Manual del Programador Competitivo

Antti Laaksonen

Traducción 24 de febrero de 2021

Índice general

Prefacio

El propósito de este libro es brindarle una sólida introducción a la programación competitiva. Se asume que el lector ya conoce los conceptos básicos de programación, pero no se necesita experiencia previa en la programación competitiva.

El libro está especialmente destinado a estudiantes que quieren aprender algoritmos y posiblemente participar en la Olimpiada Internacional de Informática (IOI) o en el Concurso Internacional de Programación Universitario (ICPC). Por supuesto, el libro también es adecuado para cualquier otra persona interesada en la programación competitiva.

Se necesita mucho tiempo para convertirse en un buen programador competitivo, pero también es una oportunidad para un gran aprendizaje. El lector puede estar seguro de que obtendrá una buena comprensión de los algoritmos si pasa tiempo leyendo el libro, resolviendo problemas y participando en concursos.

El libro está en continuo desarrollo. Puedes enviar comentarios sobre el libro al correo `ahslaaks@cs.helsinki.fi`.

Helsinki, Agosto 2019
Antti Laaksonen

Parte I

Técnicas básicas

Capítulo 1

Introducción

La programación competitiva combina dos asuntos: (1) el diseño de algoritmos y (2) la implementación de algoritmos.

El **diseño de algoritmos** consiste de la resolución de problemas y pensamiento matemático. Se necesitan habilidades para analizar y resolver problemas creativamente. Un algoritmo para resolver un problema tiene que ser correcto y eficiente, y a menudo la esencia del problema requiere la invención de un algoritmo eficiente.

El conocimiento teórico de los algoritmos es importante para los programadores competitivos. Normalmente, una solución a un problema es una combinación de técnicas conocidas y de nuevas ideas. Las técnicas que aparecen en la programación competitiva también forman la base para la investigación científica de algoritmos.

La **implementación de algoritmos** requiere de buenas habilidades en programación. En la programación competitiva, las soluciones son evaluadas utilizando un conjunto de casos de prueba sobre un algoritmo implementado. Por lo tanto, no es suficiente con que la idea del algoritmo sea correcta, la implementación también tiene que serlo.

En los concursos se utiliza un estilo de codificación sencillo y conciso. Los programas deben escribirse rápidamente, porque no hay mucho tiempo disponible. A diferencia de la ingeniería de software tradicional, los programas son cortos (cuando mucho unos cientos de líneas de código), y no necesitan mantenimiento después del concurso.

1.1. Lenguajes de programación

Al momento, los lenguajes de programación más populares usados en los concursos son C++, Python y Java. Por ejemplo, en el Google Code Jam 2017, entre los 3,000 mejores participantes, 79 % usaron C++, 16 % usaron Python y 8 % usaron Java [29]. Algunos participantes también utilizaron varios lenguajes de programación.

Mucha gente piensa que C++ es la mejor opción para un programador competitivo, y C++ casi siempre está disponible en todas las plataformas de concursos.

Los beneficios de usar C++ son que es un lenguaje muy eficiente y su biblioteca estándar contiene una gran colección de estructuras de datos y algoritmos.

Por otro lado, es bueno dominar varios lenguajes y comprender sus puntos fuertes. Por ejemplo, si se necesitan números enteros grandes en el problema, Python puede ser una buena opción, porque incorpora operaciones específicas para realizar cálculos con números enteros grandes. Aún así, la mayoría de los problemas en los concursos de programación están configurados para que el uso de un lenguaje de programación específico no sea una ventaja injusta.

Todos los programas de ejemplo de este libro están escritos en C++, y a menudo se utilizan las estructuras de datos y algoritmos de su biblioteca estándar. Los programas siguen el estándar C++11, que se puede utilizar en la mayoría de los concursos hoy en día. Si el lector aún no sabe programar en C++, ahora es un buen momento para empezar a aprender.

Plantilla de código de C++

La siguiente es una típica plantilla de C++ en programación competitiva:

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

using namespace std;

int main() {
    // aqui va la solución
}
```

La línea `#include` al inicio del código es una característica del compilador `g++` que nos permite incluir toda la biblioteca estándar. Por lo tanto, no es necesario incluir bibliotecas por separado tales como `iostream`, `vector` y `algorithm`, sino que están disponibles automáticamente.

La línea `using` declara que las clases y funciones de la biblioteca estándar se pueden utilizar directamente en el código. Sin la línea `using` tendríamos que escribir, por ejemplo, `std::cout`, pero ahora basta con escribir `cout`.

El código se puede compilar usando el siguiente comando:

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

Este comando produce el archivo binario `test` a partir del código fuente `test.cpp`. El compilador sigue el estándar C++11 (`-std=c++11`), optimiza el código (`-O2`) y muestra advertencias sobre posibles errores (`-Wall`).

1.2. Entrada y salida

En la mayoría de los concursos, los flujos estándar se utilizan para leer la entrada y escribir la salida. En C++, los flujos estándar son `cin` para entrada y `cout` para salida. Además, se pueden utilizar las funciones `scanf` y `printf` del lenguaje C.

La entrada del programa generalmente consiste en números y cadenas que están separados por espacios y nuevas líneas. Se pueden leer desde el flujo de entrada `cin` de la siguiente manera:

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

Este tipo de código siempre funciona, asumiendo que hay al menos un espacio o nueva línea entre cada elemento de la entrada. Por ejemplo, el código anterior puede leer las siguientes entradas de la misma manera:

```
123 456 monkey
```

```
123   456
monkey
```

El flujo `cout` se utiliza para la salida de la siguiente manera:

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

A veces, la entrada y la salida son un cuello de botella en el programa. Las siguientes líneas al inicio del programa hacen que la entrada y la salida sean más eficientes:

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

Se debe tener en cuenta que la nueva línea `"\n"` funciona más rápido que `endl`, porque `endl` siempre causa una operación de vaciado del búfer.

Las funciones de C `scanf` y `printf` son una alternativa a los flujos estándar de C++. Suelen ser un poco más rápidas pero también son más difíciles de usar. El siguiente código lee dos números enteros de la entrada:

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

El siguiente código imprime dos enteros:

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

A veces, el programa debe leer una línea completa de la entrada, posiblemente conteniendo espacios. Esto se puede lograr utilizando la función `getline`:

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

Si se desconoce la cantidad de datos, el siguiente ciclo es muy útil:

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

Este ciclo lee elementos de la entrada uno tras otro, hasta que no haya más datos disponibles en la entrada.

Algunas plataformas de concursos utilizan archivos para entrada y salida. Una solución fácil para esto es escribir el código como de costumbre usando flujos estándar, pero agregando las siguientes líneas al principio del código:

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

Después de esto, el programa lee la entrada del archivo "input.txt" y escribe la salida en el archivo "output.txt".

1.3. Trabajando con números

Enteros

El tipo de entero más utilizado en programación competitiva es `int`, el cual consta de 32 bits que proporcionan un rango de valores de $-2^{31} \ldots 2^{31}-1$ o cerca de $-2 \cdot 10^9 \ldots 2 \cdot 10^9$. Si el tipo `int` no es suficiente, se puede utilizar el tipo `long long` de 64 bits. Éste tiene un rango de valores de $-2^{63} \ldots 2^{63}-1$ o cerca de $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$.

El siguiente código define una variable `long long`:

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

El sufijo `LL` significa que el tipo de dato del número es `long long`.

Un error común cuando se usa el tipo `long long` es que el tipo `int` todavía se utilice en algún otro lugar. Por ejemplo, el siguiente código continue un error muy sutil:

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

Aunque la variable `b` es de tipo `long long`, ambos números en la expresión `a*a` son de tipo `int` y el resultado también es de tipo `int`. Debido a esto, la variable `b` contendrá un resultado equivocado. El problema se puede resolver cambiando el tipo de la variable `a` por `long long` o cambiando la expresión por `(long long)a*a`.

Por lo general, los problemas del concurso se establecen de manera que el tipo `long long` sea suficiente. Aun así, es bueno saber que el compilador `g++` también

provee un tipo `__int128_t` de 128 bits con un rango de valores de $-2^{127} \ldots 2^{127}-1$ o cerca de $-10^{38} \ldots 10^{38}$. Sin embargo, este tipo de dato no está disponible en todas las plataformas de concursos.

Aritmética modular

Denotamos a x mód m como el residuo cuando x es dividido por m. Por ejemplo, 17 mód $5 = 2$, porque $17 = 3 \cdot 5 + 2$.

A veces, la respuesta a un problema es un un número muy grande pero es suficiente imprimirlo "módulo m", es decir, el residuo cuando la respuesta es dividida por m (por ejemplo, "módulo $10^9 + 7$"). La idea es que incluso si la respuesta es muy grande, es suficiente utilizar los tipos `int` y `long long`.

Una propiedad importante del residuo es que en las operaciones de suma, resta y multiplicación, el residuo se puede realizar antes de la operación:

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

Por lo tanto, podemos aplicar el residuo después de cada operación y los números nunca serán demasiado grandes.

Por ejemplo, el siguiente código calcula $n!$, el factorial de n, módulo m:

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

Por lo general, queremos que el residuo siempre esté entre $0 \ldots m-1$. Sin embargo, en C++ y otros lenguajes, el residuo de un número negativo es cero o negativo. Una forma sencilla de asegurarse que no hay residuos negativos es primero calcular el residuo como siempre y sumarle m si el resultado es negativo:

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

Sin embargo, esto solo es necesario cuando hay restas en el código y el residuo puede volverse negativo.

Números de punto flotante

Los tipos de dato de punto flotante más comunes en programación competitiva son el `double` de 64 bits y, como una extensión en el compilador `g++`, el `long double` de 80 bits. En la mayoría de los casos, `double` es suficiente, pero `long double` es más preciso.

La precisión requerida de la respuesta generalmente se da en el enunciado del problema. Una forma sencilla de generar la respuesta es utilizar la función

`printf` y dar el número de lugares decimales en la cadena de formato. Por ejemplo, el siguiente código imprime el valor de x con 9 decimales:

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

Una dificultad al usar números de punto flotante es que algunos números no se pueden representar exactamente como tal, y habrá errores de redondeo. Por ejemplo, el resultado del siguiente código es sorprendente:

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

Debido a un error de redondeo, el valor de `x` es un poco menor que 1, mientras que el valor correcto sería 1.

Es arriesgado comparar números de punto flotante con el operador ==, porque es posible que aunque los valores son iguales al final de cuentas no lo sean debido a errores de precisión. Una mejor manera de comparar números de punto flotante es asumir que dos números son iguales si la diferencia entre ellos es menor que ε, donde ε es un número pequeño.

En la práctcia, los números se pueden comparar de la siguiente manera ($\varepsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a y b son iguales
}
```

Se debe tener en cuenta que aunque los números de punto flotante son inexactos, los números enteros todavía pueden ser representados con precisión hasta un cierto límite. Por ejemplo, usando `double`, es posible representar con precisión todos los enteros cuyo valor absoluto sea como máximo 2^{53}.

1.4. Abreviando el código

El código abreviado es ideal en la programación competitiva, porque los programas deben ser escritos tan rápido como sea posible. Debido a esto, los programadores competitivos a menudo definen nombres más cortos para tipos de datos y otras partes del código.

Nombres de tipo

Usando el comando `typedef` es posible dar un nombre más corto a un tipo de dato. Por ejemplo, el nombre `long long` es largo, así que podemos definir un nombre más corto `ll`:

```
typedef long long ll;
```

Después de esto, el código

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

se puede abreviar de la siguiente manera:

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

El comando `typedef` también se puede utilizar con tipos más complejos. Por ejemplo, el siguiente código establece el nombre `vi` para un vector de enteros y el nombre `pi` para un par que contiene dos enteros.

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

Macros

Otra forma de acortar el código es mediante **macros**. Una macro significa que ciertas cadenas en el código se cambiarán antes de la compilación. En C++, las macros se definen utilizando la palabra clave `#define`.

Por ejemplo, podemos definir las siguientes macros:

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

Después de esto, el código

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

puede abreviarse de la siguiente manera:

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

Una macro también puede tener parámetros que permiten abreviar ciclos y otras estructuras de control. Por ejemplo, podemos definir la siguiente macro:

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

Después de esto, el código

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

puede abreviarse de la siguiente manera:

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

A veces, las macros causan errores que pueden resultar difíciles detectar. Por ejemplo, considere la siguiente macro que calcula el cuadrado de un número:

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

Esta macro *no* siempre funciona como se espera. Por ejemplo, el código

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

corresponde al código

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

Una mejor versión de la macro es la siguiente:

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

Ahora el código

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

corresponde al código

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

1.5. Matemáticas

Las matemáticas tienen un papel importante en la programación competitiva, y no es posible convertirse en un programador competitivo exitoso sin tener buenas habilidades matemáticas. En esta sección se analizan algunos conceptos matemáticos y fórmulas que se necesitan más adelante en el libro.

Fórmulas de sumas

Cada suma de la forma

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

donde k es un enero positivo, tiene una fórmula de forma cerrada que es un polinomio de grado $k+1$. Por ejemplo[1],

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

y

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

Una **progresión aritmética** es una secuencia de números donde la diferencia entre dos números consecutivos es constante. Por ejemplo,

$$3,7,11,15$$

es una progresión aritmética con constante 4. Se puede calcular la suma de una progresión aritmética usando la fórmula

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

donde a es el primer número, b es el último número y n es la cantidad de números. Por ejemplo,

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

La fórmula se basa en el hecho de que la suma consta de n números y el valor de cada número es $(a+b)/2$ en promedio.

Una **progresión geométrica** es una secuencia de números donde la relación entre dos números consecutivos es constante. Por ejemplo,

$$3,6,12,24$$

es una progresión geométrica con constante 2. La suma de una progresión geométrica se puede calcular usando la fórmula

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

donde a es el primer número, b es el último número y la relación entre números consecutivos es k. Por ejemplo,

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

Esta fórmula se puede derivar de la siguiente manera. Sea

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

Al multiplicar ambos lados por k, obtenemos

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

[1] Incluso existe una fórmula general para tales sumas, llamada **fórmula de Faulhaber**, pero es demasiado compleja para presentarla aquí.

y resolviendo la ecuación

$$kS - S = bk - a$$

produce la fórmula.

Un caso especial de suma de una progresión geométrica es la fórmula

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

Una **suma armónica** es una suma de la forma

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

Un límite superior para una suma armónica es $\log_2(n) + 1$. Es decir, podemos modificar cada término $1/k$ de tal manera que k se convierte en la potencia más cercana de dos que no excede k. Por ejemplo, cuando $n = 6$, podemos estimar la suma como sigue:

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

Este límite superior consta de $\log_2(n) + 1$ partes (1, $2 \cdot 1/2$, $4 \cdot 1/4$, etc.), y el valor de cada parte es como máximo 1.

Teoría de conjuntos

Un **conjunto** es una colección de elementos. Por ejemplo, el conjunto

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

contiene los elementos 2, 4 y 7. El símbolo $\emptyset$ denota un conjunto vacío, y $|S|$ denota el tamaño de un conjunto S, es decir, el número de elementos en el conjunto. Por ejemplo, en el conjunto anterior, $|X| = 3$.

Si un conjunto S contiene un elemento x, escribimos $x \in S$, de lo contrario escribimos $x \notin S$. Por ejemplo, en el conjunto anterior

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

Se pueden construir nuevos conjuntos usando operaciones de conjuntos:

- La **intersección** $A \cap B$ consiste de elementos que están en ambos A y B. Por ejemplo, si $A = \{1, 2, 5\}$ y $B = \{2, 4\}$, entonces $A \cap B = \{2\}$.
- La **unión** $A \cup B$ consiste de elementos que están en A o en B o en ambos. Por ejemplo, si $A = \{3, 7\}$ and $B = \{2, 3, 8\}$, entonces $A \cup B = \{2, 3, 7, 8\}$.
- El **complemento** $\bar{A}$ consiste de elementos que no están en A. La interpretación de un complemento depende de el **conjunto universal**, que contiene todos los elementos posibles. Por ejemplo, si $A = \{1, 2, 5, 7\}$ y el conjunto universal es $\{1, 2, \ldots, 10\}$, entonces $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- La **diferencia** $A \setminus B = A \cap \bar{B}$ consiste de elementos que están en A pero no en B. Tenga en cuenta que B puede contener elementos que no están en A. Por ejemplo, si $A = \{2, 3, 7, 8\}$ y $B = \{3, 5, 8\}$, entonces $A \setminus B = \{2, 7\}$.

Si cada elemento de A también pertenece a S, podemos decir que A es un **subconjunto** de S, denotado por $A \subset S$. Un conjunto S siempre tiene $2^{|S|}$ subcon-

juntos, incluyendo el conjunto vacío. Por ejemplo, los subconjuntos del conjunto $\{2,4,7\}$ son

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

Algunos conjuntos de uso frecuente son $\mathbb{N}$ (números naturales), $\mathbb{Z}$ (enteros), $\mathbb{Q}$ (números racionales) y $\mathbb{R}$ (números reales). El conjunto $\mathbb{N}$ se puede definir de dos formas, dependiendo sobre la situación: cualquiera de $\mathbb{N} = \{0,1,2,\ldots\}$ o $\mathbb{N} = \{1,2,3,...\}$.

También podemos construir un conjunto usando una regla de la forma

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

donde $f(n)$ es alguna función. Este conjunto contiene todos los elementos de la forma $f(n)$, donde n es un elemento en S. Por ejemplo, el conjunto

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

contiene todos los enteros pares

Lógica

El valor de una expresión lógica es **true** (1) o **false** (0). Los operadores lógicos más importantes son $\neg$ (**negación**), $\wedge$ (**conjunción**), $\vee$ (**disyunción**), $\Rightarrow$ (**implicación**) y $\Leftrightarrow$ (**equivalencia**). La siguiente tabla muestra los significados de estos 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

La expresión $\neg A$ tiene el valor opuesto de A. La expresión $A \wedge B$ es verdadera si ambos A y B son verdaderos, y la expresión $A \vee B$ es verdadera si A o B o ambos son verdaderos. La expresión $A \Rightarrow B$ es verdadera si cuando A es verdadera, también B es verdadera. La expresión $A \Leftrightarrow B$ es verdadera si A y B son ambas verdaderas o ambas falsas.

Un **predicado** es una expresión que es verdadera o falsa dependiendo de sus parámetros. Los predicados generalmente se denotan con letras mayúsculas. Por ejemplo, podemos definir un predicado $P(x)$ que sea verdadero exactamente cuando x sea un número primo. Usando esta definición, $P(7)$ es verdadero pero $P(8)$ es falso.

Un **cuantificador** conecta una expresión lógica a los elementos de un conjunto. Los cuantificadores más importantes son $\forall$ (**para todo**) y $\exists$ (**existe**). Por ejemplo,

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

significa que por cada elemento x en el conjunto, existe un elemento y en el conjunto tal que y es más pequeño que x. Esto es cierto en el conjunto de números enteros, pero falso en el conjunto de los números naturales.

Usando la notación descrita anteriormente, podemos expresar muchos tipos de proposiciones lógicas. Por ejemplo,

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

significa que si un número x es mayor que 1 y no es un número primo, entonces existen números a y b que son mayores que 1 y cuyo producto es x. Esta proposición es verdadera en el conjunto de números enteros.

Funciones

La función $\lfloor x \rfloor$ redondea el número x hacia el entero inmediato inferior, y la función $\lceil x \rceil$ redondea el número x hacia el entero inmediato superior. Por ejemplo,

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

Las funciones $\text{mín}(x_1, x_2, \ldots, x_n)$ y $\text{máx}(x_1, x_2, \ldots, x_n)$ dan el más pequeño y más grande de los valores $x_1, x_2, \ldots, x_n$. Por ejemplo,

$$\text{mín}(1,2,3) = 1 \quad \text{y} \quad \text{máx}(1,2,3) = 3.$$

El **factorial** $n!$ se puede definir como

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

o recursivamente

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

Los **números de Fibonacci** surgen en muchas situaciones. Estos números pueden definirse recursivamente de la siguiente manera:

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

Los primeros números de Fibonacci son

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

También hay una fórmula de forma cerrada para calcular los números de Fibonacci, que a veces se llama **fórmula de Binet**:

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

Logaritmos

El **logaritmo** de un número x se denota $\log_k(x)$, donde k es la base del logaritmo. Según la definición, $\log_k(x) = a$ exactamente cuando $k^a = x$.

Una propiedad útil de los logaritmos es que $\log_k(x)$ es igual al número de veces que se debe dividir x por k antes de obtener el número 1. Por ejemplo, $\log_2(32) = 5$ porque se requieren 5 divisiones por 2:

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

Los logaritmos se utilizan a menudo en el análisis de algoritmos, porque muchos algoritmos eficientes reducen algo a la mitad en cada paso. Por tanto, podemos estimar la eficiencia de tales algoritmos usando logaritmos.

El logaritmo de un producto es

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

y consecuentemente,

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

Además, el logaritmo de un cociente es

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

Otra fórmula útil es

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

y usando esto, es posible calcular logaritmos a cualquier base si hay una manera de calcular logaritmos a una base fija.

El **logaritmo natural** $\ln(x)$ de un número x es un logaritmo cuya base es $e \approx 2.71828$. Otra propiedad de los logaritmos es que el número de dígitos de un entero x en base b es $\lfloor \log_b(x) + 1 \rfloor$. Por ejemplo, la representación de 123 en base 2 es 1111011 y $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6. Concursos y recursos

IOI

La Olimpiada Internacional de Informática (IOI por sus siglas en inglés) es un concurso de programación anual para estudiantes de escuela secundaria y preparatoria. Cada país puede enviar un equipo de cuatro estudiantes al concurso. Suele haber unos 300 participantes de 80 países.

La IOI consta de dos concursos de cinco horas de duración. En ambos concursos, los participantes tienen que resolver tres tareas algorítmicas de diversa dificultad. Las tareas se dividen en subtareas, cada uno de los cuales tiene una puntuación asignada. Incluso si los concursantes se dividen en equipos, compiten de manera individual.

El programa de estudios del IOI [41] establece los temas que pueden aparecer en las tareas. Casi todos los temas del programa de estudios del IOI están cubiertos por este libro.

Los participantes del IOI se seleccionan mediante concursos nacionales. Antes del IOI, se organizan muchos concursos regionales, como la Olimpiada Báltica de Informática (BOI), la Olimpiada Centroeuropea de Informática (CEOI) y la Olimpiada de Informática de Asia y el Pacífico (APIO).

Algunos países organizan concursos de práctica en línea para los futuros participantes de la IOI, tales como el Concurso Abierto Croata de Informática [11] y la Olimpiada de Computación de Estados Unidos [68]. Además, una gran colección de problemas de concursos polacos está disponible en línea [60].

ICPC

El Concurso Internacional de Programación Universitaria (ICPC por sus siglas en inglés) es un concurso anual de programación para estudiantes universitarios. Cada equipo consta de tres estudiantes, y a diferencia del IOI, los estudiantes trabajan juntos; solo hay una computadora disponible para cada equipo.

El ICPC consta de varias etapas, y los mejores equipos son invitados a la Final Mundial. Si bien hay decenas de miles de participantes en el concurso, solo hay un pequeño número [2] de lugares disponibles en la final, así que incluso avanzar a la final ya es un gran logro en algunas regiones del mundo.

En cada concurso del ICPC, los equipos tienen cinco horas para resolver alrededor de diez problemas de algoritmos. Se acepta una solución a un problema solo si se resuelven todos los casos de prueba de manera eficiente. Durante el concurso, los competidores pueden ver los resultados de otros equipos, pero durante la última hora el marcador está congelado y no es posible ver los resultados de los últimos envíos.

Los temas que pueden aparecer en el ICPC no están tan bien especificados como los del IOI. En cualquier caso, está claro que en el ICPC se necesitan más conocimientos, especialmente más habilidades matemáticas.

Concursos en línea

También hay muchos concursos en línea que están al alcance de todos. Por el momento, el sitio de concursos más activo es Codeforces, que organiza concursos cada semana. En Codeforces, los participantes se dividen en dos divisiones: los principiantes compiten en la división Div2 y los más experimentados en la división Div1. Otros sitios de concursos incluyen AtCoder, CS Academy, HackerRank y Topcoder.

Algunas empresas organizan concursos en línea con finales presenciales. Ejemplos de tales concursos son Facebook Hacker Cup, Google Code Jam y Yandex.Algorithm. Por supuesto, las empresas también utilizan esos concursos

[2] El número exacto de lugares disponibles puede variar de un año a otro; en 2017, hubo 133 lugares en la final.

para reclutar: obtener un buen desempeño en un concurso es una buena manera de demostrar sus habilidades.

Libros

Hay algunos libros (además de este) que se enfocan en la programación competitiva y la resolución algorítmica de problemas:

- S. S. Skiena y M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim y 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]

Los dos primeros libros están destinados a principiantes, mientras que el último libro contiene material avanzado.

Por supuesto, los libros de algoritmos en general también son adecuados para programadores competitivos. Algunos libros populares son:

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

Capítulo 2

Complejidad temporal

La eficiencia de los algoritmos es importante en la programación competitiva. Por lo general, es fácil diseñar un algoritmo. que resuelve el problema lentamente, pero el verdadero desafío es inventar un algoritmo rápido. Si el algoritmo es demasiado lento, solo obtendrá puntos parciales o ningún punto.

La **complejidad temporal** de un algoritmo estima cuánto tiempo utilizará el algoritmo para alguna entrada. La idea es representar la eficiencia como una función cuyo parámetro es el tamaño de la entrada. Al calcular la complejidad temporal se puede averiguar si el algoritmo es lo suficientemente rápido sin necesidad de implementarlo.

2.1. Reglas de cálculo

La complejidad temporal de un algoritmo se denota $O(\cdots)$ donde los tres puntos representan alguna función. Por lo general, la variable n denota el tamaño de entrada. Por ejemplo, si la entrada es una matriz de números, n será el tamaño de la matriz, y si la entrada es una cadena, n será la longitud de la cadena.

Ciclos

Una razón común por la que un algoritmo es lento se debe a que contiene muchos ciclos que pasan por la entrada. Entre más ciclos anidados contenga el algoritmo, más lento será. Si hay k ciclos anidados, la complejidad temporal es $O(n^k)$.

Por ejemplo, la complejidad temporal del siguiente código es $O(n)$:

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

Y la complejidad temporal del siguiente código es $O(n^2)$:

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

```
    }
}
```

Orden de magnitud

Una complejidad temporal no especifica el número exacto de veces que se ejecuta el código dentro de un ciclo, solamente indica el orden de magnitud. En los siguientes ejemplos, el código dentro del ciclo se ejecuta $3n$, $n+5$ y $\lceil n/2 \rceil$ veces, pero la complejidad temporal de cada código es $O(n)$.

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

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

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

Como otro ejemplo, la complejidad temporal del siguiente código es $O(n^2)$:

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

Fases

Si el algoritmo consta de fases consecutivas, la complejidad temporal total es la mayor complejidad temporal de una sola fase. La razón de esto es que la fase más lenta suele ser el cuello de botella del código.

Por ejemplo, el siguiente código consta de tres fases con complejidades temporales $O(n)$, $O(n^2)$ y $O(n)$. Por tanto, la complejidad temporal total es $O(n^2)$.

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

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

Varias variables

A veces, la complejidad temporal depende de varios factores. En este caso, la fórmula de complejidad temporal contiene varias variables.

Por ejemplo, la complejidad temporal del el siguiente código es $O(nm)$:

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

Recursividad

La complejidad temporal de una función recursiva depende del número de veces que se llama a la función y de la complejidad temporal de una sola llamada. La complejidad temporal total es el producto de estos valores.

Por ejemplo, considere la siguiente función:

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

La llamada `f`(n) realiza n llamadas a la función, y la complejidad temporal de cada llamada es $O(1)$. Por lo tanto, la complejidad temporal total es $O(n)$.

Como otro ejemplo, considere la siguiente función:

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

En este caso cada llamada a la función genera otras dos llamadas, excepto cuando $n = 1$. Veamos qué sucede cuando se llama a la función g con el parámetro n. La siguiente tabla muestra las llamadas a función producidas por esta única llamada:

llamada a función	número de llamadas
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
$\cdots$	$\cdots$
$g(1)$	2^{n-1}

Basado en esto, la complejidad temporal es

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

2.2. Clases de complejidad

La siguiente lista contiene las complejidades temporales más comunes en algoritmos:

$O(1)$ El tiempo de ejecución en un algoritmo de **tiempo constante** no depende del tamaño de la entrada. Un algoritmo típico de tiempo constante es una fórmula que calcula una respuesta.

$O(\log n)$ Una algoritmo **logarítmico** a menudo divide el tamaño de la entrada a la mitad en cada paso. El tiempo de ejecución de tal algoritmo es logarítmico, porque $\log_2 n$ es igual al número de veces que n se debe dividir por 2 para obtener 1.

$O(\sqrt{n})$ Un **algoritmo de raíz cuadrada** es más lento que $O(\log n)$ pero más rápido que $O(n)$. Una propiedad especial de las raíces cuadradas es que $\sqrt{n} = n/\sqrt{n}$, de tal modo que la raiz cuadrada $\sqrt{n}$ se encuentra, en cierto sentido, en medio de la entrada.

$O(n)$ Un algoritmo **lineal** recorre la entrada un número constante de veces. Esta suele ser la mejor complejidad de tiempo posible, porque suele ser necesario acceder a cada elemento de entrada al menos una vez antes de producir la respuesta.

$O(n \log n)$ Esta complejidad temporal a menudo indica que el algoritmo ordena la entrada, porque la complejidad temporal de los algoritmos de ordenación eficientes es $O(n \log n)$. Otra posibilidad es que el algoritmo utiliza una estructura de datos donde cada operación toma un tiempo $O(\log n)$.

$O(n^2)$ Un algoritmo **cuadrático** a menudo contiene dos ciclos anidados. Es posible recorrer todos los pares de elementos de entrada en un tiempo $O(n^2)$.

$O(n^3)$ Un algoritmo **cúbico** a menudo contiene tres ciclos anidados. Es posible recorrer todos las tripletas de elementos de entrada en un tiempo $O(n^3)$.

$O(2^n)$ Esta complejidad temporal a menudo indica que el algoritmo itera a través de todos subconjuntos de los elementos de entrada. Por ejemplo, los subconjuntos de $\{1,2,3\}$ son $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ y $\{1,2,3\}$.

$O(n!)$ Esta complejidad temporal a menudo indica que el algoritmo itera a través de todas las permutaciones de los elementos de entrada. Por ejemplos, las permutaciones de $\{1,2,3\}$ son $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ y $(3,2,1)$.

Un algoritmo es **polinomial** si su complejidad temporal es como mucho $O(n^k)$ donde k es una constante. Todas las complejidades temporales anteriores excepto $O(2^n)$ y $O(n!)$ son polinomiales. En la práctica, la constante k suele ser pequeña, y por lo tanto una complejidad temporal polinomial aproximadamente significa que el algoritmo es *eficiente*.

La mayoría de los algoritmos de este libro son polinomiales. Sin embargo, existen muchos problemas importantes para los que no se conoce ningún algoritmo polinomial, es decir, nadie sabe cómo resolverlos de manera eficiente. Los problemas **NP-difícil** son un conjunto importante de problemas, para los cuales no se conoce un algoritmo polinomial[1].

2.3. Estimación de la eficiencia

Al calcular la complejidad temporal de un algoritmo, es posible comprobar que el algoritmo sea lo suficientemente eficiente para el problema antes de implementarlo. El punto de partida para las estimación es el hecho de que una computadora moderna puede realizar cientos de millones de operaciones en un segundo.

Por ejemplo, suponga que el límite de tiempo para un problema es de un segundo y el tamaño de la entrada es $n = 10^5$. Si la complejidad temporal es $O(n^2)$, el algoritmo realizará alrededor de $(10^5)^2 = 10^{10}$ operaciones. Esto debería llevar al menos algunas decenas de segundos, por lo que el algoritmo parece ser demasiado lento para resolver el problema.

Por otro lado, dado el tamaño de entrada, podemos intentar *adivinar* la complejidad de tiempo requerida del algoritmo que resuelve el problema. La siguiente tabla contiene algunas estimaciones útiles asumiendo un límite de tiempo de un segundo.

tamaño de la entrada	complejidad temporal requerido
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ o $O(n)$
n es grande	$O(1)$ o $O(\log n)$

Por ejemplo, si el tamaño de la entrada es $n = 10^5$, probablemente se espera que la complejidad temporal del algoritmo sea $O(n)$ o $O(n \log n)$. Esta información

[1]Un libro clásico sobre el tema es M. R. Garey's y D. S. Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* [28].

facilita el diseño del algoritmo, porque descarta enfoques que producirían un algoritmo con una complejidad temporal peor.

Aún así, es importante recordar que la complejidad temporal solamente es una estimación de la eficiencia, porque oculta los *factores constantes*. Por ejemplo, un algoritmo que se ejecuta en tiempo $O(n)$ puede realizar $n/2$ o $5n$ operaciones. Esto tiene un efecto importante en el tiempo de ejecución del algoritmo.

2.4. Suma máxima del subarreglo

A menudo hay diferentes algoritmos para resolver un mismo problema, de tal manera que tienen complejidades temporales diferentes. Esta sección analiza un problema clásico que tiene una sencilla solución $O(n^3)$. Sin embargo, al diseñar un mejor algoritmo, es posible resolver el problema en un tiempo $O(n^2)$ e inclusive en un tiempo $O(n)$.

Dado un arreglo de n números, la tarea es calcular la **suma máxima del subarreglo**[2], es decir, la mayor suma posible de una secuencia de valores consecutivos en el arreglo. El problema es interesante cuando puede haber valores negativos en el arreglo. Por ejemplo, en el arreglo

-1	2	4	-3	5	2	-5	2

el siguiente subarreglo produce la suma máxima 10:

-1	2	4	-3	5	2	-5	2

Suponemos que se permite un subarreglo vacío, por lo que la suma máxima del subarreglo es siempre al menos 0.

Algoritmo 1

Una forma sencilla de resolver el problema es recorrer todos los subarreglos posibles, calcular la suma de valores en cada subarreglo y mantener la suma máxima. El siguiente código implementa este 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] El libro *Programming Pearls* [8] de J. Bentley popularizó este problema.

Las variables a y b establecen el primer y último índice del subarreglo, y la suma de valores se calcula en la variable sum. La variable best contiene la suma máxima encontrada durante la búsqueda.

La complejidad temporal de este algoritmo es $O(n^3)$, porque contiene tres ciclos anidados que recorren la entrada.

Algoritmo 2

Es fácil hacer que el algoritmo 1 sea más eficiente quitando un ciclo de él. Esto es posible calculando la suma en el mismo momento en el que se mueve el extremo derecho del subarreglo. El resultado es el siguiente 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";
```

Después de este cambio, la complejidad temporal es $O(n^2)$.

Algoritmo 3

Sorprendentemente, es posible resolver el problema en tiempo $O(n)$ [3], lo cual significa que solamente con un ciclo es suficiente. La idea es calcular, para cada posición del arreglo, la suma máxima de un subarreglo que termina en esa posición. Después de esto, la respuesta al problema es la máximo de esas sumas.

Considere el subproblema de encontrar el subarreglo de suma máxima que termina en la posición k. Hay dos posibilidades:

1. El subarreglo solo contiene el elemento en la posición k.
2. El subarreglo consta de un subarreglo que termina en la posición $k-1$, seguido del elemento en la posición k.

En este último caso, ya que queremos encontrar un subarreglo con suma máxima, el subarreglo que termina en la posición $k-1$ también debe tener la suma máxima. Por lo tanto, podemos resolver el problema de manera eficiente calculando la suma máxima del subarreglo para cada posición final de izquierda a derecha.

El siguiente código implementa el algoritmo:

[3] En [8], este algoritmo lineal es atribuído a J. B. Kadane, y el algoritmo es a veces llamado **algoritmo de Kadane**.

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

El algoritmo solamente contiene un ciclo que recorre la entrada, por lo que la complejidad temporal es $O(n)$. Esta también es la mejor complejidad temporal posible, porque cualquier algoritmo para el problema tiene que examinar todos los elementos del arreglo al menos una vez.

Comparación de eficiencia

Es interesante estudiar cuán eficiente son los algoritmos en la práctica. La siguiente tabla muestra los tiempos de ejecución de los algoritmos anteriores para diferentes valores de n en una computadora moderna.

En cada prueba, la entrada se generó de forma aleatoria. El tiempo necesario para leer la entrada no fue medido.

arreglo de tamaño 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

La comparación muestra que todos los algoritmos son eficientes cuando el tamaño de entrada es pequeño, pero las entradas más grandes resaltan diferencias en los tiempos de ejecución. El algoritmo 1 se vuelve lento cuando $n = 10^4$, y el algoritmo 2 se vuelve lento cuando $n = 10^5$. Solamente el algoritmo 3 puede procesar casi instantáneamente incluso las entradas más grandes.

Bibliografía

[1] A. V. Aho, J. E. Hopcroft y J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] R. K. Ahuja y 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 y 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 y K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.

[7] M. A. Bender y 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 y 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 y 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 y 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 y 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 y A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.

[20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.

[21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.

[23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.

[25] L. R. Ford y 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 y 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 y 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 y F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.

[34] M. Held y 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 y 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 y J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.

[39] E. Horowitz y 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 y 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 y M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.

[45] J. Kleinberg y 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 y 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 y 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 y 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 y 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 y 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 y 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 alfabético