Manual do Programador Competitivo

Antti Laaksonen

Rascunho de 30 de maio de 2023

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.

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