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. Pressupõe-se que você já conheça os conceitos básicos de programação, mas não é necessário ter experiência prévia em 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). 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 é também uma oportunidade para aprender muito. Você pode ter certeza de que obterá um bom entendimento geral de algoritmos se dedicar tempo lendo o livro, resolvendo problemas e participando de contests.

O livro está em constante desenvolvimento. Você pode sempre enviar feedback sobre o livro para `ahslaaks@cs.helsinki.fi`.

Helsinki, Agosto 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 não precisam ser mantidos 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 várias linguagens.

Muitas pessoas pensam que C++ é a melhor escolha para um programador competitivo e quase sempre está disponível nos sistemas de competição. Os benefícios de usar C++ são que é uma linguagem muito eficiente e sua biblioteca padrão contém 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 no 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 seja 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.

C++ template de código

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

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

using namespace std;

int main() {
    // solucao vem 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 sim, 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 provoca uma operação de limpeza.

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 loop é útil:

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

Este loop 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 cerca de $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$.

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 cerca de $-10^{38} \ldots 10^{38}$. 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

Short code is ideal in competitive programming, because programs should be written as fast as possible. Because of this, competitive programmers often define shorter names for datatypes and other parts of code.

Type names

Using the command `typedef` it is possible to give a shorter name to a datatype. For example, the name `long long` is long, so we can define a shorter name `ll`:

```
typedef long long ll;
```

After this, the code

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

can be shortened as follows:

```
ll a = 123456789;
```

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

The command `typedef` can also be used with more complex types. For example, the following code gives the name `vi` for a vector of integers and the name `pi` for a pair that contains two integers.

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

Macros

Another way to shorten code is to define **macros**. A macro means that certain strings in the code will be changed before the compilation. In C++, macros are defined using the `#define` keyword.

For example, we can define the following macros:

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

After this, the code

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

can be shortened as follows:

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

A macro can also have parameters which makes it possible to shorten loops and other structures. For example, we can define the following macro:

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

After this, the code

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

can be shortened as follows:

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

```
}
```

Sometimes macros cause bugs that may be difficult to detect. For example, consider the following macro that calculates the square of a number:

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

This macro *does not* always work as expected. For example, the code

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

corresponds to the code

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

A better version of the macro is as follows:

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

Now the code

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

corresponds to the code

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

1.5 Mathematics

Mathematics plays an important role in competitive programming, and it is not possible to become a successful competitive programmer without having good mathematical skills. This section discusses some important mathematical concepts and formulas that are needed later in the book.

Sum formulas

Each sum of the form

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

where k is a positive integer, has a closed-form formula that is a polynomial of degree $k+1$. For example[1],

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

[1] There is even a general formula for such sums, called **Faulhaber's formula**, but it is too complex to be presented here.

and

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

An **arithmetic progression** is a sequence of numbers where the difference between any two consecutive numbers is constant. For example,

$$3, 7, 11, 15$$

is an arithmetic progression with constant 4. The sum of an arithmetic progression can be calculated using the formula

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

where a is the first number, b is the last number and n is the amount of numbers. For example,

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

The formula is based on the fact that the sum consists of n numbers and the value of each number is $(a+b)/2$ on average.

A **geometric progression** is a sequence of numbers where the ratio between any two consecutive numbers is constant. For example,

$$3, 6, 12, 24$$

is a geometric progression with constant 2. The sum of a geometric progression can be calculated using the formula

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

where a is the first number, b is the last number and the ratio between consecutive numbers is k. For example,

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

This formula can be derived as follows. Let

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

By multiplying both sides by k, we get

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

and solving the equation

$$kS - S = bk - a$$

yields the formula.

A special case of a sum of a geometric progression is the formula

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

A **harmonic sum** is a sum of the form

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

An upper bound for a harmonic sum is $\log_2(n)+1$. Namely, we can modify each term $1/k$ so that k becomes the nearest power of two that does not exceed k. For example, when $n = 6$, we can estimate the sum as follows:

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

This upper bound consists of $\log_2(n)+1$ parts (1, $2 \cdot 1/2$, $4 \cdot 1/4$, etc.), and the value of each part is at most 1.

Set theory

A **set** is a collection of elements. For example, the set

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

contains elements 2, 4 and 7. The symbol $\emptyset$ denotes an empty set, and $|S|$ denotes the size of a set S, i.e., the number of elements in the set. For example, in the above set, $|X| = 3$.

If a set S contains an element x, we write $x \in S$, and otherwise we write $x \notin S$. For example, in the above set

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

New sets can be constructed using set operations:

- The **intersection** $A \cap B$ consists of elements that are in both A and B. For example, if $A = \{1,2,5\}$ and $B = \{2,4\}$, then $A \cap B = \{2\}$.
- The **union** $A \cup B$ consists of elements that are in A or B or both. For example, if $A = \{3,7\}$ and $B = \{2,3,8\}$, then $A \cup B = \{2,3,7,8\}$.
- The **complement** $\bar{A}$ consists of elements that are not in A. The interpretation of a complement depends on the **universal set**, which contains all possible elements. For example, if $A = \{1,2,5,7\}$ and the universal set is $\{1,2,\ldots,10\}$, then $\bar{A} = \{3,4,6,8,9,10\}$.
- The **difference** $A \setminus B = A \cap \bar{B}$ consists of elements that are in A but not in B. Note that B can contain elements that are not in A. For example, if $A = \{2,3,7,8\}$ and $B = \{3,5,8\}$, then $A \setminus B = \{2,7\}$.

If each element of A also belongs to S, we say that A is a **subset** of S, denoted by $A \subset S$. A set S always has $2^{|S|}$ subsets, including the empty set. For example, the subsets of the set $\{2,4,7\}$ are

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

Some often used sets are $\mathbb{N}$ (natural numbers), $\mathbb{Z}$ (integers), $\mathbb{Q}$ (rational numbers) and $\mathbb{R}$ (real numbers). The set $\mathbb{N}$ can be defined in two ways, depending on the situation: either $\mathbb{N} = \{0, 1, 2, \ldots\}$ or $\mathbb{N} = \{1, 2, 3, \ldots\}$.

We can also construct a set using a rule of the form

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

where $f(n)$ is some function. This set contains all elements of the form $f(n)$, where n is an element in S. For example, the set

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

contains all even integers.

Logic

The value of a logical expression is either **true** (1) or **false** (0). The most important logical operators are $\neg$ (**negation**), $\wedge$ (**conjunction**), $\vee$ (**disjunction**), $\Rightarrow$ (**implication**) and $\Leftrightarrow$ (**equivalence**). The following table shows the meanings of these operators:

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

The expression $\neg A$ has the opposite value of A. The expression $A \wedge B$ is true if both A and B are true, and the expression $A \vee B$ is true if A or B or both are true. The expression $A \Rightarrow B$ is true if whenever A is true, also B is true. The expression $A \Leftrightarrow B$ is true if A and B are both true or both false.

A **predicate** is an expression that is true or false depending on its parameters. Predicates are usually denoted by capital letters. For example, we can define a predicate $P(x)$ that is true exactly when x is a prime number. Using this definition, $P(7)$ is true but $P(8)$ is false.

A **quantifier** connects a logical expression to the elements of a set. The most important quantifiers are $\forall$ (**for all**) and $\exists$ (**there is**). For example,

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

means that for each element x in the set, there is an element y in the set such that y is smaller than x. This is true in the set of integers, but false in the set of natural numbers.

Using the notation described above, we can express many kinds of logical propositions. For example,

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

means that if a number x is larger than 1 and not a prime number, then there are numbers a and b that are larger than 1 and whose product is x. This proposition is true in the set of integers.

Functions

The function $\lfloor x \rfloor$ rounds the number x down to an integer, and the function $\lceil x \rceil$ rounds the number x up to an integer. For example,

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

The functions $\min(x_1, x_2, \ldots, x_n)$ and $\max(x_1, x_2, \ldots, x_n)$ give the smallest and largest of values $x_1, x_2, \ldots, x_n$. For example,

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

The **factorial** $n!$ can be defined

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

or recursively

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

The **Fibonacci numbers** arise in many situations. They can be defined recursively as follows:

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

The first Fibonacci numbers are

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

There is also a closed-form formula for calculating Fibonacci numbers, which is sometimes called **Binet's formula**:

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

Logarithms

The **logarithm** of a number x is denoted $\log_k(x)$, where k is the base of the logarithm. According to the definition, $\log_k(x) = a$ exactly when $k^a = x$.

A useful property of logarithms is that $\log_k(x)$ equals the number of times we have to divide x by k before we reach the number 1. For example, $\log_2(32) = 5$ because 5 divisions by 2 are needed:

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

Logarithms are often used in the analysis of algorithms, because many efficient algorithms halve something at each step. Hence, we can estimate the efficiency of such algorithms using logarithms.

The logarithm of a product is

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

and consequently,

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

In addition, the logarithm of a quotient is

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

Another useful formula is

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

and using this, it is possible to calculate logarithms to any base if there is a way to calculate logarithms to some fixed base.

The **natural logarithm** $\ln(x)$ of a number x is a logarithm whose base is $e \approx 2.71828$. Another property of logarithms is that the number of digits of an integer x in base b is $\lfloor \log_b(x) + 1 \rfloor$. For example, the representation of 123 in base 2 is 1111011 and $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Contests and resources

IOI

The International Olympiad in Informatics (IOI) is an annual programming contest for secondary school students. Each country is allowed to send a team of four students to the contest. There are usually about 300 participants from 80 countries.

The IOI consists of two five-hour long contests. In both contests, the participants are asked to solve three algorithm tasks of various difficulty. The tasks are divided into subtasks, each of which has an assigned score. Even if the contestants are divided into teams, they compete as individuals.

The IOI syllabus [41] regulates the topics that may appear in IOI tasks. Almost all the topics in the IOI syllabus are covered by this book.

Participants for the IOI are selected through national contests. Before the IOI, many regional contests are organized, such as the Baltic Olympiad in Informatics (BOI), the Central European Olympiad in Informatics (CEOI) and the Asia-Pacific Informatics Olympiad (APIO).

Some countries organize online practice contests for future IOI participants, such as the Croatian Open Competition in Informatics [11] and the USA Computing Olympiad [68]. In addition, a large collection of problems from Polish contests is available online [60].

ICPC

The International Collegiate Programming Contest (ICPC) is an annual programming contest for university students. Each team in the contest consists of three students, and unlike in the IOI, the students work together; there is only one computer available for each team.

The ICPC consists of several stages, and finally the best teams are invited to the World Finals. While there are tens of thousands of participants in the contest, there are only a small number[2] of final slots available, so even advancing to the finals is a great achievement in some regions.

In each ICPC contest, the teams have five hours of time to solve about ten algorithm problems. A solution to a problem is accepted only if it solves all test cases efficiently. During the contest, competitors may view the results of other teams, but for the last hour the scoreboard is frozen and it is not possible to see the results of the last submissions.

The topics that may appear at the ICPC are not so well specified as those at the IOI. In any case, it is clear that more knowledge is needed at the ICPC, especially more mathematical skills.

Online contests

There are also many online contests that are open for everybody. At the moment, the most active contest site is Codeforces, which organizes contests about weekly. In Codeforces, participants are divided into two divisions: beginners compete in Div2 and more experienced programmers in Div1. Other contest sites include AtCoder, CS Academy, HackerRank and Topcoder.

Some companies organize online contests with onsite finals. Examples of such contests are Facebook Hacker Cup, Google Code Jam and Yandex.Algorithm. Of course, companies also use those contests for recruiting: performing well in a contest is a good way to prove one's skills.

Books

There are already some books (besides this book) that focus on competitive programming and algorithmic problem solving:

- 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]The exact number of final slots varies from year to year; in 2017, there were 133 final slots.

The first two books are intended for beginners, whereas the last book contains advanced material.

Of course, general algorithm books are also suitable for competitive programmers. Some popular books are:

- 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]

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