e b estão no mesmo componente, e a função `unite` une os componentes que contêm a e b .

O problema é como implementar eficientemente as funções `same` e `unite`. Uma possibilidade é implementar a função `same` como uma travessia de grafo e verificar se podemos ir do nó a ao nó b . No entanto, a complexidade de tempo de tal função seria $O(n + m)$ e o algoritmo resultante seria lento, pois a função `same` será chamada para cada aresta no grafo.

Resolveremos o problema usando uma estrutura `union-find` que implementa ambas as funções em tempo $O(\log n)$. Assim, a complexidade de tempo do algoritmo de Kruskal será $O(m \log n)$ após ordenar a lista de arestas.

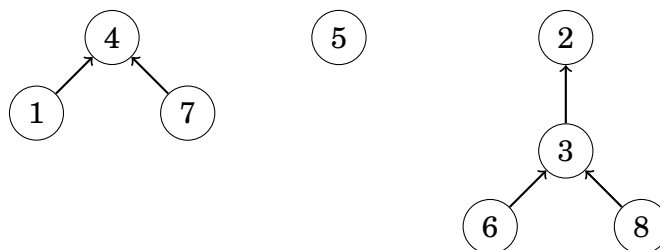
15.2 Estrutura Union-Find

Uma **Estrutura union-find** mantém uma coleção de conjuntos. Os conjuntos são disjuntos, então nenhum elemento pertence a mais de um conjunto. Duas operações de tempo $O(\log n)$ são suportadas: a operação `unite` une dois conjuntos e a operação `find` encontra o representante do conjunto que contém um determinado elemento².

Estrutura

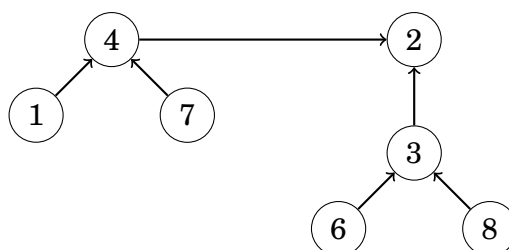
Em uma estrutura `union-find`, um elemento em cada conjunto é o representante do conjunto, e há uma cadeia de qualquer outro elemento do conjunto para o representante. Por exemplo, suponha que os conjuntos sejam $\{1, 4, 7\}$, $\{5\}$ e $\{2, 3, 6, 8\}$:

²A estrutura apresentada aqui foi introduzida em 1971 por J. D. Hopcroft e J. D. Ullman [38]. Posteriormente, em 1975, R. E. Tarjan estudou uma variante mais sofisticada da estrutura [64] que é discutida em muitos livros de algoritmos hoje em dia.



Neste caso, os representantes dos conjuntos são 4, 5 e 2. Podemos encontrar o representante de qualquer elemento seguindo a cadeia que começa no elemento. Por exemplo, o elemento 2 é o representante do elemento 6, pois seguimos a cadeia $6 \rightarrow 3 \rightarrow 2$. Dois elementos pertencem ao mesmo conjunto exatamente quando seus representantes são os mesmos.

Dois conjuntos podem ser unidos conectando o representante de um conjunto ao representante do outro conjunto. Por exemplo, os conjuntos $\{1, 4, 7\}$ e $\{2, 3, 6, 8\}$ podem ser unidos da seguinte forma:



O conjunto resultante contém os elementos $\{1, 2, 3, 4, 6, 7, 8\}$. A partir daqui, o elemento 2 é o representante de todo o conjunto e o antigo representante 4 aponta para o elemento 2.

A eficiência da estrutura union-find depende de como os conjuntos são unidos. Acontece que podemos seguir uma estratégia simples: sempre conectar o representante do conjunto *menor* ao representante do conjunto *maior* (ou, se os conjuntos forem do mesmo tamanho, podemos fazer uma escolha arbitrária). Usando esta estratégia, o comprimento de qualquer cadeia será $O(\log n)$, para que possamos encontrar o representante de qualquer elemento de forma eficiente, seguindo a cadeia correspondente.

Implementação

A estrutura union-find pode ser implementada usando arrays. Na implementação a seguir, o array `link` contém, para cada elemento, o próximo elemento na cadeia ou o próprio elemento se for um representante, e o array `size` indica para cada representante o tamanho do conjunto correspondente.

Inicialmente, cada elemento pertence a um conjunto separado:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

A função `find` retorna o representante para um elemento x . O representante pode ser encontrado seguindo a cadeia que começa em x .

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

A função `same` verifica se os elementos a e b pertencem ao mesmo conjunto. Isso pode ser feito facilmente usando a função `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

A função `unite` une os conjuntos que contêm os elementos a e b (os elementos devem estar em conjuntos diferentes). A função primeiro encontra os representantes dos conjuntos e então conecta o conjunto menor ao conjunto maior.

```
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;
}
```

A complexidade de tempo da função `find` é $O(\log n)$, assumindo que o comprimento de cada cadeia é $O(\log n)$. Neste caso, as funções `same` e `unite` também funcionam em tempo $O(\log n)$. A função `unite` garante que o comprimento de cada cadeia seja $O(\log n)$ conectando o conjunto menor ao conjunto maior.

15.3 Algoritmo de Prim

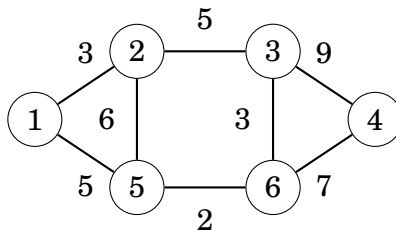
³ é um método alternativo para encontrar uma árvore geradora mínima. O algoritmo primeiro adiciona um nó arbitrário à árvore. Depois disso, o algoritmo sempre escolhe uma aresta de peso mínimo que adiciona um novo nó à árvore. Finalmente, todos os nós terão sido adicionados à árvore e uma árvore geradora mínima terá sido encontrada.

O algoritmo de Prim se assemelha ao algoritmo de Dijkstra. A diferença é que o algoritmo de Dijkstra sempre seleciona uma aresta cuja distância do nó inicial é mínima, mas o algoritmo de Prim simplesmente seleciona a aresta de peso mínimo que adiciona um novo nó à árvore.

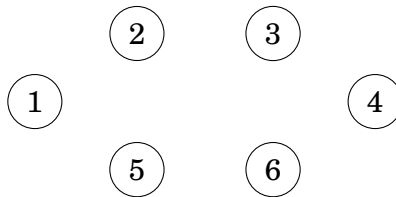
Exemplo

Vamos considerar como o algoritmo de Prim funciona no seguinte grafo:

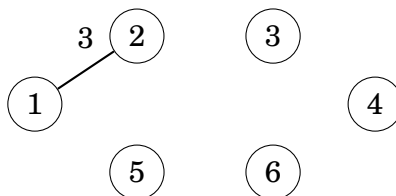
³O algoritmo foi nomeado em homenagem a R. C. Prim, que o publicou em 1957 [54]. No entanto, o mesmo algoritmo já havia sido descoberto em 1930 por V. Jarník.



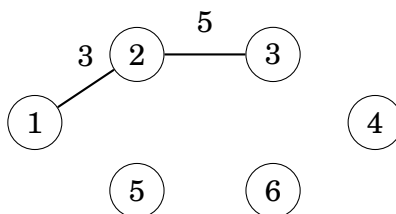
Inicialmente, não há arestas entre os nós:



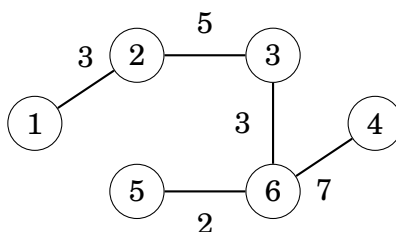
Um nó arbitrário pode ser o nó inicial, então vamos escolher o nó 1. Primeiro, adicionamos o nó 2 que está conectado por uma aresta de peso 3:



Depois disso, existem duas arestas com peso 5, então podemos adicionar o nó 3 ou o nó 5 à árvore. Vamos adicionar o nó 3 primeiro:



O processo continua até que todos os nós tenham sido incluídos na árvore:



Implementação

Assim como o algoritmo de Dijkstra, o algoritmo de Prim pode ser implementado eficientemente usando uma fila de prioridade. A fila de prioridade deve conter

todos os nós que podem ser conectados ao componente atual usando uma única aresta, em ordem crescente dos pesos das arestas correspondentes.

A complexidade de tempo do algoritmo de Prim é $O(n + m \log m)$ que é igual à complexidade de tempo do algoritmo de Dijkstra. Na prática, os algoritmos de Prim e Kruskal são ambos eficientes, e a escolha do algoritmo é uma questão de gosto. Ainda assim, a maioria dos programadores competitivos usam o algoritmo de Kruskal.

Capítulo 16

Grafos direcionados

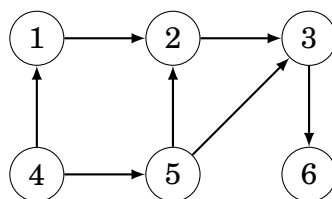
Neste capítulo, nos concentramos em duas classes de grafos direcionados:

- **Grafos acíclicos:** Não há ciclos no grafo, portanto, não há caminho de nenhum nó para ele mesmo¹.
- **Grafos de sucessão:** O grau de saída de cada nó é 1, então cada nó tem um sucessor único.

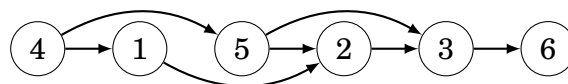
Verifica-se que em ambos os casos, podemos projetar algoritmos eficientes que são baseados nas propriedades especiais dos grafos.

16.1 Ordenação topológica

Uma **ordenação topológica** é uma ordenação dos nós de um grafo direcionado tal que se houver um caminho do nó a para o nó b , então o nó a aparece antes do nó b na ordenação. Por exemplo, para o grafo



uma ordenação topológica é [4, 1, 5, 2, 3, 6]:



Um grafo acíclico sempre tem uma ordenação topológica. No entanto, se o grafo contém um ciclo, não é possível formar uma ordenação topológica, porque nenhum nó do ciclo pode aparecer antes dos outros nós do ciclo na ordenação. Acontece que a busca em profundidade (DFS) pode ser usada para verificar se um grafo direcionado contém um ciclo e, se não contiver um ciclo, para construir uma ordenação topológica.

¹Grafos direcionados acíclicos são às vezes chamados de DAGs.

Algoritmo

A ideia é percorrer os nós do grafo e sempre iniciar uma busca em profundidade no nó atual se ele ainda não tiver sido processado. Durante as buscas, os nós têm três estados possíveis:

- estado 0: o nó não foi processado (branco)
- estado 1: o nó está sendo processado (cinza claro)
- estado 2: o nó foi processado (cinza escuro)

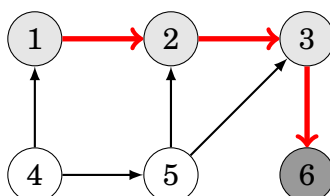
Inicialmente, o estado de cada nó é 0. Quando uma busca chega a um nó pela primeira vez, seu estado muda para 1. Finalmente, depois que todos os sucessores do nó tiverem sido processados, seu estado muda para 2.

Se o grafo contém um ciclo, descobriremos isso durante a busca, porque mais cedo ou mais tarde chegaremos a um nó cujo estado é 1. Nesse caso, não é possível construir uma ordenação topológica.

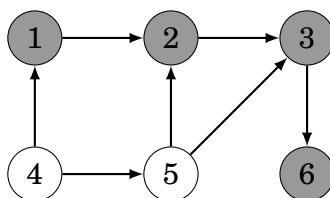
Se o grafo não contiver um ciclo, podemos construir uma ordenação topológica adicionando cada nó a uma lista quando o estado do nó se tornar 2. Esta lista em ordem inversa é uma ordenação topológica.

Exemplo 1

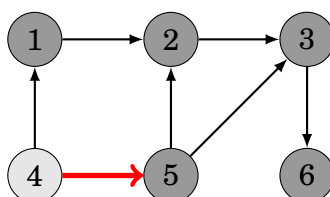
No grafo de exemplo, a busca prossegue primeiro do nó 1 ao nó 6:



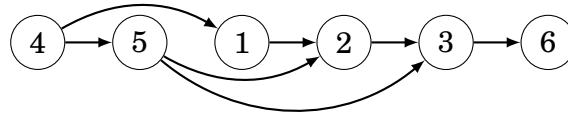
Agora, o nó 6 foi processado e adicionado à lista. Depois disso, os nós 3, 2 e 1 também são adicionados à lista:



Neste ponto, a lista é [6,3,2,1]. A próxima pesquisa começa no nó 4:



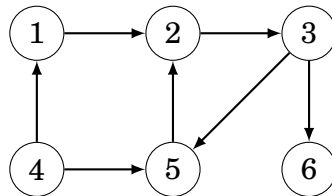
Assim, a lista final é [6,3,2,1,5,4]. Processamos todos os nós, então uma ordenação topológica foi encontrada. A ordenação topológica é a lista inversa [4,5,1,2,3,6]:



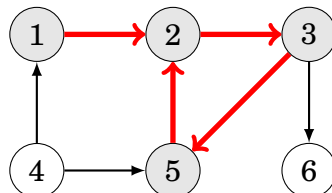
Observe que uma ordenação topológica não é única, e pode haver várias ordenações topológicas para um grafo.

Exemplo 2

Vamos agora considerar um grafo para o qual nós não podemos construir uma ordenação topológica, porque o grafo contém um ciclo:



A pesquisa prossegue da seguinte forma:



A busca chega ao nó 2, cujo estado é 1, o que significa que o grafo contém um ciclo. Neste exemplo, há um ciclo $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

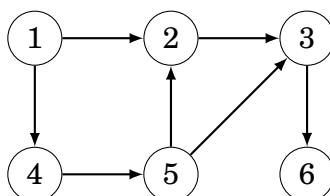
16.2 Programação dinâmica

Se um grafo direcionado for acíclico, a programação dinâmica pode ser aplicada a ele. Por exemplo, podemos resolver eficientemente os seguintes problemas relativos a caminhos de um nó inicial para um nó final:

- quantos caminhos diferentes existem?
- qual é o caminho mais curto/longo?
- qual é o número mínimo/máximo de arestas em um caminho?
- quais nós certamente aparecem em qualquer caminho?

Contando o número de caminhos

Como exemplo, vamos calcular o número de caminhos do nó 1 ao nó 6 no seguinte grafo:



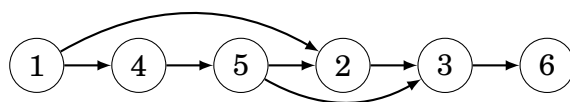
Existem no total três desses caminhos:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

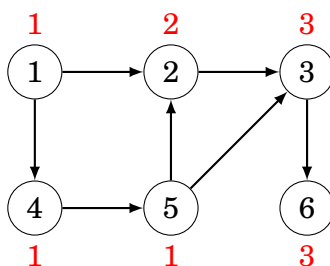
Seja $\text{paths}(x)$ o número de caminhos de nó 1 ao nó x . Como caso base, $\text{paths}(1) = 1$. Então, para calcular outros valores de $\text{paths}(x)$, podemos usar a recursão

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

onde a_1, a_2, \dots, a_k são os nós dos quais há uma aresta para x . Como o grafo é acíclico, os valores de $\text{paths}(x)$ pode ser calculado na ordem de uma ordenação topológica. Uma ordenação topológica para o grafo acima é a seguinte:



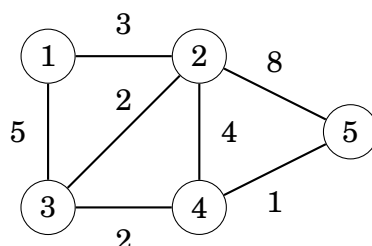
Portanto, os números de caminhos são os seguintes:



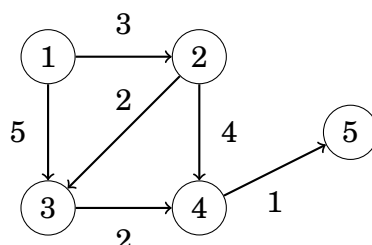
Por exemplo, para calcular o valor de $\text{paths}(3)$, podemos usar a fórmula $\text{paths}(2) + \text{paths}(5)$, porque existem arestas dos nós 2 e 5 para o nó 3. Como $\text{paths}(2) = 2$ e $\text{paths}(5) = 1$, concluímos que $\text{paths}(3) = 3$.

Estendendo o algoritmo de Dijkstra

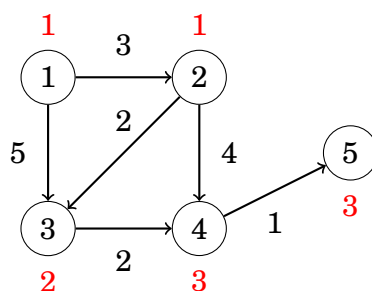
Um subproduto do algoritmo de Dijkstra é um grafo direcionado e acíclico que indica para cada nó do grafo original as maneiras possíveis de alcançar o nó usando um caminho mais curto do nó inicial. A programação dinâmica pode ser aplicada a esse grafo. Por exemplo, no grafo



os caminhos mais curtos do nó 1 podem usar as seguintes arestas:



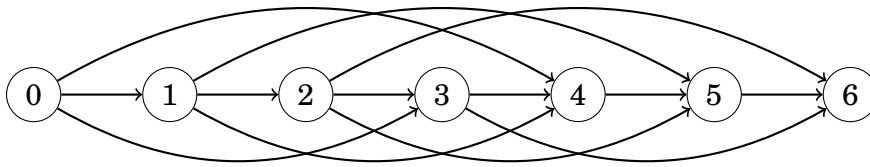
Agora podemos, por exemplo, calcular o número de caminhos mais curtos do nó 1 ao nó 5 usando programação dinâmica:



Representando problemas como grafos

Na verdade, qualquer problema de programação dinâmica pode ser representado como um grafo direcionado e acíclico. Em tal grafo, cada nó corresponde a um estado de programação dinâmica e as arestas indicam como os estados dependem uns dos outros.

Como exemplo, considere o problema de formar uma soma de dinheiro n usando moedas $\{c_1, c_2, \dots, c_k\}$. Neste problema, podemos construir um grafo onde cada nó corresponde a uma soma de dinheiro, e as arestas mostram como as moedas podem ser escolhidas. Por exemplo, para moedas $\{1, 3, 4\}$ e $n = 6$, o grafo é o seguinte:



Usando esta representação, o caminho mais curto do nó 0 ao nó n corresponde a uma solução com o número mínimo de moedas, e o número total de caminhos do nó 0 ao nó n é igual ao número total de soluções.

16.3 Caminhos sucessores

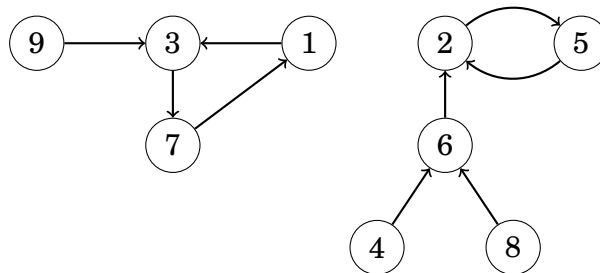
No restante deste capítulo, vamos nos concentrar em **grafos de sucessores**. Nesses grafos, o grau de saída de cada nó é 1, ou seja, exatamente uma aresta começa em cada nó. Um grafo de sucessores consiste em um ou mais componentes, cada um contendo um ciclo e alguns caminhos que levam a ele.

Os grafos de sucessores são às vezes chamados de **grafos funcionais**. A razão para isso é que qualquer grafo de sucessores corresponde a uma função que define as arestas do grafo. O parâmetro para a função é um nó do grafo, e a função fornece o sucessor desse nó.

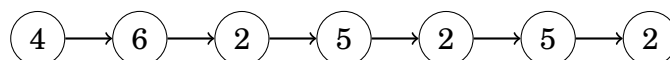
Por exemplo, a função

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

define o seguinte grafo:



Como cada nó de um grafo de sucessores tem um único sucessor, também podemos definir uma função $\text{succ}(x, k)$ que retorna o nó ao qual chegaremos se começarmos no nó x e andarmos k passos para frente. Por exemplo, no grafo acima $\text{succ}(4, 6) = 2$, porque chegaremos ao nó 2 caminhando 6 passos a partir do nó 4:



Uma maneira direta de calcular um valor de $\text{succ}(x, k)$ é começar no nó x e andar k passos para frente, o que leva um tempo $O(k)$. No entanto, usando pré-processamento, qualquer valor de $\text{succ}(x, k)$ pode ser calculado em apenas um tempo $O(\log k)$.

A ideia é pré-calculer todos os valores de $\text{succ}(x, k)$ onde k é uma potência de dois e no máximo u , onde u é o número máximo de passos que jamais daremos. Isso pode ser feito de forma eficiente, pois podemos usar a seguinte recursão:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Pré-calculer os valores leva um tempo $O(n \log u)$, porque $O(\log u)$ valores são calculados para cada nó. No grafo acima, os primeiros valores são os seguintes:

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
...									

Depois disso, qualquer valor de $\text{succ}(x, k)$ pode ser calculado representando o número de passos k como uma soma de potências de dois. Por exemplo, se quisermos calcular o valor de $\text{succ}(x, 11)$, primeiro formamos a representação $11 = 8 + 2 + 1$. Usando isso,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Por exemplo, no grafo anterior

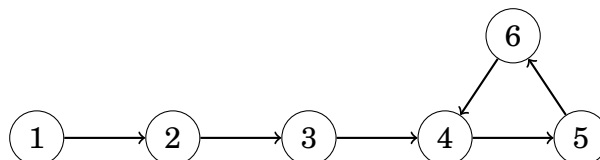
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Essa representação sempre consiste em $O(\log k)$ partes, então calcular um valor de $\text{succ}(x, k)$ leva um tempo $O(\log k)$.

16.4 Detecção de ciclo

Considere um grafo de sucessores que contém apenas um caminho que termina em um ciclo. Podemos fazer as seguintes perguntas: se começarmos nossa caminhada no nó inicial, qual é o primeiro nó no ciclo e quantos nós o ciclo contém?

Por exemplo, no grafo



começamos nossa caminhada no nó 1, o primeiro nó que pertence ao ciclo é o nó 4, e o ciclo consiste em três nós (4, 5 e 6).

Uma maneira simples de detectar o ciclo é caminhar no grafo e manter o controle de todos os nós que foram visitados. Uma vez que um nó é visitado pela

segunda vez, podemos concluir que o nó é o primeiro nó no ciclo. Este método funciona em tempo $O(n)$ e também usa memória $O(n)$.

No entanto, existem algoritmos melhores para detecção de ciclo. A complexidade de tempo de tais algoritmos ainda é $O(n)$, mas eles usam apenas memória $O(1)$. Esta é uma melhoria importante se n for grande. A seguir, discutiremos o algoritmo de Floyd que alcança essas propriedades.

Algoritmo de Floyd

O **algoritmo de Floyd**² caminha para frente no grafo usando dois ponteiros a e b . Ambos os ponteiros começam em um nó x que é o nó inicial do grafo. Então, a cada turno, o ponteiro a caminha um passo para frente e o ponteiro b caminha dois passos para frente. O processo continua até que os ponteiros se encontrem:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

Neste ponto, o ponteiro a caminhou k passos e o ponteiro b caminhou $2k$ passos, então o comprimento do ciclo divide k . Assim, o primeiro nó que pertence ao ciclo pode ser encontrado movendo o ponteiro a para o nó x e avançando os ponteiros passo a passo até que se encontrem novamente.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
primeiro = a;
```

Depois disso, o comprimento do ciclo pode ser calculado da seguinte forma:

```
b = succ(a);
comprimento = 1;
while (a != b) {
    b = succ(b);
    comprimento++;
}
```

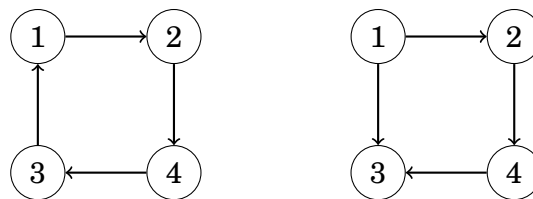
²A ideia do algoritmo é mencionada em [46] e atribuída a R. W. Floyd; no entanto, não se sabe se Floyd realmente descobriu o algoritmo.

Capítulo 17

Conectividade forte

Em um grafo direcionado, as arestas podem ser percorridas em apenas uma direção, então mesmo se o grafo for conexo, isso não garante que haverá um caminho de um nó para outro nó. Por essa razão, é significativo definir um novo conceito que requer mais do que conectividade.

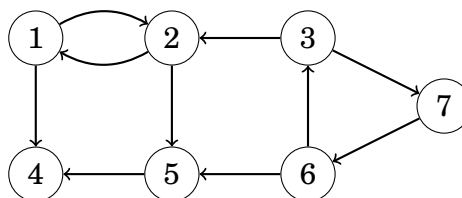
Um grafo é **fortemente conexo** se houver um caminho de qualquer nó para todos os outros nós no grafo. Por exemplo, na figura a seguir, o grafo à esquerda é fortemente conexo, enquanto o grafo à direita não é.



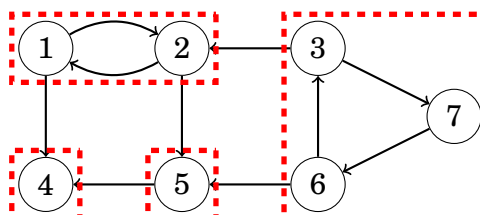
O grafo à direita não é fortemente conexo porque, por exemplo, não há caminho do nó 2 para o nó 1.

Os **componentes fortemente conexos** de um grafo dividem o grafo em partes fortemente conexas que são tão grandes quanto possível. Os componentes fortemente conexos formam um **grafo de componentes** acíclico que representa a estrutura profunda do grafo original.

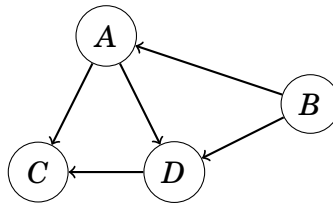
Por exemplo, para o grafo



os componentes fortemente conexos são os seguintes:



O grafo de componentes correspondente é o seguinte:



Os componentes são $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ e $D = \{5\}$.

Um grafo de componentes é um grafo direcionado acíclico, então é mais fácil de processar do que o grafo original. Como o grafo não contém ciclos, sempre podemos construir uma ordenação topológica e usar técnicas de programação dinâmica como aquelas apresentadas no Capítulo 16.

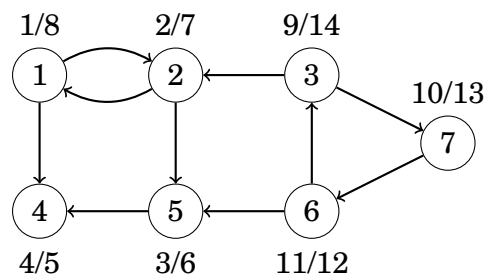
17.1 Algoritmo de Kosaraju

O **algoritmo de Kosaraju**¹ é um método eficiente para encontrar os componentes fortemente conexos de um grafo direcionado. O algoritmo realiza duas buscas em profundidade: a primeira busca constrói uma lista de nós de acordo com a estrutura do grafo, e a segunda busca forma os componentes fortemente conexos.

Busca 1

A primeira fase do algoritmo de Kosaraju constrói uma lista de nós na ordem em que uma busca em profundidade os processa. O algoritmo percorre os nós, e inicia uma busca em profundidade em cada nó não processado. Cada nó será adicionado à lista após ter sido processado.

No grafo de exemplo, os nós são processados na seguinte ordem:



A notação x/y significa que o processamento do nó começou no tempo x e terminou no tempo y . Assim, a lista correspondente é a seguinte:

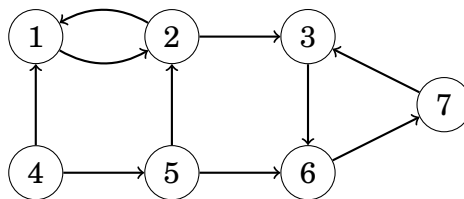
¹De acordo com [1], S. R. Kosaraju inventou este algoritmo em 1978, mas não o publicou. Em 1981, o mesmo algoritmo foi redescoberto e publicado por M. Sharir [57].

nó	tempo de processamento
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Busca 2

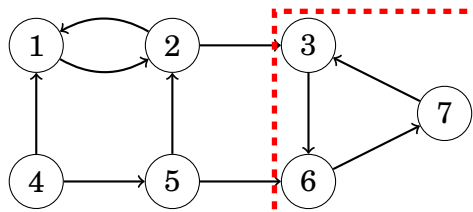
A segunda fase do algoritmo forma os componentes fortemente conexos do grafo. Primeiro, o algoritmo inverte cada aresta no grafo. Isso garante que durante a segunda busca, sempre encontraremos componentes fortemente conexos que não possuem nós extras.

Após inverter as arestas, o grafo de exemplo é o seguinte:



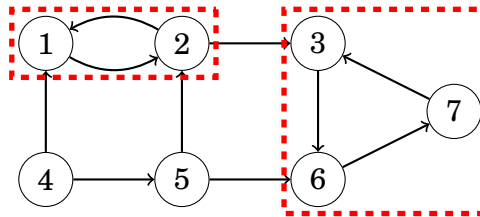
Depois disso, o algoritmo percorre a lista de nós criada pela primeira busca, em ordem *inversa*. Se um nó não pertence a um componente, o algoritmo cria um novo componente e inicia uma busca em profundidade que adiciona todos os novos nós encontrados durante a busca ao novo componente.

No grafo de exemplo, o primeiro componente começa no nó 3:

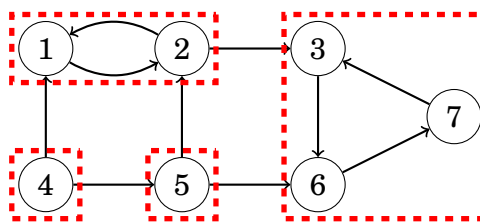


Observe que como todas as arestas são invertidas, o componente não "vaza" para outras partes no grafo.

Os próximos nós na lista são os nós 7 e 6, mas eles já pertencem a um componente, então o próximo novo componente começa no nó 1:



Finalmente, o algoritmo processa os nós 5 e 4 que criam os componentes fortemente conexos restantes:



A complexidade de tempo do algoritmo é $O(n + m)$, porque o algoritmo realiza duas buscas em profundidade.

17.2 Problema 2SAT

A conectividade forte também está relacionada ao **problema 2SAT**². Neste problema, recebemos uma fórmula lógica

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

onde cada a_i e b_i é uma variável lógica (x_1, x_2, \dots, x_n) ou uma negação de uma variável lógica ($\neg x_1, \neg x_2, \dots, \neg x_n$). Os símbolos " \wedge " e " \vee " denotam os operadores lógicos "e" e "ou". Nossa tarefa é atribuir a cada variável um valor para que a fórmula seja verdadeira, ou declarar que isso não é possível.

Por exemplo, a fórmula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

é verdadeira quando as variáveis são atribuídas da seguinte forma:

$$\begin{cases} x_1 = \text{falso} \\ x_2 = \text{falso} \\ x_3 = \text{verdadeiro} \\ x_4 = \text{verdadeiro} \end{cases}$$

²O algoritmo apresentado aqui foi introduzido em [4]. Há também outro algoritmo linear conhecido [19] que é baseado em backtracking.

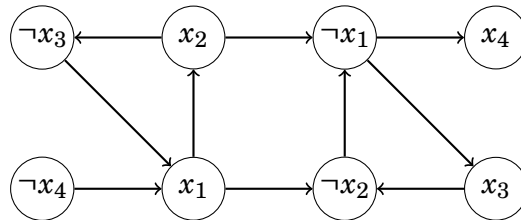
No entanto, a fórmula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

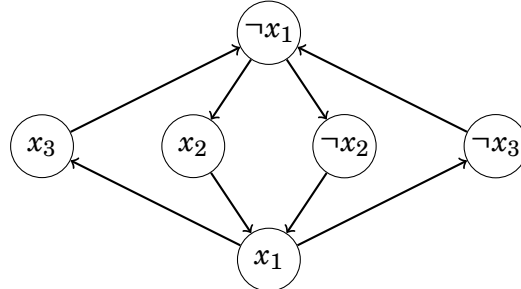
é sempre falsa, independentemente de como atribuímos os valores. A razão para isso é que não podemos escolher um valor para x_1 sem criar uma contradição. Se x_1 for falso, ambos x_2 e $\neg x_2$ devem ser verdadeiros, o que é impossível, e se x_1 for verdadeiro, ambos x_3 e $\neg x_3$ devem ser verdadeiros, o que também é impossível.

O problema 2SAT pode ser representado como um grafo cujos nós correspondem às variáveis x_i e negações $\neg x_i$, e as arestas determinam as conexões entre as variáveis. Cada par $(a_i \vee b_i)$ gera duas arestas: $\neg a_i \rightarrow b_i$ e $\neg b_i \rightarrow a_i$. Isso significa que se a_i não for verdadeiro, b_i deve ser verdadeiro, e vice-versa.

O grafo para a fórmula L_1 é:



E o grafo para a fórmula L_2 é:



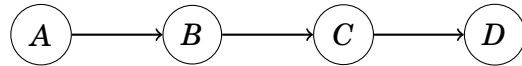
A estrutura do grafo nos diz se é possível atribuir os valores das variáveis para que a fórmula seja verdadeira. Acontece que isso pode ser feito exatamente quando não há nós x_i e $\neg x_i$ tais que ambos os nós pertençam ao mesmo componente fortemente conexo. Se houver tais nós, o grafo contém um caminho de x_i para $\neg x_i$ e também um caminho de $\neg x_i$ para x_i , então ambos x_i e $\neg x_i$ devem ser verdadeiros, o que não é possível.

No grafo da fórmula L_1 , não há nós x_i e $\neg x_i$ tais que ambos os nós pertençam ao mesmo componente fortemente conexo, então uma solução existe. No grafo da fórmula L_2 , todos os nós pertencem ao mesmo componente fortemente conexo, então uma solução não existe.

Se uma solução existir, os valores para as variáveis podem ser encontrados percorrendo os nós do grafo de componentes em uma ordem de ordenação topológica inversa. A cada passo, processamos um componente que não contém arestas que levam a um componente não processado. Se as variáveis no componente não tiverem sido atribuídas a valores, seus valores serão determinados de acordo

com os valores no componente, e se eles já tiverem valores, eles permanecem inalterados. O processo continua até que cada variável tenha sido atribuída a um valor.

O grafo de componentes para a fórmula L_1 é o seguinte:



Os componentes são $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ e $D = \{x_4\}$. Ao construir a solução, primeiro processamos o componente D onde x_4 se torna verdadeiro. Depois disso, processamos o componente C onde x_1 e x_2 se tornam falsos e x_3 se torna verdadeiro. Todas as variáveis foram atribuídas a valores, então os componentes restantes A e B não alteram as variáveis.

Observe que este método funciona porque o grafo tem uma estrutura especial: se houver caminhos do nó x_i para o nó x_j e do nó x_j para o nó $\neg x_j$, então o nó x_i nunca se torna verdadeiro. A razão para isso é que também há um caminho do nó $\neg x_j$ para o nó $\neg x_i$, e ambos x_i e x_j se tornam falsos.

Um problema mais difícil é o **problema 3SAT**, onde cada parte da fórmula é da forma $(a_i \vee b_i \vee c_i)$. Este problema é NP-difícil, então nenhum algoritmo eficiente para resolver o problema é conhecido.

Capítulo 18

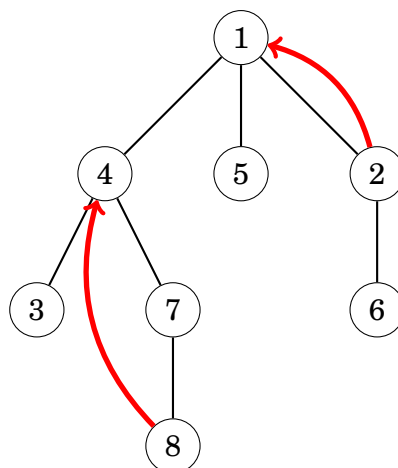
Consultas em Árvores

Este capítulo aborda técnicas para processar consultas em subárvores e caminhos de uma árvore enraizada. Por exemplo, algumas dessas consultas são:

- Qual é o k -ésimo ancestral de um nó?
- Qual é a soma dos valores na subárvore de um nó?
- Qual é a soma dos valores em um caminho entre dois nós?
- Qual é o ancestral comum mais baixo de dois nós?

18.1 Encontrando Ancestrais

O k -ésimo **ancestral** de um nó x em uma árvore enraizada é o nó que alcançaremos se subirmos k níveis a partir de x . Seja $\text{ancestral}(x, k)$ o k -ésimo ancestral de um nó x (ou 0 se não houver tal ancestral). Por exemplo, na árvore a seguir, $\text{ancestral}(2, 1) = 1$ e $\text{ancestral}(8, 2) = 4$.



Uma maneira fácil de calcular qualquer valor de $\text{ancestral}(x, k)$ é realizar uma sequência de k movimentos na árvore. No entanto, a complexidade de tempo deste método é $O(k)$, o que pode ser lento, pois uma árvore de n nós pode ter uma cadeia de n nós.

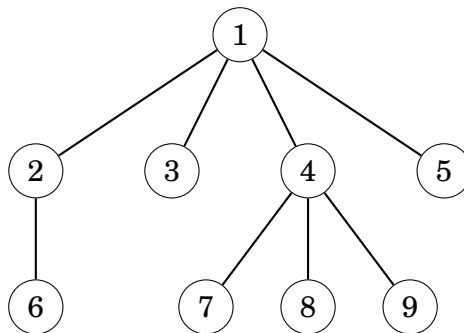
Felizmente, usando uma técnica semelhante àquela utilizada no Capítulo 16.3, qualquer valor de $\text{ancestral}(x, k)$ pode ser calculado eficientemente em tempo $O(\log k)$ após o pré-processamento. A ideia é pré-calcular todos os valores $\text{ancestral}(x, k)$ onde $k \leq n$ é uma potência de dois. Por exemplo, os valores para a árvore acima são os seguintes:

x	1	2	3	4	5	6	7	8
$\text{ancestral}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestral}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestral}(x, 4)$	0	0	0	0	0	0	0	0
...								

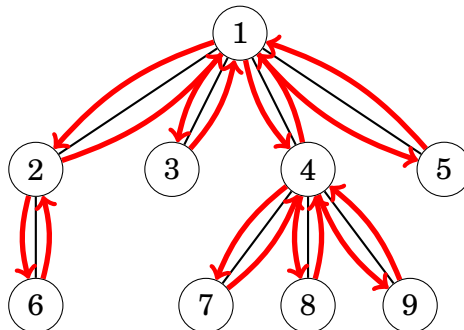
O pré-processamento leva tempo $O(n \log n)$, pois $O(\log n)$ valores são calculados para cada nó. Após isso, qualquer valor de $\text{ancestral}(x, k)$ pode ser calculado em tempo $O(\log k)$ representando k como uma soma onde cada termo é uma potência de dois.

18.2 Subárvores e Caminhos

Um **vetor de percurso de árvore** contém os nós de uma árvore enraizada na ordem em que uma busca em profundidade a partir do nó raiz os visita. Por exemplo, na árvore:



Uma busca em profundidade procede da seguinte forma:



Portanto, o vetor de percurso de árvore correspondente é o seguinte:

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

Consultas em Subárvores

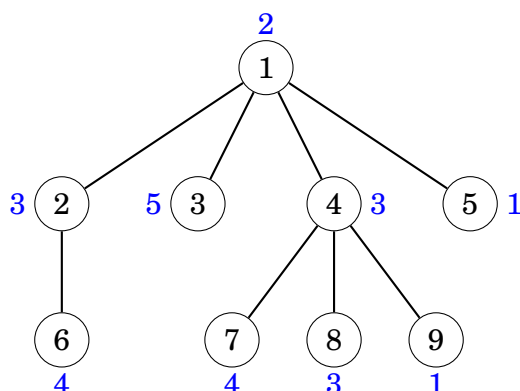
Cada subárvore de uma árvore corresponde a um subvetor do vetor de percurso de árvore, de forma que o primeiro elemento do subvetor é o nó raiz. Por exemplo, o seguinte subvetor contém os nós da subárvore do nó 4:

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

Usando esse fato, podemos processar eficientemente consultas relacionadas a subárvores de uma árvore. Como exemplo, considere um problema onde cada nó recebe um valor e nossa tarefa é oferecer suporte às seguintes consultas:

- atualizar o valor de um nó
- calcular a soma dos valores na subárvore de um nó

Considere a seguinte árvore onde os números azuis são os valores dos nós. Por exemplo, a soma da subárvore do nó 4 é $3 + 4 + 3 + 1 = 11$.



A ideia é construir um vetor de percurso de árvore que contém três valores para cada nó: o identificador do nó, o tamanho da subárvore e o valor do nó. Por exemplo, o vetor para a árvore acima é o seguinte:

id do nó	1	2	6	3	4	7	8	9	5
tamanho da subárvore	9	2	1	1	4	1	1	1	1
valor do nó	2	3	4	5	3	4	3	1	1

Usando este vetor, podemos calcular a soma dos valores em qualquer subárvore primeiro descobrindo o tamanho da subárvore e depois os valores dos nós correspondentes. Por exemplo, os valores na subárvore do nó 4 podem ser encontrados da seguinte forma:

id do nó	1	2	6	3	4	7	8	9	5
tamanho da subárvore	9	2	1	1	4	1	1	1	1
valor do nó	2	3	4	5	3	4	3	1	1

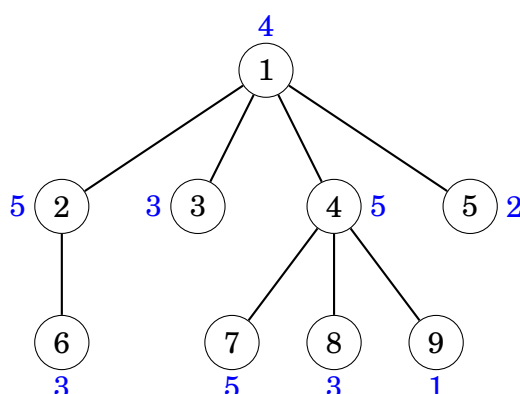
Para responder às consultas de forma eficiente, basta armazenar os valores dos nós em uma árvore binária indexada ou árvore de segmentos. Depois disso, podemos atualizar um valor e calcular a soma dos valores em tempo $O(\log n)$.

Consultas em Caminhos

Usando um vetor de percurso de árvore, também podemos calcular eficientemente somas de valores em caminhos do nó raiz para qualquer nó da árvore. Considere um problema em que nossa tarefa é oferecer suporte às seguintes consultas:

- alterar o valor de um nó
- calcular a soma dos valores em um caminho da raiz até um nó

Por exemplo, na árvore a seguir, a soma dos valores do nó raiz ao nó 7 é $4 + 5 + 5 = 14$:



Podemos resolver este problema como antes, mas agora cada valor na última linha do vetor é a soma dos valores em um caminho da raiz até o nó. Por exemplo, o seguinte vetor corresponde à árvore acima:

id do nó	1	2	6	3	4	7	8	9	5
tamanho da subárvore	9	2	1	1	4	1	1	1	1
soma do caminho	4	9	12	7	9	14	12	10	6

Quando o valor de um nó aumenta em x , as somas de todos os nós em sua subárvore aumentam em x . Por exemplo, se o valor do nó 4 aumentar em 1, o vetor muda da seguinte forma:

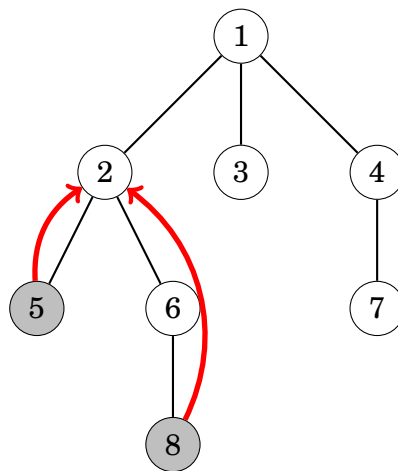
id do nó	1	2	6	3	4	7	8	9	5
tamanho da subárvore	9	2	1	1	4	1	1	1	1
soma do caminho	4	9	12	7	10	15	13	11	6

Assim, para oferecer suporte a ambas as operações, devemos ser capazes de aumentar todos os valores em um intervalo e recuperar um único valor. Isso pode ser feito em tempo $O(\log n)$ usando uma árvore de índice binário ou árvore de segmentos (consulte o Capítulo 9.4).

18.3 Ancestral Comum Mais Baixo

O **ancestral comum mais baixo** de dois nós de uma árvore enraizada é o nó mais baixo cuja subárvore contém ambos os nós. Um problema típico é processar eficientemente consultas que pedem para encontrar o ancestral comum mais baixo de dois nós.

Por exemplo, na árvore a seguir, o ancestral comum mais baixo dos nós 5 e 8 é o nó 2:



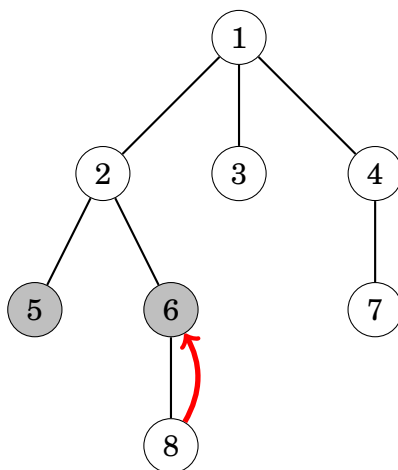
A seguir, discutiremos duas técnicas eficientes para encontrar o ancestral comum mais baixo de dois nós.

Método 1

Uma maneira de resolver o problema é usar o fato de que podemos encontrar eficientemente o k -ésimo ancestral de qualquer nó na árvore. Usando isso, podemos dividir o problema de encontrar o ancestral comum mais baixo em duas partes.

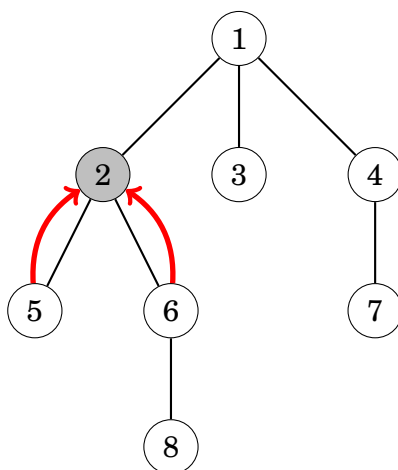
Usamos dois ponteiros que apontam inicialmente para os dois nós cujo ancestral comum mais baixo devemos encontrar. Primeiro, movemos um dos ponteiros para cima para que ambos os ponteiros apontem para nós no mesmo nível.

No cenário de exemplo, movemos o segundo ponteiro um nível para cima para que ele aponte para o nó 6, que está no mesmo nível do nó 5:



Depois disso, determinamos o número mínimo de etapas necessárias para mover ambos os ponteiros para cima para que apontem para o mesmo nó. O nó para o qual os ponteiros apontam depois disso é o ancestral comum mais baixo.

No cenário de exemplo, basta mover ambos os ponteiros um passo para cima até o nó 2, que é o ancestral comum mais baixo:

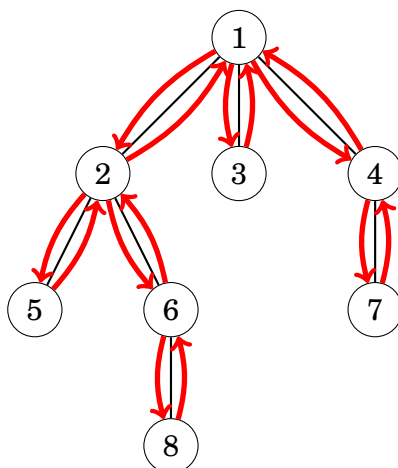


Como ambas as partes do algoritmo podem ser executadas em tempo $O(\log n)$ usando informações pré-calculadas, podemos encontrar o ancestral comum mais baixo de quaisquer dois nós em tempo $O(\log n)$.

Método 2

Outra maneira de resolver o problema é baseada em um vetor de percurso de árvore¹. Novamente, a ideia é percorrer os nós usando uma busca em profundidade:

¹Este algoritmo de ancestral comum mais baixo foi apresentado em [7]. Essa técnica às vezes é chamada de **Euler tour technique** [66].



No entanto, usamos um vetor de percurso de árvore diferente do que antes: adicionamos cada nó ao vetor *sempre* que a busca em profundidade passa pelo nó, e não apenas na primeira visita. Portanto, um nó que tem k filhos aparece $k + 1$ vezes no vetor e há um total de $2n - 1$ nós no vetor.

Armazenamos dois valores no vetor: o identificador do nó e a profundidade do nó na árvore. O seguinte vetor corresponde à árvore acima:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id do nó	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
profundidade	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Agora podemos encontrar o ancestral comum mais baixo dos nós a e b encontrando o nó com a profundidade *mínima* entre os nós a e b no vetor. Por exemplo, o ancestral comum mais baixo dos nós 5 e 8 pode ser encontrado da seguinte forma:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id do nó	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
profundidade	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

O nó 5 está na posição 2, o nó 8 está na posição 5 e o nó com profundidade mínima entre as posições 2...5 é o nó 2 na posição 3, cuja profundidade é 2. Assim, o ancestral comum mais baixo dos nós 5 e 8 é o nó 2.

Portanto, para encontrar o ancestral comum mais baixo de dois nós, basta processar uma consulta de mínimo de intervalo. Como o vetor é estático, podemos processar tais consultas em tempo $O(1)$ após um pré-processamento de tempo $O(n \log n)$.

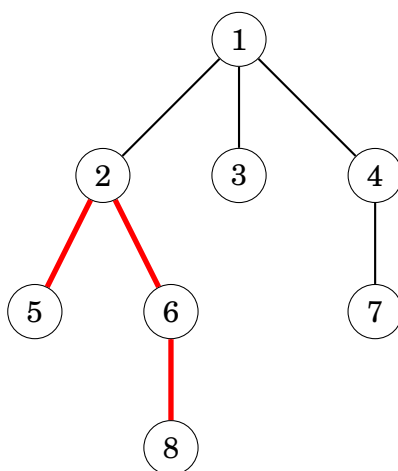
Distâncias entre Nós

A distância entre os nós a e b é igual ao comprimento do caminho de a para b . Acontece que o problema de calcular a distância entre os nós se reduz a encontrar seu ancestral comum mais baixo.

Primeiro, enraizamos a árvore arbitrariamente. Depois disso, a distância dos nós a e b pode ser calculada usando a fórmula

$$\text{profundidade}(a) + \text{profundidade}(b) - 2 \cdot \text{profundidade}(c),$$

onde c é o ancestral comum mais baixo de a e b e $\text{profundidade}(s)$ denota a profundidade do nó s . Por exemplo, considere a distância dos nós 5 e 8:



O ancestral comum mais baixo dos nós 5 e 8 é o nó 2. As profundidades dos nós são $\text{profundidade}(5) = 3$, $\text{profundidade}(8) = 4$ e $\text{profundidade}(2) = 2$, então a distância entre os nós 5 e 8 é $3 + 4 - 2 \cdot 2 = 3$.

18.4 Algoritmos Offline

Até agora, discutimos algoritmos *online* para consultas em árvores. Esses algoritmos são capazes de processar consultas uma após a outra, de forma que cada consulta seja respondida antes de receber a próxima consulta.

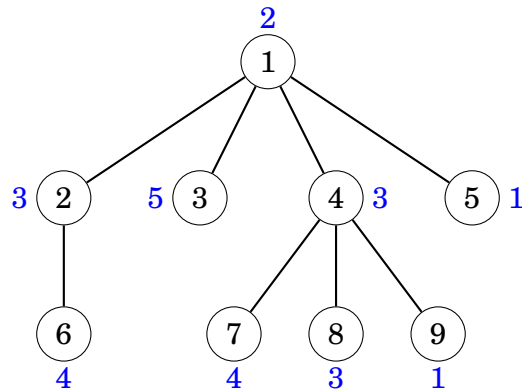
No entanto, em muitos problemas, a propriedade *online* não é necessária. Nesta seção, vamos nos concentrar em algoritmos *offline*. Esses algoritmos recebem um conjunto de consultas que podem ser respondidas em qualquer ordem. Muitas vezes, é mais fácil projetar um algoritmo *offline* em comparação com um algoritmo *online*.

Mesclando Estruturas de Dados

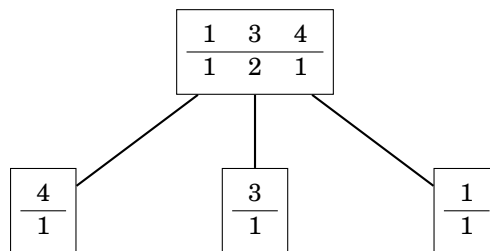
Um método para construir um algoritmo *offline* é realizar um percurso de árvore em profundidade e manter estruturas de dados nos nós. Em cada nó s , criamos uma estrutura de dados $d[s]$ que é baseada nas estruturas de dados dos filhos de

s . Então, usando esta estrutura de dados, todas as consultas relacionadas a s são processadas.

Como exemplo, considere o seguinte problema: Recebemos uma árvore onde cada nó possui algum valor. Nossa tarefa é processar consultas da forma "calcular o número de nós com valor x na subárvore do nó s ". Por exemplo, na árvore a seguir, a subárvore do nó 4 contém dois nós cujo valor é 3.



Neste problema, podemos usar mapas para responder às consultas. Por exemplo, os mapas para o nó 4 e seus filhos são os seguintes:



Se criarmos tal estrutura de dados para cada nó, podemos processar facilmente todas as consultas fornecidas, pois podemos lidar com todas as consultas relacionadas a um nó imediatamente após criar sua estrutura de dados. Por exemplo, a estrutura de mapa acima para o nó 4 nos diz que sua subárvore contém dois nós cujo valor é 3.

No entanto, seria muito lento criar todas as estruturas de dados do zero. Em vez disso, em cada nó s , criamos uma estrutura de dados inicial $d[s]$ que contém apenas o valor de s . Depois disso, percorremos os filhos de s e *mesclamos* $d[s]$ e todas as estruturas de dados $d[u]$ onde u é um filho de s .

Por exemplo, na árvore acima, o mapa para o nó 4 é criado mesclando os seguintes mapas:



Aqui, o primeiro mapa é a estrutura de dados inicial para o nó 4, e os outros três mapas correspondem aos nós 7, 8 e 9.

A mesclagem no nó s pode ser feita da seguinte forma: Percorremos os filhos de s e, em cada filho u , mesclamos $d[s]$ e $d[u]$. Sempre copiamos o conteúdo de $d[u]$ para $d[s]$. No entanto, antes disso, *trocamos* o conteúdo de $d[s]$ e $d[u]$ se $d[s]$ for menor que $d[u]$. Ao fazer isso, cada valor é copiado apenas $O(\log n)$ vezes durante o percurso da árvore, o que garante a eficiência do algoritmo.

Para trocar o conteúdo de duas estruturas de dados a e b de forma eficiente, podemos usar o seguinte código:

```
swap(a, b);
```

É garantido que o código acima funcione em tempo constante quando a e b são estruturas de dados da biblioteca padrão C++.

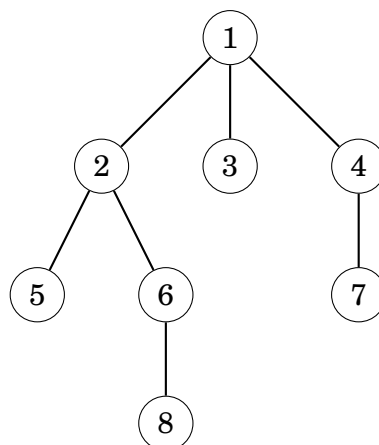
Ancestrais Comuns Mais Baixos

Há também um algoritmo offline para processar um conjunto de consultas de ancestral comum mais baixo². O algoritmo é baseado na estrutura de dados union-find (consulte o Capítulo 15.2), e o benefício do algoritmo é que ele é mais fácil de implementar do que os algoritmos discutidos anteriormente neste capítulo.

O algoritmo recebe como entrada um conjunto de pares de nós e determina para cada par o ancestral comum mais baixo dos nós. O algoritmo realiza um percurso de árvore em profundidade e mantém conjuntos disjuntos de nós. Inicialmente, cada nó pertence a um conjunto separado. Para cada conjunto, também armazenamos o nó mais alto na árvore que pertence ao conjunto.

Quando o algoritmo visita um nó x , ele percorre todos os nós y de forma que o ancestral comum mais baixo de x e y precise ser encontrado. Se y já tiver sido visitado, o algoritmo relata que o ancestral comum mais baixo de x e y é o nó mais alto no conjunto de y . Então, após processar o nó x , o algoritmo une os conjuntos de x e seu pai.

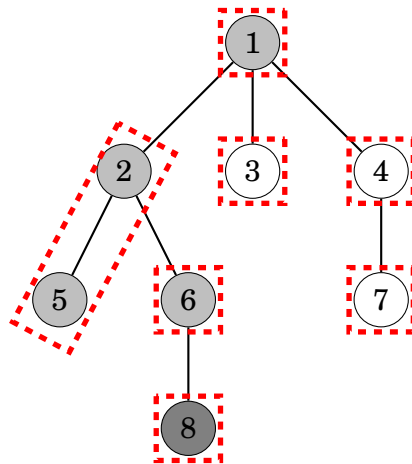
Por exemplo, suponha que queremos encontrar os ancestrais comuns mais baixos dos pares de nós (5,8) e (2,7) na árvore a seguir:



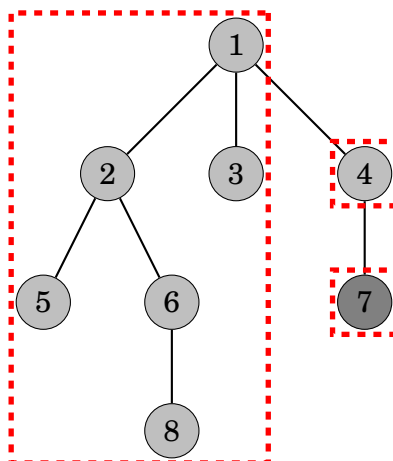
Nas árvores a seguir, os nós cinzas denotam nós visitados e grupos de nós tracejados pertencem ao mesmo conjunto. Quando o algoritmo visita o nó 8, ele

²Este algoritmo foi publicado por R. E. Tarjan em 1979 [65].

percebe que o nó 5 foi visitado e o nó mais alto em seu conjunto é 2. Assim, o ancestral comum mais baixo dos nós 5 e 8 é 2:



Mais tarde, ao visitar o nó 7, o algoritmo determina que o ancestral comum mais baixo dos nós 2 e 7 é 1:



Capítulo 19

Caminhos e circuitos

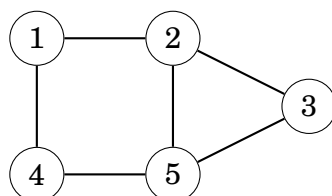
Este capítulo se concentra em dois tipos de caminhos em grafos:

- Um **caminho Euleriano** é um caminho que passa por cada aresta exatamente uma vez.
- Um **caminho Hamiltoniano** é um caminho que visita cada nó exatamente uma vez.

Embora os caminhos Eulerianos e Hamiltonianos pareçam conceitos semelhantes à primeira vista, os problemas computacionais relacionados a eles são muito diferentes. Acontece que existe uma regra simples que determina se um grafo contém um caminho Euleriano, e também existe um algoritmo eficiente para encontrar tal caminho se ele existir. Ao contrário, verificar a existência de um caminho Hamiltoniano é um problema NP-difícil, e nenhum algoritmo eficiente é conhecido por resolver o problema.

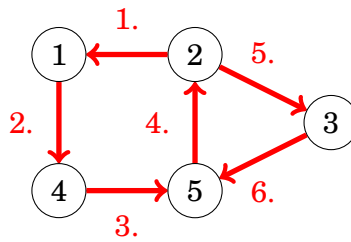
19.1 Caminhos Eulerianos

Um **caminho Euleriano**¹ é um caminho que passa exatamente uma vez por cada aresta do grafo. Por exemplo, o grafo

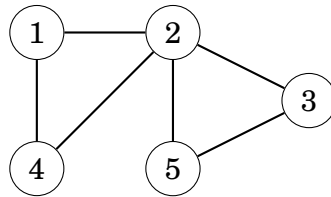


tem um caminho Euleriano do nó 2 ao nó 5:

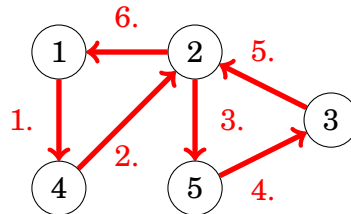
¹L. Euler estudou tais caminhos em 1736 quando ele resolveu o famoso problema da ponte de Königsberg. Este foi o nascimento da teoria dos grafos.



Um **circuito Euleriano** é um caminho Euleriano que começa e termina no mesmo nó. Por exemplo, o grafo



possui um circuito Euleriano que começa e termina no nó 1:



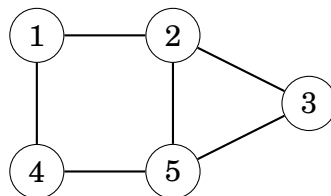
Existência

A existência de caminhos e circuitos Eulerianos depende dos graus dos nós. Primeiro, um grafo não direcionado tem um caminho Euleriano exatamente quando todas as arestas pertencem ao mesmo componente conexo e

- o grau de cada nó é par *ou*
- o grau de exatamente dois nós é ímpar, e o grau de todos os outros nós é par.

No primeiro caso, cada caminho Euleriano também é um circuito Euleriano. No segundo caso, os nós de grau ímpar são os nós inicial e final de um caminho Euleriano que não é um circuito Euleriano.

Por exemplo, no grafo



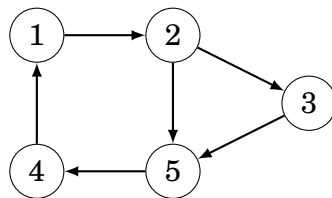
os nós 1, 3 e 4 têm grau 2, e os nós 2 e 5 têm grau 3. Exatamente dois nós têm grau ímpar, então há um caminho Euleriano entre os nós 2 e 5, mas o grafo não contém um circuito Euleriano.

Em um grafo direcionado, focamos nos graus de entrada e saída dos nós. Um grafo direcionado contém um caminho Euleriano exatamente quando todas as arestas pertencem ao mesmo componente conexo e

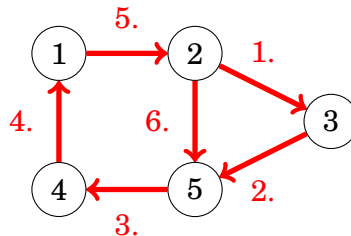
- em cada nó, o grau de entrada é igual ao grau de saída, *ou*
- em um nó, o grau de entrada é um a mais que o grau de saída, em outro nó, o grau de saída é um a mais que o grau de entrada, e em todos os outros nós, o grau de entrada é igual ao grau de saída.

No primeiro caso, cada caminho Euleriano também é um circuito Euleriano, e no segundo caso, o grafo contém um caminho Euleriano que começa no nó cujo grau de saída é maior e termina no nó cujo grau de entrada é maior.

Por exemplo, no grafo



os nós 1, 3 e 4 têm grau de entrada 1 e grau de saída 1, o nó 2 tem grau de entrada 1 e grau de saída 2, e o nó 5 tem grau de entrada 2 e grau de saída 1. Portanto, o grafo contém um caminho Euleriano do nó 2 ao nó 5:



Algoritmo de Hierholzer

O **algoritmo de Hierholzer**² é um método eficiente para construir um circuito Euleriano. O algoritmo consiste em várias rodadas, cada uma das quais adiciona novas arestas ao circuito. Claro, assumimos que o grafo contém um circuito Euleriano; caso contrário, o algoritmo de Hierholzer não pode encontrá-lo.

Primeiro, o algoritmo constrói um circuito que contém algumas (não necessariamente todas) das arestas do grafo. Depois disso, o algoritmo estende o circuito passo a passo adicionando subcircuitos a ele. O processo continua até que todas as arestas tenham sido adicionadas ao circuito.

O algoritmo estende o circuito sempre encontrando um nó x que pertence ao circuito, mas tem uma aresta de saída que não está incluída no circuito. O algoritmo constrói um novo caminho a partir do nó x que contém apenas arestas

²O algoritmo foi publicado em 1873 após a morte de Hierholzer [35].

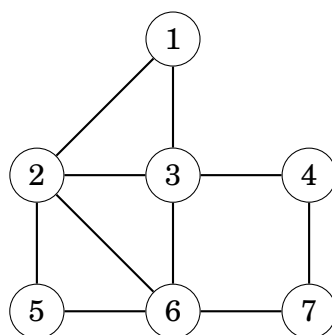
que ainda não estão no circuito. Cedo ou tarde, o caminho retornará ao nó x , o que cria um subcircuito.

Se o grafo contiver apenas um caminho Euleriano, ainda podemos usar o algoritmo de Hierholzer para encontrá-lo adicionando uma aresta extra ao grafo e removendo a aresta após o circuito ter sido construído. Por exemplo, em um grafo não direcionado, adicionamos a aresta extra entre os dois nós de grau ímpar.

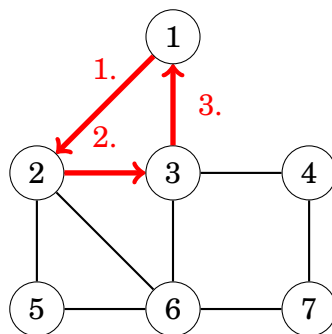
A seguir, veremos como o algoritmo de Hierholzer constrói um circuito Euleriano para um grafo não direcionado.

Exemplo

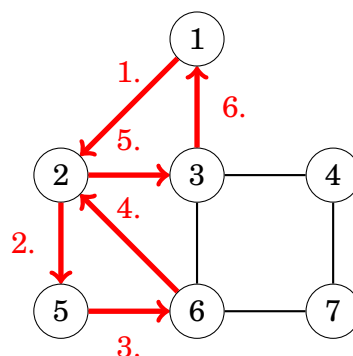
Vamos considerar o seguinte grafo:



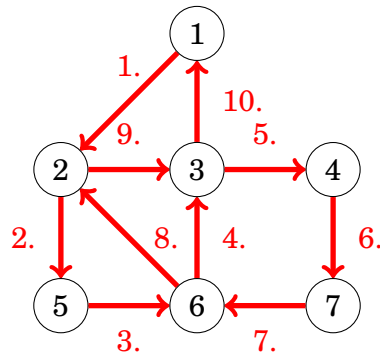
Suponha que o algoritmo primeiro cria um circuito que começa no nó 1. Um circuito possível é $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



Depois disso, o algoritmo adiciona o subcircuito $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ ao circuito:



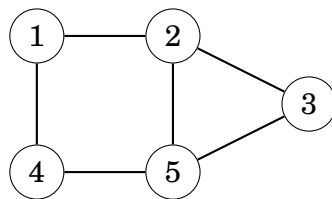
Finalmente, o algoritmo adiciona o subcircuito $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ ao circuito:



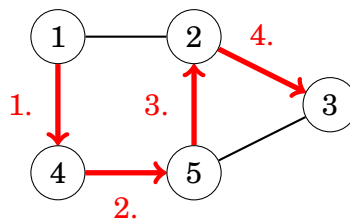
Agora todas as arestas estão incluídas no circuito, então construímos com sucesso um circuito Euleriano.

19.2 Caminhos Hamiltonianos

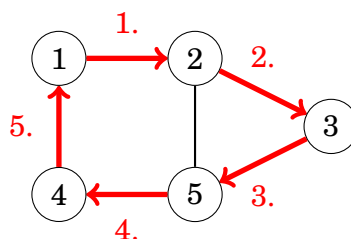
Um **caminho Hamiltoniano** é um caminho que visita cada nó do grafo exatamente uma vez. Por exemplo, o grafo



contém um caminho Hamiltoniano do nó 1 ao nó 3:



Se um caminho Hamiltoniano começa e termina no mesmo nó, ele é chamado de **circuito Hamiltoniano**. O grafo acima também possui um circuito Hamiltoniano que começa e termina no nó 1:



Existência

Nenhum método eficiente é conhecido para testar se um grafo contém um caminho Hamiltoniano, e o problema é NP-difícil. Ainda assim, em alguns casos especiais, podemos ter certeza de que um grafo contém um caminho Hamiltoniano.

Uma observação simples é que se o grafo for completo, ou seja, há uma aresta entre todos os pares de nós, ele também contém um caminho Hamiltoniano. Resultados ainda mais fortes foram alcançados:

- **Teorema de Dirac:** Se o grau de cada nó for pelo menos $n/2$, o grafo contém um caminho Hamiltoniano.
- **Teorema de Ore:** Se a soma dos graus de cada par de nós não adjacentes for pelo menos n , o grafo contém um caminho Hamiltoniano.

Uma propriedade comum nesses teoremas e outros resultados é que eles garantem a existência de um caminho Hamiltoniano se o grafo tiver *um grande número* de arestas. Isso faz sentido, porque quanto mais arestas o grafo contiver, mais possibilidades existem para construir um caminho Hamiltoniano.

Construção

Como não há como verificar eficientemente se um Hamiltoniano caminho existe, é claro que também não há método para construir o caminho de forma eficiente, porque caso contrário poderíamos apenas tentar construir o caminho e ver se ele existe.

Uma maneira simples de procurar um caminho Hamiltoniano é usar um algoritmo de backtracking que percorre todas as maneiras possíveis de construir o caminho. A complexidade de tempo de tal algoritmo é pelo menos $O(n!)$, porque existem $n!$ maneiras diferentes de escolher a ordem de n nós.

Uma solução mais eficiente é baseada em programação dinâmica (ver Capítulo 10.5). A ideia é calcular valores de uma função possível(S, x), onde S é um subconjunto de nós e x é um dos nós. A função indica se há um caminho Hamiltoniano que visita os nós de S e termina no nó x . É possível implementar esta solução em tempo $O(2^n n^2)$.

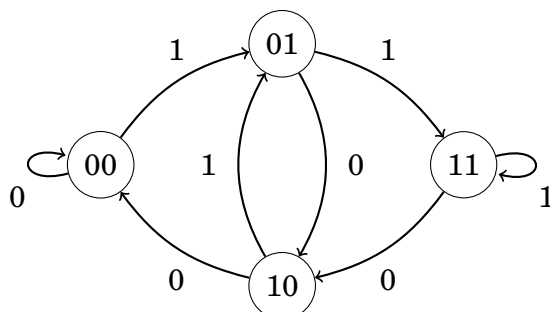
19.3 Sequências de De Bruijn

Uma **sequência de De Bruijn** é uma string que contém cada string de comprimento n exatamente uma vez como uma substring, para um alfabeto fixo de k caracteres. O comprimento de tal string é $k^n + n - 1$ caracteres. Por exemplo, quando $n = 3$ e $k = 2$, um exemplo de uma sequência de De Bruijn é

0001011100.

As substrings desta string são todas combinações de três bits: 000, 001, 010, 011, 100, 101, 110 e 111.

Acontece que cada sequência de De Bruijn corresponde a um caminho Euleriano em um grafo. A ideia é construir um grafo onde cada nó contém uma string de $n - 1$ caracteres e cada aresta adiciona um caractere à string. O seguinte grafo corresponde ao cenário acima:



Um caminho Euleriano neste grafo corresponde a uma string que contém todas as strings de comprimento n . A string contém os caracteres do nó inicial e todos os caracteres das arestas. O nó inicial tem $n - 1$ caracteres e há k^n caracteres nas arestas, então o comprimento da string é $k^n + n - 1$.

19.4 Passeio do Cavalo

Um **passeio do cavalo** é uma sequência de movimentos de um cavalo em um tabuleiro de xadrez $n \times n$ seguindo as regras do xadrez, de forma que o cavalo visite cada casa exatamente uma vez. Um passeio do cavalo é chamado de *passeio fechado* se o cavalo finalmente retorna à casa inicial e caso contrário, é chamado de *passeio aberto*.

Por exemplo, aqui está um passeio do cavalo aberto em um tabuleiro 5×5 :

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Um passeio do cavalo corresponde a um caminho Hamiltoniano em um grafo cujos nós representam as casas do tabuleiro, e dois nós são conectados com uma aresta se um cavalo pode se mover entre as casas de acordo com as regras do xadrez.

Uma maneira natural de construir um passeio do cavalo é usar o algoritmo de backtracking. A busca pode ser mais eficiente usando *heurísticas* que tentam guiar o cavalo para que um passeio completo seja encontrado rapidamente.

Regra de Warnsdorf

A **regra de Warnsdorf** é uma heurística simples e eficaz para encontrar um passeio do cavalo³. Usando a regra, é possível construir um passeio de forma eficiente mesmo em um tabuleiro grande. A ideia é mover o cavalo sempre para que ele termine em uma casa onde o número de movimentos possíveis seja o mais *pequeno* possível.

Por exemplo, na seguinte situação, existem cinco casas possíveis para as quais o cavalo pode se mover (casas $a \dots e$):

1				a
		2		
b				e
	c		d	

Nesta situação, a regra de Warnsdorf move o cavalo para a casa a , porque após esta escolha, há apenas um único movimento possível. As outras escolhas moveriam o cavalo para casas onde haveria três movimentos disponíveis.

³Essa heurística foi proposta no livro de Warnsdorf [69] em 1823. Existem também algoritmos polinomiais para encontrar passeios do cavalo [52], mas eles são mais complicados.

Capítulo 20

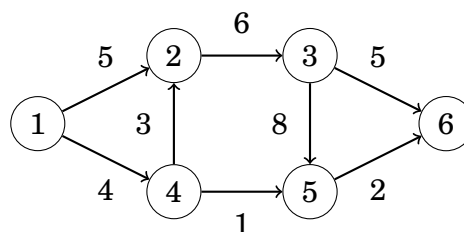
Fluxos e cortes

Neste capítulo, vamos nos concentrar nos seguintes dois problemas:

- **Encontrar um fluxo máximo:** Qual é a quantidade máxima de fluxo que podemos enviar de um nó para outro nó?
- **Encontrar um corte mínimo:** Qual é o conjunto de arestas de peso mínimo que separa dois nós do grafo?

A entrada para ambos os problemas é um grafo direcionado e ponderado que contém dois nós especiais: a *fonte* é um nó sem arestas de entrada, e o *sumidouro* é um nó sem arestas de saída.

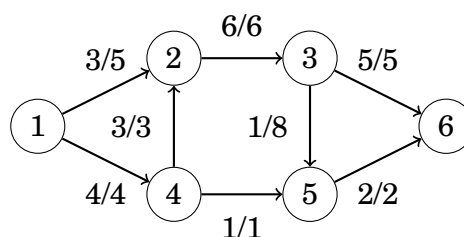
Como exemplo, usaremos o seguinte grafo onde o nó 1 é a fonte e o nó 6 é o sumidouro:



Fluxo máximo

No problema do **fluxo máximo**, nossa tarefa é enviar o máximo de fluxo possível da fonte para o sumidouro. O peso de cada aresta é uma capacidade que restringe o fluxo que pode passar pela aresta. Em cada nó intermediário, o fluxo de entrada e o de saída devem ser iguais.

Por exemplo, o tamanho máximo de um fluxo no grafo de exemplo é 7. A figura a seguir mostra como podemos rotear o fluxo:

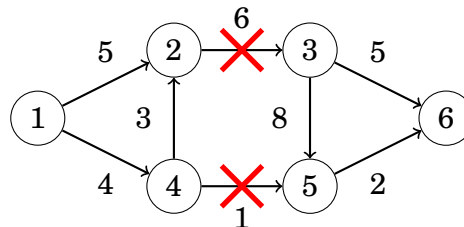


A notação v/k significa que um fluxo de v unidades está sendo roteado através de uma aresta cuja capacidade é de k unidades. O tamanho do fluxo é 7, pois a fonte envia $3 + 4$ unidades de fluxo e o sumidouro recebe $5 + 2$ unidades de fluxo. É fácil ver que este fluxo é máximo, pois a capacidade total das arestas que levam ao sumidouro é 7.

Corte mínimo

No problema do **corte mínimo**, nossa tarefa é remover um conjunto de arestas do grafo de forma que não haja caminho da fonte ao destino após a remoção e o peso total das arestas removidas seja mínimo.

O tamanho mínimo de um corte no grafo de exemplo é 7. Basta remover as arestas $2 \rightarrow 3$ e $4 \rightarrow 5$:



Após a remoção das arestas, não haverá caminho da fonte para o sumidouro. O tamanho do corte é 7, pois os pesos das arestas removidas são 6 e 1. O corte é mínimo, pois não há maneira válida de remover arestas do grafo de forma que seu peso total seja menor que 7.

Não é uma coincidência que o tamanho máximo de um fluxo e o tamanho mínimo de um corte sejam iguais no exemplo acima. Acontece que um fluxo máximo e um corte mínimo são *sempre* igualmente grandes, então os conceitos são dois lados da mesma moeda.

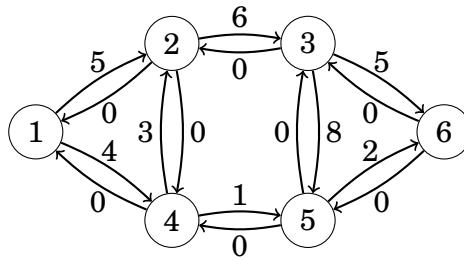
Em seguida, discutiremos o algoritmo de Ford–Fulkerson que pode ser usado para encontrar o fluxo máximo e o corte mínimo de um grafo. O algoritmo também nos ajuda a entender *por que* eles são igualmente grandes.

20.1 Algoritmo de Ford–Fulkerson

O **algoritmo de Ford–Fulkerson** [25] encontra o fluxo máximo em um grafo. O algoritmo começa com um fluxo vazio, e a cada passo encontra um caminho da fonte para o sumidouro que gera mais fluxo. Finalmente, quando o algoritmo não consegue mais aumentar o fluxo, o fluxo máximo foi encontrado.

O algoritmo utiliza uma representação especial do grafo em que cada aresta original possui uma aresta reversa em outra direção. O peso de cada aresta indica quanto mais fluxo poderia ser direcionado por ela. No início do algoritmo, o peso de cada aresta original é igual à capacidade da aresta e o peso de cada aresta reversa é zero.

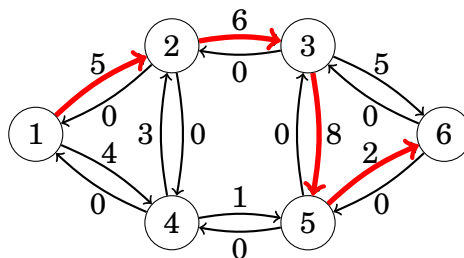
A nova representação para o grafo de exemplo é a seguinte:



Descrição do algoritmo

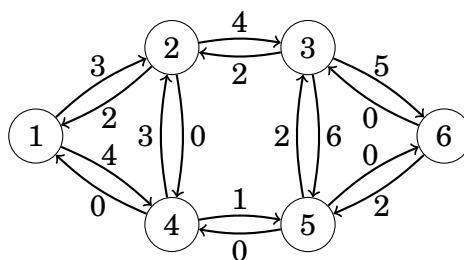
O algoritmo de Ford–Fulkerson consiste em várias rodadas. Em cada rodada, o algoritmo encontra um caminho da fonte para o sumidouro tal que cada aresta no caminho tenha um peso positivo. Se houver mais de um caminho possível disponível, podemos escolher qualquer um deles.

Por exemplo, suponha que escolhemos o seguinte caminho:



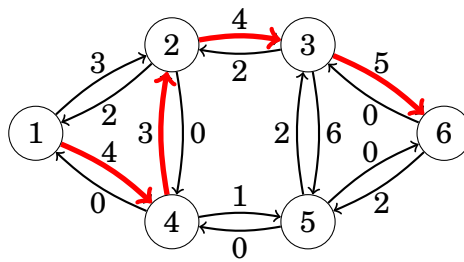
Após escolher o caminho, o fluxo aumenta em x unidades, onde x é o menor peso de aresta no caminho. Além disso, o peso de cada aresta no caminho diminui em x e o peso de cada aresta reversa aumenta em x .

No caminho acima, os pesos das arestas são 5, 6, 8 e 2. O menor peso é 2, então o fluxo aumenta em 2 e o novo grafo é o seguinte:



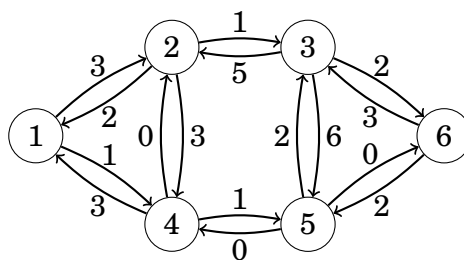
A ideia é que aumentar o fluxo diminui a quantidade de fluxo que pode passar pelas arestas no futuro. Por outro lado, é possível cancelar o fluxo posteriormente usando as arestas reversas do grafo se isso se mostrar benéfico para direcionar o fluxo de outra forma.

O algoritmo aumenta o fluxo enquanto houver um caminho da fonte para o sumidouro por meio de arestas de peso positivo. No presente exemplo, nosso próximo caminho pode ser o seguinte:

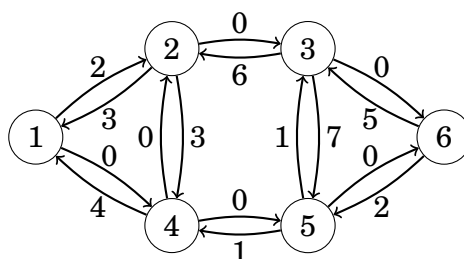


O menor peso de aresta neste caminho é 3, então o caminho aumenta o fluxo em 3, e o fluxo total após o processamento do caminho é 5.

O novo grafo será o seguinte:



Ainda precisamos de mais duas rodadas antes de atingir o fluxo máximo. Por exemplo, podemos escolher os caminhos $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ e $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$. Ambos os caminhos aumentam o fluxo em 1, e o grafo final é o seguinte:



Não é possível aumentar mais o fluxo, pois não há caminho da fonte para o sumidouro com arestas de peso positivo. Portanto, o algoritmo termina e o fluxo máximo é 7.

Encontrando caminhos

O algoritmo de Ford–Fulkerson não especifica como devemos escolher os caminhos que aumentam o fluxo. De qualquer forma, o algoritmo terminará mais cedo ou mais tarde e encontrará corretamente o fluxo máximo. No entanto, a eficiência do algoritmo depende da maneira como os caminhos são escolhidos.

Uma maneira simples de encontrar caminhos é usar a busca em profundidade. Geralmente, isso funciona bem, mas no pior caso, cada caminho só aumenta o fluxo em 1 e o algoritmo é lento. Felizmente, podemos evitar esta situação usando uma das seguintes técnicas:

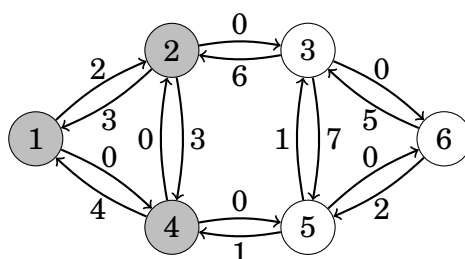
O **algoritmo de Edmonds-Karp** [18] escolhe cada caminho de forma que o número de arestas no caminho seja o menor possível. Isso pode ser feito utilizando busca em largura em vez de busca em profundidade para encontrar os caminhos. Pode-se provar que isso garante que o fluxo aumente rapidamente e que a complexidade de tempo do algoritmo seja $O(m^2n)$.

O **algoritmo de escalonamento** [2] utiliza busca em profundidade para encontrar caminhos onde o peso de cada aresta seja pelo menos um valor limite. Inicialmente, o valor limite é algum número grande, por exemplo, a soma de todos os pesos das arestas do grafo. Sempre que um caminho não puder ser encontrado, o valor limite é dividido por 2. A complexidade de tempo do algoritmo é $O(m^2 \log c)$, onde c é o valor limite inicial.

Na prática, o algoritmo de escalonamento é mais fácil de implementar, pois a busca em profundidade pode ser utilizada para encontrar os caminhos. Ambos os algoritmos são eficientes o suficiente para problemas que normalmente aparecem em competições de programação.

Cortes mínimos

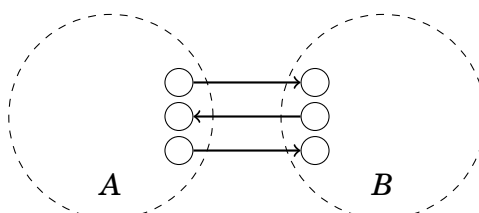
Verifica-se que, uma vez que o algoritmo de Ford-Fulkerson encontra um fluxo máximo, ele também determina um corte mínimo. Seja A o conjunto de nós que podem ser alcançados a partir da fonte usando arestas de peso positivo. No grafo de exemplo, A contém os nós 1, 2 e 4:



Agora, o corte mínimo consiste nas arestas do grafo original que começam em algum nó em A , terminam em algum nó fora de A , e cuja capacidade é totalmente utilizada no fluxo máximo. No grafo acima, tais arestas são $2 \rightarrow 3$ e $4 \rightarrow 5$, que correspondem ao corte mínimo $6 + 1 = 7$.

Por que o fluxo produzido pelo algoritmo é máximo e por que o corte é mínimo? A razão é que um grafo não pode conter um fluxo cujo tamanho seja maior que o peso de qualquer corte do grafo. Portanto, sempre que um fluxo e um corte têm o mesmo tamanho, eles são um fluxo máximo e um corte mínimo.

Vamos considerar qualquer corte do grafo tal que a fonte pertença a A , o sumidouro pertença a B e haja algumas arestas entre os conjuntos:



O tamanho do corte é a soma das arestas que vão de A para B . Este é um limite superior para o fluxo no grafo, pois o fluxo precisa prosseguir de A para B . Assim, o tamanho de um fluxo máximo é menor ou igual a o tamanho de qualquer corte no grafo.

Por outro lado, o algoritmo de Ford-Fulkerson produz um fluxo cujo tamanho é *exatamente* igual ao tamanho de um corte no grafo. Assim, o fluxo precisa ser um fluxo máximo e o corte precisa ser um corte mínimo.

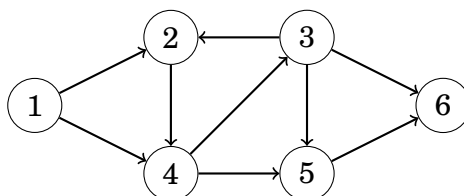
20.2 Caminhos disjuntos

Muitos problemas de grafos podem ser resolvidos reduzindo-os ao problema do fluxo máximo. Nosso primeiro exemplo de tal problema é o seguinte: recebemos um grafo direcionado com uma fonte e um sumidouro, e nossa tarefa é encontrar o número máximo de caminhos disjuntos da fonte ao sumidouro.

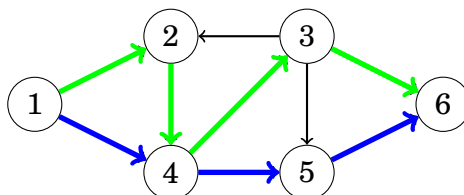
Caminhos Disjuntos nas Arestas

Vamos primeiro focar no problema de encontrar o número máximo de **caminhos disjuntos nas arestas** da origem ao sumidouro. Isso significa que devemos construir um conjunto de caminhos de forma que cada aresta apareça em, no máximo, um caminho.

Por exemplo, considere o seguinte grafo:



Neste grafo, o número máximo de caminhos disjuntos nas arestas é 2. Podemos escolher os caminhos $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ e $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ como segue:



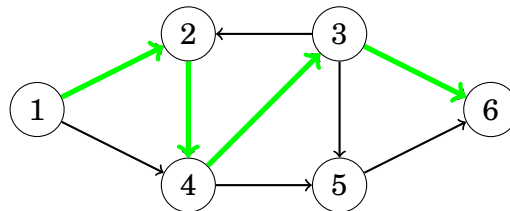
Verifica-se que o número máximo de caminhos disjuntos nas arestas é igual ao fluxo máximo do grafo, assumindo que a capacidade de cada aresta é um. Depois que o fluxo máximo é construído, os caminhos disjuntos nas arestas podem ser encontrados avidamente seguindo os caminhos da origem ao sumidouro.

Caminhos Disjuntos nos Vértices

Agora vamos considerar outro problema: encontrar o número máximo de **caminhos disjuntos nos vértices** da origem ao sumidouro. Neste problema,

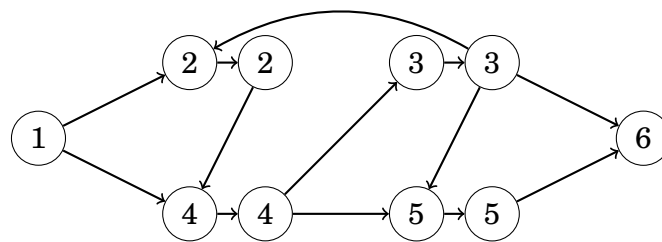
cada vértice, exceto a origem e o sumidouro, pode aparecer em, no máximo, um caminho. O número de caminhos disjuntos nos vértices pode ser menor que o número de caminhos disjuntos nas arestas.

Por exemplo, no grafo anterior, o número máximo de caminhos disjuntos nos vértices é 1:

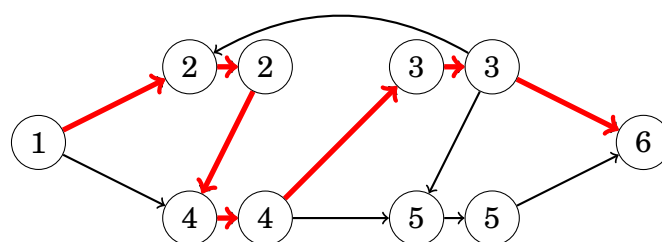


Também podemos reduzir este problema ao problema do fluxo máximo. Como cada vértice pode aparecer em, no máximo, um caminho, temos que limitar o fluxo que passa pelos vértices. Um método padrão para isso é dividir cada vértice em dois vértices, de forma que o primeiro vértice tenha as arestas de entrada do vértice original, o segundo vértice tenha as arestas de saída do vértice original e haja uma nova aresta do primeiro vértice para o segundo vértice.

No nosso exemplo, o grafo se torna o seguinte:



O fluxo máximo para o grafo é o seguinte:



Assim, o número máximo de caminhos disjuntos nos vértices da origem ao sumidouro é 1.

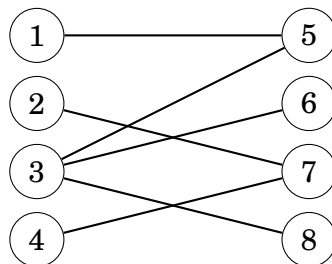
20.3 Emparelhamentos Máximos

O problema do **emparelhamento máximo** pede para encontrar um conjunto de tamanho máximo de pares de vértices em um grafo não direcionado, de forma que cada par esteja conectado por uma aresta e cada vértice pertença a, no máximo, um par.

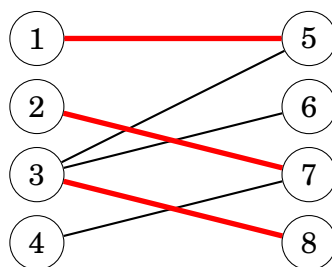
Existem algoritmos polinomiais para encontrar emparelhamentos máximos em grafos gerais [17], mas tais algoritmos são complexos e raramente vistos em competições de programação. No entanto, em grafos bipartidos, o problema do emparelhamento máximo é muito mais fácil de resolver, porque podemos reduzi-lo ao problema do fluxo máximo.

Encontrando Emparelhamentos Máximos

Os vértices de um grafo bipartido sempre podem ser divididos em dois grupos, de forma que todas as arestas do grafo vão do grupo esquerdo para o grupo direito. Por exemplo, no seguinte grafo bipartido, os grupos são $\{1, 2, 3, 4\}$ e $\{5, 6, 7, 8\}$.

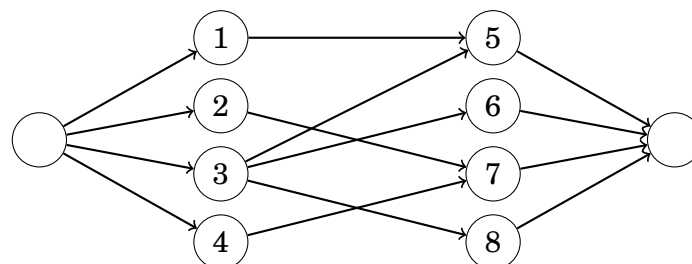


O tamanho de um emparelhamento máximo deste grafo é 3:

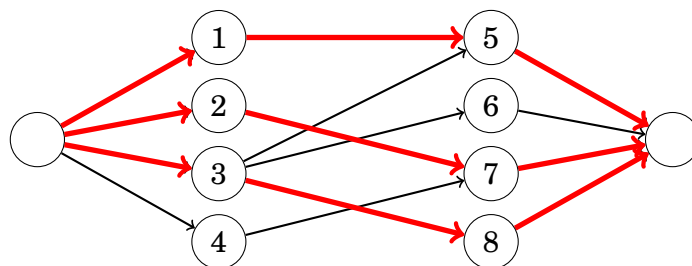


Podemos reduzir o problema do emparelhamento máximo bipartido ao problema do fluxo máximo adicionando dois novos vértices ao grafo: uma origem e um sumidouro. Também adicionamos arestas da origem para cada vértice esquerdo e de cada vértice direito para o sumidouro. Depois disso, o tamanho de um fluxo máximo no grafo é igual ao tamanho de um emparelhamento máximo no grafo original.

Por exemplo, a redução para o grafo acima é a seguinte:



O fluxo máximo deste grafo é o seguinte:

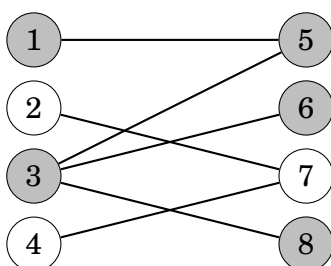


Teorema de Hall

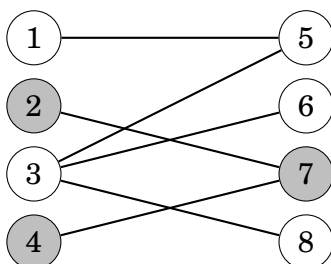
O **Teorema de Hall** pode ser usado para descobrir se um grafo bipartido possui um emparelhamento que contém todos os vértices esquerdos ou direitos. Se o número de vértices esquerdos e direitos for o mesmo, o Teorema de Hall nos diz se é possível construir um **emparelhamento perfeito** que contenha todos os vértices do grafo.

Assuma que queremos encontrar um emparelhamento que contenha todos os vértices esquerdos. Seja X qualquer conjunto de vértices esquerdos e seja $f(X)$ o conjunto de seus vizinhos. De acordo com o Teorema de Hall, um emparelhamento que contém todos os vértices esquerdos existe exatamente quando, para cada X , a condição $|X| \leq |f(X)|$ é válida.

Vamos estudar o Teorema de Hall no grafo de exemplo. Primeiro, seja $X = \{1, 3\}$ o que produz $f(X) = \{5, 6, 8\}$:



A condição do Teorema de Hall é válida, pois $|X| = 2$ e $|f(X)| = 3$. Em seguida, seja $X = \{2, 4\}$ o que produz $f(X) = \{7\}$:



Neste caso, $|X| = 2$ e $|f(X)| = 1$, então a condição do Teorema de Hall não é válida. Isso significa que não é possível formar um emparelhamento perfeito para o grafo. Este resultado não é surpreendente, porque já sabemos que o emparelhamento máximo do grafo é 3 e não 4.

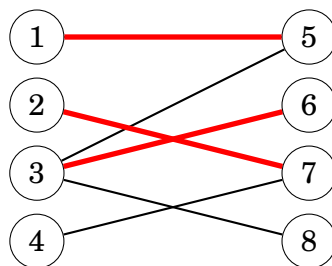
Se a condição do Teorema de Hall não for válida, o conjunto X fornece uma explicação *por que* não podemos formar tal emparelhamento. Como X contém

mais vértices do que $f(X)$, não há pares para todos os vértices em X . Por exemplo, no grafo acima, ambos os vértices 2 e 4 deveriam ser conectados ao vértice 7, o que não é possível.

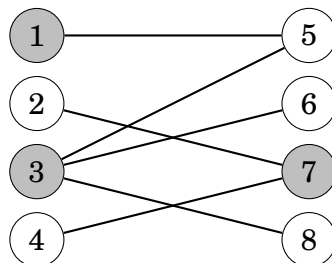
Teorema de Kőnig

Uma **cobertura mínima por vértices** de um grafo é um conjunto mínimo de vértices, de forma que cada aresta do grafo tenha pelo menos um vértice no conjunto. Em um grafo geral, encontrar uma cobertura mínima por vértices é um problema NP-difícil. No entanto, se o grafo for bipartido, o **Teorema de Kőnig** nos diz que o tamanho de uma cobertura mínima por vértices e o tamanho de um emparelhamento máximo são sempre iguais. Assim, podemos calcular o tamanho de uma cobertura mínima por vértices usando um algoritmo de fluxo máximo.

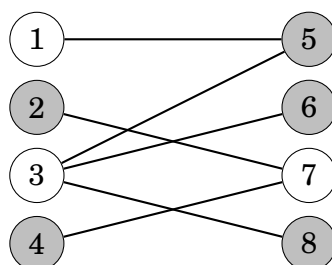
Vamos considerar o seguinte grafo com um emparelhamento máximo de tamanho 3:



Agora, o Teorema de Kőnig nos diz que o tamanho de uma cobertura mínima por vértices também é 3. Tal cobertura pode ser construída da seguinte forma:



Os vértices que *não* pertencem a uma cobertura mínima por vértices formam um **conjunto independente máximo**. Este é o maior conjunto possível de vértices, de forma que não haja dois vértices no conjunto que estejam conectados por uma aresta. Mais uma vez, encontrar um conjunto independente máximo em um grafo geral é um problema NP-difícil, mas, em um grafo bipartido, podemos usar o Teorema de Kőnig para resolver o problema de forma eficiente. No grafo de exemplo, o conjunto independente máximo é o seguinte:

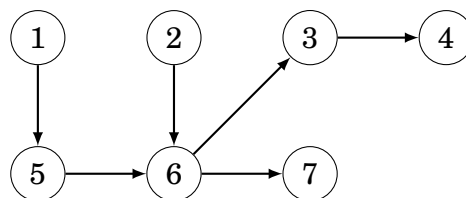


20.4 Coberturas por Caminhos

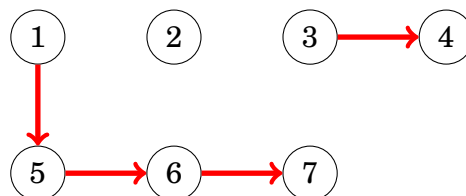
Uma **cobertura por caminhos** é um conjunto de caminhos em um grafo, de forma que cada vértice do grafo pertença a pelo menos um caminho. Acontece que, em grafos direcionados acíclicos, podemos reduzir o problema de encontrar uma cobertura mínima por caminhos ao problema de encontrar um fluxo máximo em outro grafo.

Cobertura por Caminhos Disjuntos nos Vértices

Em uma **cobertura por caminhos disjuntos nos vértices**, cada vértice pertence a exatamente um caminho. Como exemplo, considere o seguinte grafo:



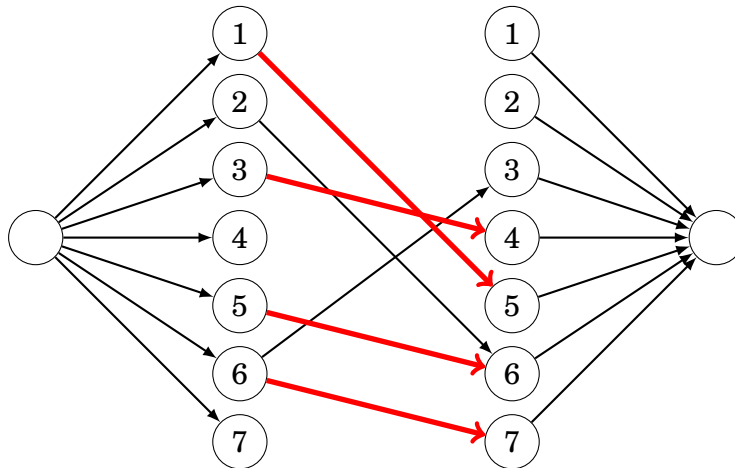
Uma cobertura mínima por caminhos disjuntos nos vértices deste grafo consiste em três caminhos. Por exemplo, podemos escolher os seguintes caminhos:



Observe que um dos caminhos contém apenas o vértice 2, então é possível que um caminho não contenha nenhuma aresta.

Podemos encontrar uma cobertura mínima por caminhos disjuntos nos vértices construindo um *grafo de emparelhamento*, onde cada vértice do grafo original é representado por dois vértices: um vértice esquerdo e um vértice direito. Há uma aresta de um vértice esquerdo para um vértice direito se houver tal aresta no grafo original. Além disso, o grafo de emparelhamento contém uma origem e um sumidouro, e há arestas da origem para todos os vértices esquerdos e de todos os vértices direitos para o sumidouro.

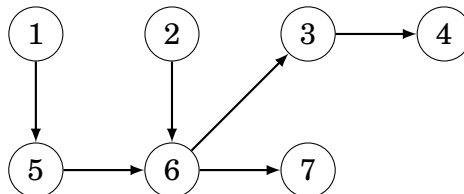
Um emparelhamento máximo no grafo resultante corresponde a uma cobertura mínima por caminhos disjuntos nos vértices no grafo original. Por exemplo, o seguinte grafo de emparelhamento para o grafo acima contém um emparelhamento máximo de tamanho 4:



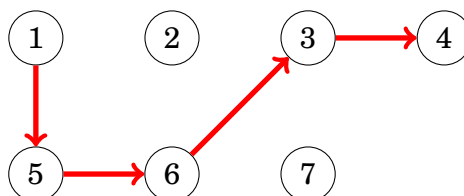
Cada aresta no emparelhamento máximo do grafo de emparelhamento corresponde a uma aresta na cobertura mínima por caminhos disjuntos nos vértices do grafo original. Assim, o tamanho da cobertura mínima por caminhos disjuntos nos vértices é $n - c$, onde n é o número de vértices no grafo original e c é o tamanho do emparelhamento máximo.

Cobertura Geral por Caminhos

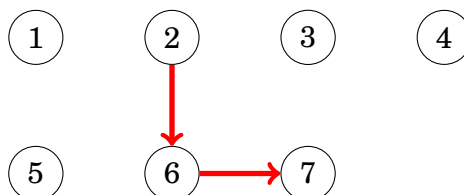
Uma **cobertura geral por caminhos** é uma cobertura por caminhos onde um vértice pode pertencer a mais de um caminho. Uma cobertura geral mínima por caminhos pode ser menor que uma cobertura mínima por caminhos disjuntos nos vértices, porque um vértice pode ser usado várias vezes em caminhos. Considere novamente o seguinte grafo:



A cobertura geral mínima por caminhos deste grafo consiste em dois caminhos. Por exemplo, o primeiro caminho pode ser o seguinte:

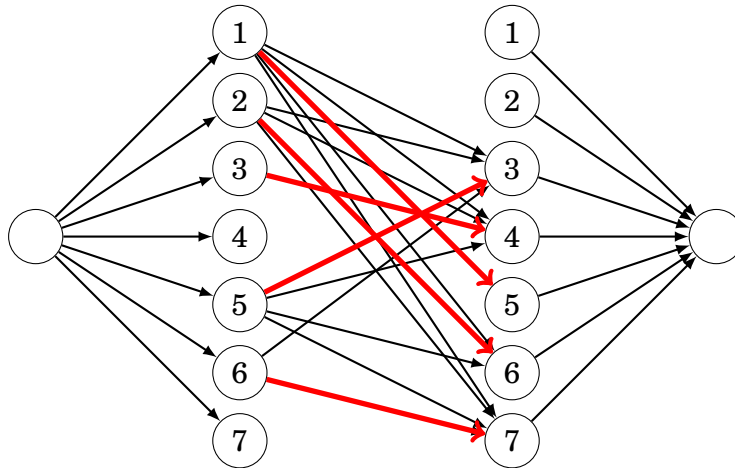


E o segundo caminho pode ser o seguinte:



Uma cobertura geral mínima por caminhos pode ser encontrada quase como uma cobertura mínima por caminhos disjuntos nos vértices. Basta adicionar algumas arestas novas ao grafo de emparelhamento, de forma que haja uma aresta $a \rightarrow b$ sempre que houver um caminho de a para b no grafo original (possivelmente através de várias arestas).

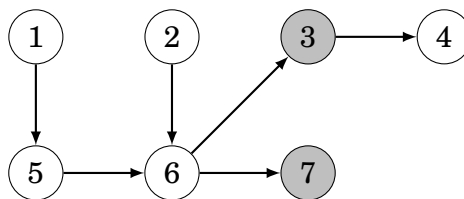
O grafo de emparelhamento para o grafo acima é o seguinte:



Teorema de Dilworth

Uma **anticadeia** é um conjunto de vértices de um grafo tal que não existe um caminho de qualquer vértice para outro vértice usando as arestas do grafo. O **Teorema de Dilworth** afirma que em um grafo direcionado acíclico, o tamanho de uma cobertura geral mínima por caminhos é igual ao tamanho de uma anticadeia máxima.

Por exemplo, os vértices 3 e 7 formam uma anticadeia no seguinte grafo:



Esta é uma anticadeia máxima, pois não é possível construir qualquer anticadeia que contenha três vértices. Vimos anteriormente que o tamanho de uma cobertura geral mínima por caminhos deste grafo consiste em dois caminhos.

Referências Bibliográficas

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).

- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>

- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).

- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).
- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

Índice Remissivo

- 2SUM problem, 86
- 3SUM problem, 87

- algoritmo cúbico, 22
- Algoritmo de Bellman–Ford, 133
- Algoritmo de Dijkstra, 136
- algoritmo de Dijkstra, 165
- Algoritmo de Edmonds–Karp, 198
- algoritmo de escalonamento, 199
- Algoritmo de Floyd, 168
- Algoritmo de Floyd–Warshall, 139
- Algoritmo de Ford–Fulkerson, 196
- algoritmo de Kadane, 25
- algoritmo de Kosaraju, 170
- Algoritmo de Kruskal, 152
- Algoritmo de Prim, 157
- algoritmo de tempo constante, 22
- algoritmo guloso, 63
- algoritmo linear, 22
- algoritmo logarítmico, 22
- algoritmo polinomial, 23
- algoritmo quadrático, 22
- Algoritmo SPFA, 136
- amortized analysis, 85
- ancestor, 175
- anticadeia, 207
- aresta, 117
- aritmética modular, 7

- backtracking, 56
- binary indexed tree, 94
- binary tree, 149
- bitset, 46
- bubble sort, 27
- busca binária, 34
- busca em largura, 127
- busca em profundidade, 125

- caminho, 117
- caminho mínimo, 133
- child, 143
- ciclo, 117, 129, 161, 167
- ciclo negativo, 135
- classes de complexidade, 22
- cobertura mínima por vértices, 204
- cobertura por caminhos, 205
- cobertura por vértices, 204
- Codificação de Huffman, 69
- coloração, 120
- complemento, 12
- complexidade de tempo, 19
- componente, 118
- componente fortemente conexo, 169
- compressão de dados, 68
- compressão de índices, 101
- conjunto, 12
- conjunto independente, 204
- conjunto independente máximo, 204
- conjunto universo, 12
- conjunção, 13
- corte, 196
- corte mínimo, 196, 199
- counting sort, 30
- código binário, 68

- data structure, 39
- De Bruijn sequence, 192
- deque, 47
- deslocamento de bits, 105
- detecção de ciclo, 167
- diameter, 145
- diferença, 12
- Dirac’s theorem, 192
- disjunção, 13
- distância de edição, 80

distância de Hamming, 108
 distância de Levenshtein, 80
 dynamic array, 39

 emparelhamento, 201
 emparelhamento máximo, 201
 emparelhamento perfeito, 203
 encontro no meio, 60
 entrada e saída, 4
 equivalência, 13
 Estrutura union-find, 155
 Euler tour technique, 180
 Eulerian circuit, 188
 Eulerian path, 187

 fator constante, 23
 fatorial, 14
 Fenwick tree, 94
 fluxo, 195
 fluxo máximo, 195
 função de comparação, 34
 fórmula de Binet, 14
 fórmula de Faulhaber, 10

 grafo, 117
 grafo bipartido, 120, 130
 grafo completo, 119
 grafo conectado, 129
 grafo conexo, 118
 grafo de componentes, 169
 grafo de sucessores, 166
 grafo direcionado, 118
 grafo fortemente conexo, 169
 grafo funcional, 166
 grafo ponderado, 119
 grafo regular, 119
 grafo simples, 120
 grau, 119
 grau de entrada, 119
 grau de saída, 119

 Hamiltonian circuit, 191
 Hamiltonian path, 191
 heap, 48
 heuristic, 194
 Hierholzer's algorithm, 189

 implicação, 13

 in-order, 149
 inteiro, 6
 intersecção, 12
 inversão, 28
 iterator, 43

 janela deslizante, 89

 knight's tour, 193

 leaf, 143
 linguagem de programação, 3
 lista de adjacência, 121
 lista de arestas, 123
 logaritmo, 14
 logaritmo natural, 15
 lowest common ancestor, 179
 lógica, 13

 macro, 9
 maior subsequência crescente, 76
 map, 42
 matriz de adjacência, 122
 maximum query, 91
 memoização, 73
 merge sort, 29
 minimum query, 91
 mochila, 79
 mínimo da janela deslizante, 89

 nearest smaller elements, 87
 negação, 13
 next_permutation, 55
 nó, 117
 número de Fibonacci, 14
 números com ponto flutuante, 7

 operador de comparação, 33
 operação E, 104
 operação NÃO, 105
 operação OU, 104
 operação XOR, 105
 Ordenação, 27
 ordenação topológica, 161
 Ore's theorem, 192

 pair, 33
 palavra de código, 68
 parent, 143

permutation, 55
post-order, 149
pre-order, 149
predicado, 13
prefix sum array, 92
priority queue, 48
problema 2SAT, 172
problema 3SAT, 174
problema NP-difícil, 23
programação dinâmica, 71
progressão aritmética, 11

quantificador, 13
queen problem, 56
queue, 48

random_shuffle, 43
range query, 91
representação de bits, 103
resto, 7
reverse, 43
root, 143
rooted tree, 143

segment tree, 97
set, 41
soma harmônica, 12
soma máxima de subvetor, 24
sort, 32, 43
sparse table, 93

stack, 47
string, 40
subconjuntos, 12
subset, 53
subtree, 143
sum query, 91

Teorema de Dilworth, 207
Teorema de Hall, 203
Teorema de König, 204
teoria dos conjuntos, 12
tree, 143
tree query, 175
tree traversal array, 176
tuple, 33
typedef, 8
two pointers method, 85

união, 12

vector, 39
vetor de diferenças, 102
vizinho, 119

Warnsdorf's rule, 194

Árvore geradora máxima, 152
Árvore geradora mínima, 151
árvore, 118
árvore geradora, 151