Manual do Programador Competitivo

Antti Laaksonen

Rascunho de 17 de julho de 2024

Sumário

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 inicio 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) {
    // codigo
}
```

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 $-2^{31} \ldots 2^{31}-1$, ou cerca de $-2 \cdot 10^9 \ldots 2 \cdot 10^9$. Se o tipo `int` não for suficiente, o tipo de 64 bits `long long` pode ser utilizado. Ele possui uma faixa de valores de $-2^{63} \ldots 2^{63}-1$, ou aproximadamente $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$.

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 $-2^{127} \ldots 2^{127}-1$, ou aproximadamente $-10^{38} \ldots 10^{38}$. No entanto, este tipo não está disponível em todos os sistemas de competição.

Aritmética modular

Denotamos por x mod 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 $0 \ldots m-1$. 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 ($\varepsilon = 10^{-9}$):

```
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

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

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

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

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

e

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

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

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

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

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

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

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

Multiplicando ambos os lados por k, obtemos

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

e resolvendo a equação

$$kS - S = bk - a$$

obtemos a fórmula.

Um caso especial da soma de progressão aritmética é a fórmula

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

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

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

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 $n = 6$, pomode 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 $\{1, 2, \ldots, 10\}$, 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 $\mathbb{N} = \{0,1,2,\ldots\}$ ou $\mathbb{N} = \{1,2,3,\ldots\}$.

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

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

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

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

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

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

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

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

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 consequentemente,

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

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 número[2] 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]

[2] O 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 $O(\cdots)$ 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++) {
    // codigo
}
```

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++) {
        // codigo
    }
```

```
}
```

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++) {
    // codigo
}
```

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

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

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++) {
        // codigo
    }
}
```

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

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

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++) {
        // codigo
    }
}
```

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
$\cdots$	$\cdots$
$g(1)$	2^{n-1}

Com base nisso, a complexidade de tempo é

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

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 conhecido[1].

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.

[1] Um 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 vetor[2]. 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";
```

[2] O 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++) {
```

[3]Em [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

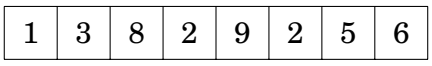
algoritmo de ordenação em tempo $O(n^2)$ é o **bubble sort** onde os elementos "flutuam"no array de acordo com seus valores.

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:

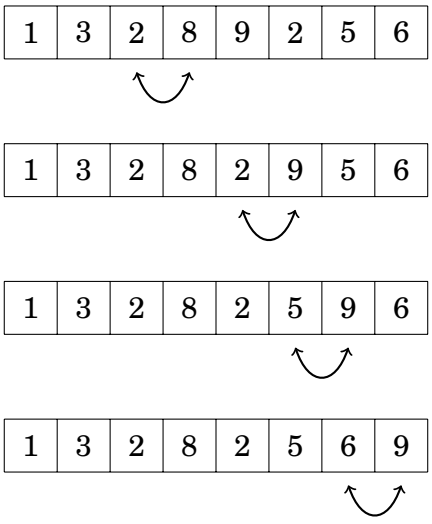
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```

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



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 $(\texttt{array}[a], \texttt{array}[b])$ tal que $a < b$ and $\texttt{array}[a] > \texttt{array}[b]$, 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:

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

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 $\texttt{array}[a \ldots b]$ 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 $\texttt{array}[a \ldots k]$.
4. Ordene recursivamente o subarray $\texttt{array}[k+1 \ldots b]$.
5. *Junte* os subarrays ordenados $\texttt{array}[a \ldots k]$ e $\texttt{array}[k+1 \ldots b]$ em um subarray ordenado $\texttt{array}[a \ldots b]$.

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 $\texttt{array}[a \ldots k]$ e $\texttt{array}[k+1 \ldots b]$ 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:

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

1	2	3	6

2	5	8	9

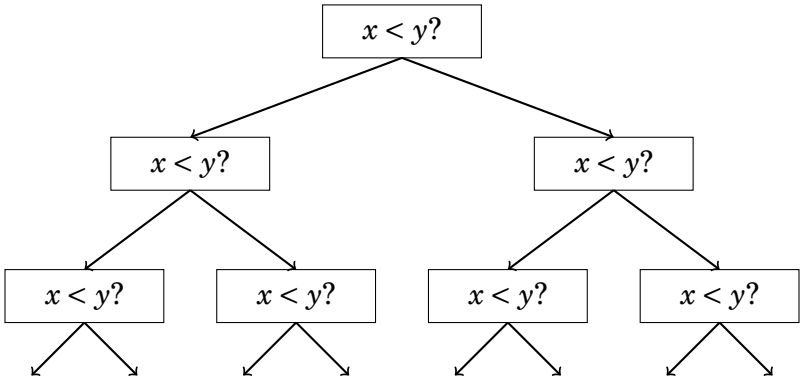
Finalmente, o algoritmo junta os subarrays ordenados e cria o array final ordenado:

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

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

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 $0 \ldots c$ 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.

[2]Note 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 indice k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Nesta implementação, a região ativa é $a \ldots b$, e inicialmente a região é $0 \ldots n-1$. 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 indice 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 indice 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. emos 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	$\cdots$	$k-1$	k	$k+1$	$\cdots$
ok(x)	false	false	$\cdots$	false	true	true	$\cdots$

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 vector 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 $0, 1, \ldots, n-1$, 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["harpsichord"] = 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 ”aybabtu” com valor 0 é adicionada ao mapa.

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

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

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

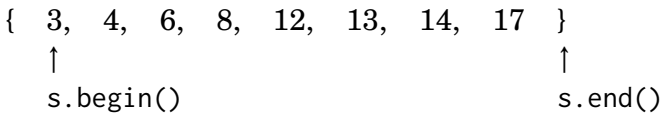
O código a seguir imprime todas as chaves e valores em um mapa:

```
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:



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 $1 \ldots 10^9$:

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

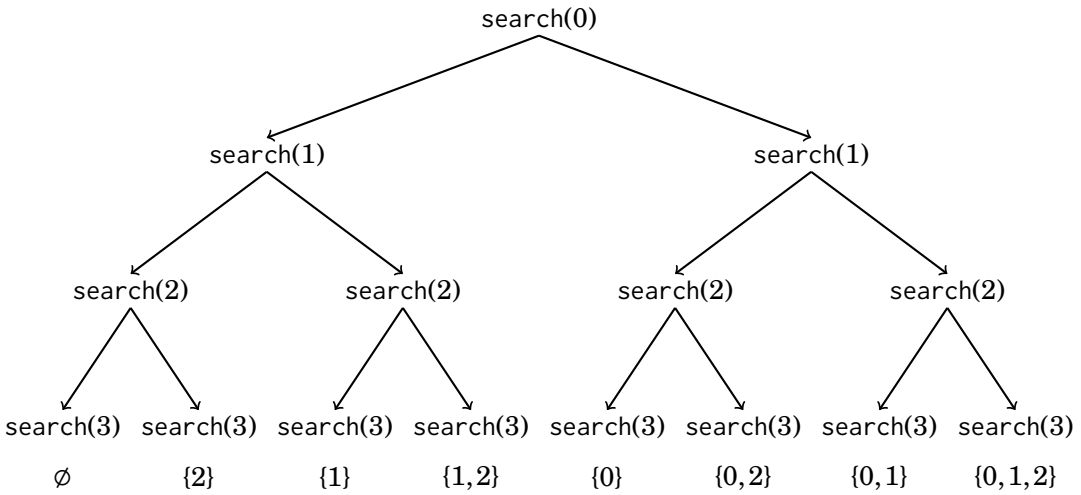
Uma maneira elegante de percorrer todos os subconjuntos de um conjunto é usar recursão. A seguinte função `search` gera os subconjuntos do conjunto $\{0,1,\ldots,n-1\}$. A função mantém um vetor `subset` que conterá os elementos de cada subconjunto. A busca começa quando a função é chamada com o parâmetro 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 $0 \ldots 2^n - 1$. 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 $\{0,1,\ldots,n-1\}$. 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 $\{0,1,\ldots,n-1\}$ 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:

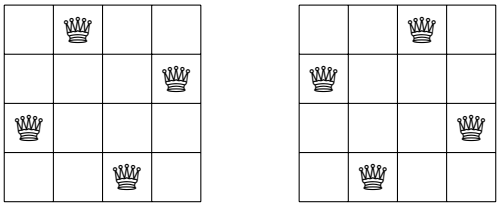
```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
```

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

5.3 Backtracking

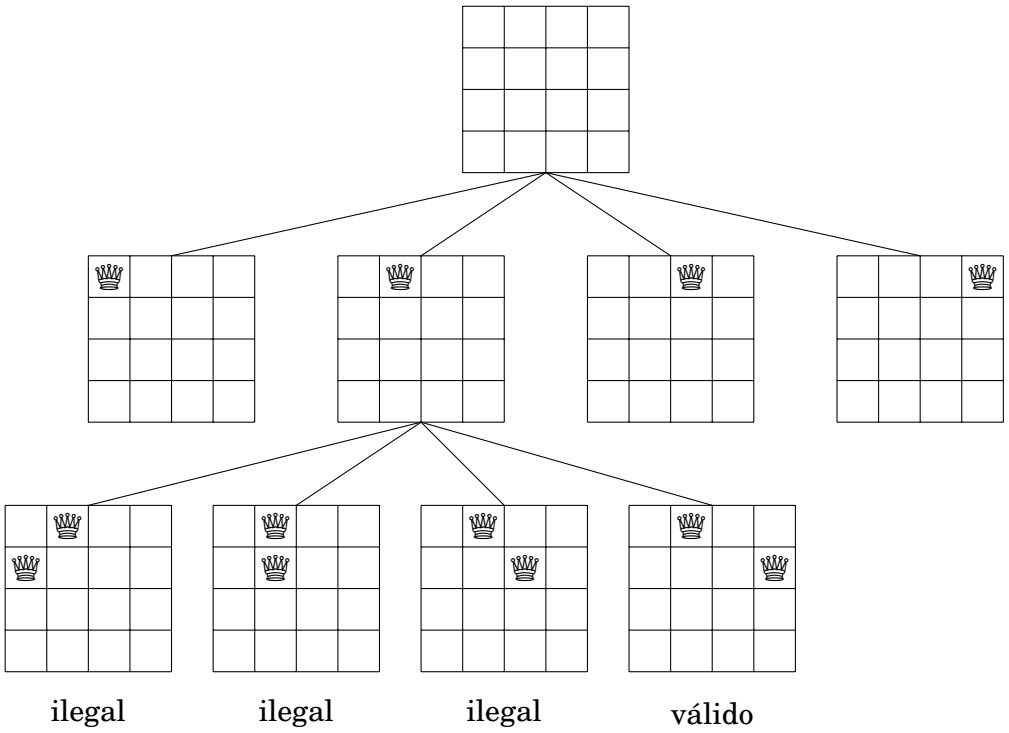
Um algoritmo de **backtracking** começa com uma solução vazia e estende a solução passo a passo. A busca percorre recursivamente todas as maneiras diferentes de como uma solução pode ser construída.

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`

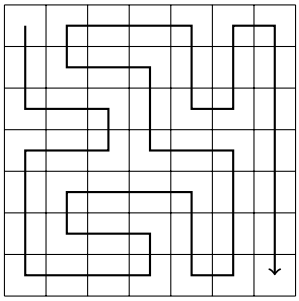
Seja $q(n)$ o número de maneiras de colocar n rainhas em um tabuleiro de xadrez $n \times n$. O algoritmo de backtracking acima nos diz que, por exemplo, $q(8) = 92$. Quando n aumenta, a busca rapidamente se torna lenta, porque o número de soluções aumenta exponencialmente. Por exemplo, calcular $q(16) = 14772512$ usando o algoritmo acima já leva cerca de um minuto em um computador moderno[1].

[1]Não 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.

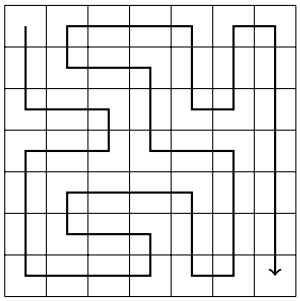
Algoritmo básico

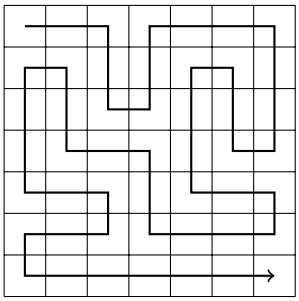
A primeira versão do algoritmo não contém nenhuma otimização. Nós simplesmente usamos backtracking para gerar todos os caminhos possíveis do canto superior esquerdo para o canto inferior direito e contamos o número de tais caminhos.

- 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:



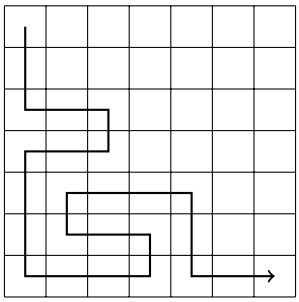


Portanto, podemos decidir que sempre nos movemos um passo para baixo (ou para a direita), e finalmente multiplicamos o número de soluções por dois.

- 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:

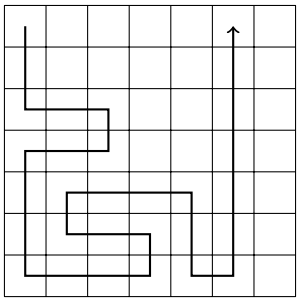


Usando essa observação, podemos encerrar a busca imediatamente se atingirmos o quadrado inferior direito muito cedo.

- 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:

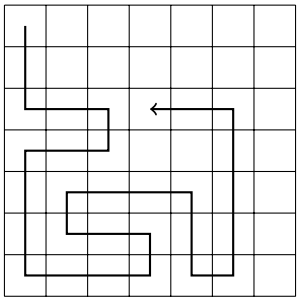


Neste caso, não podemos mais visitar todos os quadrados, então podemos encerrar a busca. Esta otimização é muito útil:

- 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 eficiente[2]. 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 contenham 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.

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

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 multicommodity 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 componemts 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