

# Competitive Programmer's Handbook

Traducido al español

Antti Laaksonen

Creado August 19, 2019

Traducido June 20, 2024



# Contents

<b>Prefacio</b>	<b>ix</b>
<b>I Técnicas básicas</b>	<b>1</b>
<b>1 Introducción</b>	<b>3</b>
1.1 Lenguajes de programación . . . . .	3
1.2 Input y output . . . . .	4
1.3 Trabajando con números . . . . .	6
1.4 Acortando código . . . . .	8
1.5 Matemáticas . . . . .	10
1.6 Concursos y recursos . . . . .	15
<b>2 Complejidad temporal</b>	<b>19</b>
2.1 Reglas de cálculo . . . . .	19
2.2 Clases de complejidad . . . . .	22
2.3 Estimación de eficiencia . . . . .	23
2.4 Suma máxima de subarreglo . . . . .	24
<b>3 Ordenamiento</b>	<b>27</b>
3.1 Teoría de ordenamiento . . . . .	27
3.2 Ordenación en C++ . . . . .	32
3.3 Búsqueda binaria . . . . .	34
<b>4 Estructuras de datos</b>	<b>39</b>
4.1 Arreglos dinámicos . . . . .	39
4.2 Estructuras de conjunto . . . . .	41
4.3 Estructuras de mapa . . . . .	42
4.4 Iteradores y rangos . . . . .	43
4.5 Otras estructuras . . . . .	46
4.6 Comparación con la clasificación . . . . .	50
<b>5 Búsqueda completa</b>	<b>53</b>
5.1 Generación de subconjuntos . . . . .	53
5.2 Generación de permutaciones . . . . .	55
5.3 Backtracking . . . . .	56
5.4 Poda de la búsqueda . . . . .	58
5.5 Encuentro en el medio . . . . .	61

<b>6 Algoritmos voraces</b>	<b>63</b>
6.1 Problema de la moneda . . . . .	63
6.2 Planificación . . . . .	64
6.3 Tareas y plazos . . . . .	66
6.4 Minimizar sumas . . . . .	67
6.5 Compresión de datos . . . . .	68
<b>7 Programación dinámica</b>	<b>71</b>
7.1 Problema de la moneda . . . . .	71
7.2 Subsecuencia creciente más larga . . . . .	76
7.3 Caminos en una cuadrícula . . . . .	77
7.4 Problemas de la mochila . . . . .	79
7.5 Distancia de edición . . . . .	80
7.6 Contando mosaicos . . . . .	82
<b>8 Análisis amortizado</b>	<b>85</b>
8.1 Método de dos punteros . . . . .	85
8.2 Elementos más pequeños cercanos . . . . .	87
8.3 Mínimo de ventana deslizante . . . . .	89
<b>9 Consultas de rango</b>	<b>91</b>
9.1 Consultas de matriz estática . . . . .	92
9.2 Árbol indexado binario . . . . .	94
9.3 Árbol de segmentos . . . . .	97
9.4 Técnicas adicionales . . . . .	101
<b>10 Manipulación de bits</b>	<b>103</b>
10.1 Representación de bits . . . . .	103
10.2 Operaciones de bits . . . . .	104
10.3 Representar conjuntos . . . . .	106
10.4 Optimizaciones de bits . . . . .	108
10.5 Programación dinámica . . . . .	110
<b>II Algoritmos gráficos</b>	<b>117</b>
<b>11 Conceptos básicos de grafos</b>	<b>119</b>
11.1 Terminología de grafos . . . . .	119
11.2 Representación de grafos . . . . .	123
<b>12 Recorrido de grafos</b>	<b>127</b>
12.1 Búsqueda en profundidad . . . . .	127
12.2 Búsqueda en amplitud . . . . .	129
12.3 Aplicaciones . . . . .	131

<b>13 Caminos más cortos</b>	<b>135</b>
13.1 Algoritmo de Bellman–Ford . . . . .	135
13.2 Algoritmo de Dijkstra . . . . .	138
13.3 Algoritmo de Floyd–Warshall . . . . .	141
<b>14 Algoritmos de árbol</b>	<b>145</b>
14.1 Recorrido de árbol . . . . .	146
14.2 Diámetro . . . . .	147
14.3 Todos los caminos más largos . . . . .	149
14.4 Árboles binarios . . . . .	151
<b>15 Árboles de expansión</b>	<b>153</b>
15.1 Algoritmo de Kruskal . . . . .	154
15.2 Estructura de unión-búsqueda . . . . .	157
15.3 Algoritmo de Prim . . . . .	159
<b>16 Grafos dirigidos</b>	<b>163</b>
16.1 Ordenación topológica . . . . .	163
16.2 Programación dinámica . . . . .	165
16.3 Caminos sucesores . . . . .	168
16.4 Detección de ciclos . . . . .	169
<b>17 Conectividad fuerte</b>	<b>173</b>
17.1 Algoritmo de Kosaraju . . . . .	174
17.2 Problema 2SAT . . . . .	176
<b>18 Consultas de árboles</b>	<b>179</b>
18.1 Encontrar ancestros . . . . .	179
18.2 Subárboles y caminos . . . . .	180
18.3 Antepasado común más bajo . . . . .	183
18.4 Algoritmos fuera de línea . . . . .	186
<b>19 Caminos y circuitos</b>	<b>191</b>
19.1 Caminos Eulerianos . . . . .	191
19.2 Caminos hamiltonianos . . . . .	195
19.3 Secuencias de De Bruijn . . . . .	196
19.4 Recorridos del caballo . . . . .	197
<b>20 Flujos y cortes</b>	<b>199</b>
20.1 Algoritmo de Ford–Fulkerson . . . . .	200
20.2 Caminos disjuntos . . . . .	204
20.3 Emparejamientos máximos . . . . .	205
20.4 Coberturas de caminos . . . . .	209

<b>III Temas avanzados</b>	<b>213</b>
<b>21 Teoría de números</b>	<b>215</b>
21.1 Primos y factores . . . . .	215
21.2 Aritmética modular . . . . .	219
21.3 Resolver ecuaciones . . . . .	222
21.4 Otros resultados . . . . .	223
<b>22 Combinatoria</b>	<b>225</b>
22.1 Coeficientes binomiales . . . . .	226
22.2 Números de Catalan . . . . .	229
22.3 Inclusión-exclusión . . . . .	231
22.4 Lema de Burnside . . . . .	232
22.5 Fórmula de Cayley . . . . .	233
<b>23 Matrices</b>	<b>237</b>
23.1 Operaciones . . . . .	237
23.2 Recurrencias lineales . . . . .	240
23.3 Gráficos y matrices . . . . .	242
<b>24 Probabilidad</b>	<b>245</b>
24.1 Cálculo . . . . .	245
24.2 Eventos . . . . .	246
24.3 Variables aleatorias . . . . .	248
24.4 Cadenas de Markov . . . . .	250
24.5 Algoritmos aleatorios . . . . .	251
<b>25 Teoría de juegos</b>	<b>255</b>
25.1 Estados del juego . . . . .	255
25.2 Juego de Nim . . . . .	257
25.3 Teorema de Sprague–Grundy . . . . .	258
<b>26 Algoritmos de cadenas</b>	<b>263</b>
26.1 Terminología de cadenas . . . . .	263
26.2 Estructura de trie . . . . .	264
26.3 Hashing de cadenas . . . . .	265
26.4 Algoritmo Z . . . . .	268
<b>27 Algoritmos de raíz cuadrada</b>	<b>273</b>
27.1 Combinación de algoritmos . . . . .	274
27.2 Particiones de enteros . . . . .	276
27.3 Algoritmo de Mo . . . . .	277
<b>28 Árboles de segmentos revisitados</b>	<b>281</b>
28.1 Propagación perezosa . . . . .	282
28.2 Árboles dinámicos . . . . .	285
28.3 Estructuras de datos . . . . .	287
28.4 Bidimensionalidad . . . . .	288

<b>29 Geometría</b>	<b>291</b>
29.1 Números complejos . . . . .	292
29.2 Puntos y líneas . . . . .	294
29.3 Área del polígono . . . . .	297
29.4 Funciones de distancia . . . . .	299
<b>30 Algoritmos de línea de barrido</b>	<b>303</b>
30.1 Puntos de intersección . . . . .	304
30.2 Problema del par más cercano . . . . .	305
30.3 Problema de la envolvente convexa . . . . .	306
<b>Bibliography</b>	<b>309</b>
<b>Index</b>	<b>315</b>





# Prólogo

El propósito de este libro es darte una introducción completa a la programación competitiva. Se asume que ya dominas lo básico de programación, pero no es necesario que poseas conocimientos previos de programación competitiva.

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

Toma bastante tiempo convertirse en un buen concursante de programación, sin embargo también es una oportunidad de aprender muchísimo. Puedes estar seguro que obtendrás un buen entendimiento general sobre los algoritmos si le dedicas tiempo a la lectura de este libro, resolviendo problemas y compitiendo en muchos concursos.

El libro está en continuo desarrollo. Siempre puedes enviar comentarios sobre el mismo a `ahslaaks@cs.helsinki.fi`<sup>1</sup>.

Helsinki, March 2017  
Antti Laaksonen

---

<sup>1</sup>Escriba a ese correo solo en caso de que sean comentarios sobre el libro original en inglés. En caso de que sean sobre esta traducción visite la url: <https://github.com/zlarosav/cphb-es>.



# **Part I**

## **Técnicas básicas**



# Chapter 1

## Introducción

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

El **diseño de algoritmos** consiste en la resolución de problemas y el pensamiento matemático. Se necesitan habilidades para analizar problemas y resolverlos creativamente. Un algoritmo para resolver un problema debe ser tanto correcto como eficiente, y el núcleo del problema suele tratar sobre la invención de un algoritmo eficiente.

El conocimiento teórico de algoritmos es importante para los programadores competitivos. Típicamente, una solución a un problema es una combinación de técnicas bien conocidas y 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 buenas habilidades de programación. En la programación competitiva, las soluciones se evalúan probando un algoritmo implementado utilizando un conjunto de casos de prueba. Por lo tanto, no basta con que la idea del algoritmo sea correcta, sino que la implementación también debe ser correcta.

Un buen estilo de codificación en concursos es directo 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 (por lo general, como máximo unas pocas cientos de líneas de código), y no necesitan mantenerse después del concurso.

### 1.1 Lenguajes de programación

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

Mucha gente piensa que C++ es la mejor opción para un programador competitivo, y C++ está casi siempre disponible en los sistemas de concurso. 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 entender sus fortalezas. Por ejemplo, si se necesitan números enteros grandes en el problema, Python puede ser una buena elección, porque contiene operaciones integradas para calcular con números enteros grandes. Aun así, la mayoría de los problemas en concursos de programación están diseñados de modo que usar un lenguaje de programación específico no sea una ventaja injusta.

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

## Plantilla de código en C++

Una plantilla típica de código en C++ para programación competitiva se ve así:

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

using namespace std;

int main() {
    // la solucion viene aqui
}
```

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

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

El código puede compilarse usando el siguiente comando:

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

Este comando produce un 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 Input y output

En la mayoría de los concursos, se usan flujos estándar para leer input y escribir output. En C++, los flujos estándar son `cin` para input y `cout` para output. Además, se pueden usar las funciones de C `scanf` y `printf`.

El input para el programa usualmente consiste en números y cadenas que están separados por espacios y saltos de línea. Se pueden leer del flujo `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 salto de línea entre cada elemento en el input. Por ejemplo, el código anterior puede leer cualquiera de los siguientes inputs:

```
123 456 monkey
```

```
123    456  
monkey
```

El flujo cout se usa para el output de la siguiente manera:

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

El input y el output a veces son un cuello de botella en el programa. Las siguientes líneas al principio del código hacen que el input y el output sean más eficientes:

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

Ten en cuenta que el salto de línea "\n" funciona más rápido que endl, porque endl siempre provoca una operación de vaciado del buffer.

Las funciones de C scanf y printf son una alternativa a los flujos estándar de C++. Usualmente son un poco más rápidas, pero también son más difíciles de usar. El siguiente código lee dos enteros del input:

```
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 del input, posiblemente conteniendo espacios. Esto se puede lograr usando la función getline:

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

Si la cantidad de datos es desconocida, el siguiente bucle es útil:

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

Este bucle lee elementos del input uno tras otro, hasta que no haya más datos disponibles en el input.

En algunos sistemas de concursos, se usan archivos para el input y el output. Una solución fácil para esto es escribir el código como de costumbre usando flujos estándar, pero agregar las siguientes líneas al principio del código:

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

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

## 1.3 Trabajando con números

### Enteros

El tipo de entero más usado en la programación competitiva es `int`, que es un tipo de 32 bits con un rango de valores de  $-2^{31} \dots 2^{31} - 1$  o aproximadamente  $-2 \cdot 10^9 \dots 2 \cdot 10^9$ . Si el tipo `int` no es suficiente, se puede usar el tipo de 64 bits `long long`. Tiene un rango de valores de  $-2^{63} \dots 2^{63} - 1$  o aproximadamente  $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$ .

El siguiente código define una variable `long long`:

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

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

Un error común al usar el tipo `long long` es que el tipo `int` aún se usa en alguna parte del código. Por ejemplo, el siguiente código contiene un error sutil:

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

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

Usualmente los problemas de concursos están configurados de manera que el tipo `long long` es suficiente. Aun así, es bueno saber que el compilador `g++` también proporciona un tipo de 128 bits `__int128_t` con un rango de valores de  $-2^{127} \dots 2^{127} - 1$  o aproximadamente  $-10^{38} \dots 10^{38}$ . Sin embargo, este tipo no está disponible en todos los sistemas de concursos.



## Aritmética modular

Denotamos por  $x \bmod m$  el resto cuando  $x$  se divide por  $m$ . Por ejemplo,  $17 \bmod 5 = 2$ , porque  $17 = 3 \cdot 5 + 2$ .

A veces, la respuesta a un problema es un número muy grande pero es suficiente mostrarlo "módulo  $m$ ", es decir, el resto cuando la respuesta se divide por  $m$  (por ejemplo, "módulo  $10^9 + 7$ "). La idea es que incluso si la respuesta real es muy grande, basta con usar los tipos `int` y `long long`.

Una propiedad importante del resto es que en la suma, resta y multiplicación, el resto se puede tomar 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 tomar el resto después de cada operación y los números nunca se volverán 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";
```

Usualmente queremos que el resto siempre esté entre  $0 \dots m - 1$ . Sin embargo, en C++ y otros lenguajes, el resto de un número negativo es cero o negativo. Una forma fácil de asegurarse de que no haya restos negativos es primero calcular el resto como de costumbre y luego sumar  $m$  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 resto puede volverse negativo.

## Números de punto flotante

Los tipos de punto flotante usuales en la 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 usualmente se da en el enunciado del problema. Una forma fácil de mostrar la respuesta es usar la función `printf` y dar el número de lugares decimales en la cadena de formato. Por ejemplo, el siguiente código muestra el valor de  $x$  con 9 lugares decimales:

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

Una dificultad al usar números de punto flotante es que algunos números no pueden ser representados con precisión como números de punto flotante, 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 debería ser 1.

Es arriesgado comparar números de punto flotante con el operador `==`, porque es posible que los valores deberían ser iguales pero no lo son 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  $\varepsilon$ , donde  $\varepsilon$  es un número pequeño.

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

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

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

## 1.4 Acortando código

El código corto es ideal en la programación competitiva, porque los programas deben escribirse lo más rápido posible. Debido a esto, los programadores competitivos a menudo definen nombres más cortos para los tipos de datos y otras partes del código.

### Nombres de tipos

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 acortar de la siguiente manera:

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

El comando `typedef` también se puede usar con tipos más complejos. Por ejemplo, el siguiente código da el nombre `vi` a un vector de enteros y el nombre `pi` a una pareja que contiene dos enteros.

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

## Macros

Otra forma de acortar el código es definir **macros**. Un macro significa que ciertas cadenas en el código serán cambiadas antes de la compilación. En C++, los macros se definen usando la palabra clave `#define`.

Por ejemplo, podemos definir los 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;
```

se puede acortar de la siguiente manera:

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

Un macro también puede tener parámetros, lo que hace posible acortar bucles y otras estructuras. Por ejemplo, podemos definir el 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);
}
```

se puede acortar de la siguiente manera:

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

A veces, los macros causan errores que pueden ser difíciles de detectar. Por ejemplo, considera el siguiente macro que calcula el cuadrado de un número:

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

Este 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 versión mejor del 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 juegan un papel importante en la programación competitiva, y no es posible convertirse en un programador competitivo exitoso sin tener buenas habilidades matemáticas. Esta sección discute algunos conceptos y fórmulas matemáticas importantes que se necesitarán más adelante en el libro.

### Fórmulas de suma

Cada suma de la forma

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

donde  $k$  es un entero positivo, tiene una fórmula cerrada que es un polinomio de grado  $k + 1$ . Por ejemplo<sup>1</sup>,

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

y

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

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. La suma de una progresión aritmética se puede calcular usando la fórmula

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

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 consiste en  $n$  números y el valor de cada número es  $(a + b)/2$  en promedio.

Una **progresión geométrica** es una secuencia de números donde la razón 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

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

donde  $a$  es el primer número,  $b$  es el último número y la razón 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

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

Multiplicando ambos lados por  $k$ , obtenemos

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

---

<sup>1</sup> Incluso hay una fórmula general para tales sumas, llamada **fórmula de Faulhaber**, pero es demasiado compleja para presentarla aquí.

y resolviendo la ecuación

$$kS - S = bk - a$$

se obtiene la fórmula.

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

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

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

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

Un límite superior para una suma armónica es  $\log_2(n) + 1$ . A saber, podemos modificar cada término  $1/k$  de modo que  $k$  se convierta en la potencia de dos más cercana que no exceda a  $k$ . Por ejemplo, cuando  $n = 6$ , podemos estimar la suma de la siguiente manera:

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

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  $x \in S$ , y 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 utilizando operaciones de conjuntos:

- La **intersección**  $A \cap B$  consiste en los 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 en los elementos que están en  $A$  o  $B$  o en ambos. Por ejemplo, si  $A = \{3, 7\}$  y  $B = \{2, 3, 8\}$ , entonces  $A \cup B = \{2, 3, 7, 8\}$ .
- El **complemento**  $\bar{A}$  consiste en los elementos que no están en  $A$ . La interpretación de un complemento depende del **conjunto universal**, que contiene todos los elementos posibles. Por ejemplo, si  $A = \{1, 2, 5, 7\}$  y el conjunto universal es  $\{1, 2, \dots, 10\}$ , entonces  $\bar{A} = \{3, 4, 6, 8, 9, 10\}$ .
- La **diferencia**  $A \setminus B = A \cap \bar{B}$  consiste en los elementos que están en  $A$  pero no en  $B$ . Nótese 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  $S$ , decimos que  $A$  es un **subconjunto** de  $S$ , denotado por  $A \subset S$ . Un conjunto  $S$  siempre tiene  $2^{|S|}$  subconjuntos,

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

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

Algunos conjuntos usados frecuentemente son  $\mathbb{N}$  (números naturales),  $\mathbb{Z}$  (números enteros),  $\mathbb{Q}$  (números racionales) y  $\mathbb{R}$  (números reales). El conjunto  $\mathbb{N}$  se puede definir de dos maneras, dependiendo de la situación: ya sea  $\mathbb{N} = \{0, 1, 2, \dots\}$  o  $\mathbb{N} = \{1, 2, 3, \dots\}$ .

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 **verdadero** (1) o **falso** (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 cada vez que  $A$  es verdadero, también  $B$  es verdadero. La expresión  $A \Leftrightarrow B$  es verdadera si  $A$  y  $B$  son ambos verdaderos o ambos falsos.

Un **predicado** es una expresión que es verdadera o falsa dependiendo de sus parámetros. Los predicados usualmente se denotan con letras mayúsculas. Por ejemplo, podemos definir un predicado  $P(x)$  que es verdadero exactamente cuando  $x$  es 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 todos**) y  $\exists$  (**existe**). Por ejemplo,

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

significa que para cada elemento  $x$  en el conjunto, hay un elemento  $y$  en el conjunto tal que  $y$  es menor que  $x$ . Esto es verdadero en el conjunto de los 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 hay números  $a$  y  $b$  que son mayores que 1 y cuyo producto es  $x$ . Esta proposición es verdadera en el conjunto de los enteros.

## Funciones

La función  $\lfloor x \rfloor$  redondea el número  $x$  hacia abajo a un entero, y la función  $\lceil x \rceil$  redondea el número  $x$  hacia arriba a un entero. Por ejemplo,

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

Las funciones  $\min(x_1, x_2, \dots, x_n)$  y  $\max(x_1, x_2, \dots, x_n)$  dan el valor más pequeño y el más grande de los valores  $x_1, x_2, \dots, x_n$ . Por ejemplo,

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

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

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

o recursivamente

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

Los **números de Fibonacci** aparecen en muchas situaciones. Se pueden definir 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

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

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  $\log_k(x)$  equivale al número de veces que tenemos que dividir  $x$  por  $k$  antes de llegar al número 1. Por ejemplo,  $\log_2(32) = 5$  porque se necesitan 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 dividen algo a la mitad en cada paso. Por lo 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 en consecuencia,

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

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 en cualquier base si hay una forma de calcular logaritmos en alguna 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) es un concurso de programación anual para estudiantes de secundaria. Cada país puede enviar un equipo de cuatro estudiantes al concurso. Usualmente hay alrededor de 300 participantes de 80 países.

La IOI consiste en dos concursos de cinco horas cada uno. En ambos concursos, se les pide a los participantes resolver tres tareas de algoritmos de diversas dificultades. Las tareas se dividen en subtareas, cada una de las cuales tiene una puntuación asignada. Aunque los concursantes se dividen en equipos, compiten como individuos.

El temario de la IOI [41] regula los temas que pueden aparecer en las tareas de la IOI. Casi todos los temas del temario de la IOI están cubiertos en este libro.

Los participantes para la IOI se seleccionan a través de concursos nacionales. Antes de la 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-Pacífico (APIO).

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

## ICPC

El Concurso Internacional Colegial de Programación (ICPC) es un concurso de programación anual para estudiantes universitarios. Cada equipo en el concurso está compuesto por tres estudiantes, y a diferencia de la IOI, los estudiantes trabajan juntos; hay solo una computadora disponible para cada equipo.

El ICPC consiste en varias etapas, y finalmente los mejores equipos son invitados a las Finales Mundiales. Aunque hay decenas de miles de participantes en el concurso, solo hay un número pequeño<sup>2</sup> de plazas finales disponibles, así que incluso avanzar a las finales es un gran logro en algunas regiones.

En cada concurso ICPC, los equipos tienen cinco horas para resolver alrededor de diez problemas de algoritmos. Una solución a un problema se acepta solo si resuelve todos los casos de prueba eficientemente. Durante el concurso, los competidores pueden ver los resultados de otros equipos, pero durante la última hora la tabla de clasificación se congela y no es posible ver los resultados de las últimas entregas.

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

## Concursos en línea

También hay muchos concursos en línea que están abiertos para todos. En este momento, el sitio de concursos más activo es Codeforces, que organiza concursos aproximadamente cada semana. En Codeforces, los participantes se dividen en dos divisiones: los principiantes compiten en Div2 y los programadores más experimentados en 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 usan esos concursos para reclutar: desempeñarse bien en un concurso es una buena manera de demostrar las propias habilidades.

---

<sup>2</sup>El número exacto de plazas finales varía de año en año; en 2017, hubo 133 plazas finales.

## Libros

Ya existen algunos libros (además de este libro) que se enfocan en la programación competitiva y la resolución de problemas algorítmicos:

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

Por supuesto, los libros generales de algoritmos 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]



# Chapter 2

## Complejidad temporal

La eficiencia de los algoritmos es importante en la programación competitiva. Usualmente, es fácil diseñar un algoritmo que resuelva el problema lentamente, pero el verdadero desafío es inventar un algoritmo rápido. Si el algoritmo es demasiado lento, solo obtendrá puntos parciales o no obtendrá puntos en absoluto.

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

### 2.1 Reglas de cálculo

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

#### Bucles

Una razón común por la cual un algoritmo es lento es que contiene muchos bucles que recorren la entrada. Cuantos más bucles anidados contenga el algoritmo, más lento será. Si hay  $k$  bucles 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++) {  
    // codigo  
}
```

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

```
}
```

## Orden de magnitud

Una complejidad temporal no nos dice el número exacto de veces que se ejecuta el código dentro de un bucle, sino que solo muestra el orden de magnitud. En los siguientes ejemplos, el código dentro del bucle 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 consiste en 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 usualmente es el cuello de botella del código.

Por ejemplo, el siguiente código consiste en tres fases con complejidades temporales  $O(n)$ ,  $O(n^2)$  y  $O(n)$ . Por lo 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++) {
    // codigo
}

```

## Varias variables

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

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

```

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

```

## Recursión

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, considera la siguiente función:

```

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

```

La llamada  $f(n)$  provoca  $n$  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, considera 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 dos otras llamadas, excepto para  $n = 1$ . Veamos qué pasa cuando se llama a  $g$  con el parámetro  $n$ . La siguiente tabla muestra las llamadas a la función producidas por esta única llamada:

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

Basado en esto, la complejidad temporal es

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

## 2.2 Clases de complejidad

La siguiente lista contiene complejidades temporales comunes de algoritmos:

- $O(1)$  El tiempo de ejecución de un algoritmo de **tiempo constante** no depende del tamaño de la entrada. Un algoritmo típico de tiempo constante es una fórmula directa que calcula la respuesta.
- $O(\log n)$  Un algoritmo **logarítmico** a menudo divide a la mitad el tamaño de la entrada en cada paso. El tiempo de ejecución de dicho algoritmo es logarítmico, porque  $\log_2 n$  es igual al número de veces que  $n$  debe dividirse 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  $\sqrt{n} = n/\sqrt{n}$ , por lo que la raíz cuadrada  $\sqrt{n}$  se encuentra, en cierto sentido, en el medio de la entrada.
- $O(n)$  Un algoritmo **lineal** recorre la entrada un número constante de veces. Esta suele ser la mejor complejidad temporal posible, porque generalmente es necesario acceder a cada elemento de entrada al menos una vez antes de informar 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 use una estructura de datos donde cada operación toma tiempo  $O(\log n)$ .
- $O(n^2)$  Un algoritmo **cuadrático** a menudo contiene dos bucles anidados. Es posible recorrer todos los pares de elementos de entrada en tiempo  $O(n^2)$ .
- $O(n^3)$  Un algoritmo **cúbico** a menudo contiene tres bucles anidados. Es posible recorrer todas las tripletas de elementos de entrada en tiempo  $O(n^3)$ .
- $O(2^n)$  Esta complejidad temporal a menudo indica que el algoritmo itera a través de todos los 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 ejemplo, 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 a lo sumo  $O(n^k)$  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 significa aproximadamente que el algoritmo es *eficiente*.

La mayoría de los algoritmos en este libro son polinomiales. Sin embargo, hay muchos problemas importantes para los cuales no se conoce ningún algoritmo polinomial, es decir, nadie sabe cómo resolverlos de manera eficiente. Los problemas **NP-hard** son un conjunto importante de problemas para los cuales no se conoce ningún algoritmo polinomial<sup>1</sup>.

## 2.3 Estimación de eficiencia

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

Por ejemplo, supongamos que el límite de tiempo para un problema es de un segundo y el tamaño de entrada es  $n = 10^5$ . Si la complejidad temporal es  $O(n^2)$ , el algoritmo realizará aproximadamente  $(10^5)^2 = 10^{10}$  operaciones. Esto debería tomar al menos varias 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 temporal requerida por el algoritmo que resuelve el problema. La siguiente tabla contiene algunas estimaciones útiles suponiendo un límite de tiempo de un segundo.

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

Por ejemplo, si el tamaño de entrada es  $n = 10^5$ , probablemente se espera que la complejidad temporal del algoritmo sea  $O(n)$  o  $O(n \log n)$ . Esta información facilita el diseño del algoritmo, porque descarta enfoques que producirían un algoritmo con una peor complejidad temporal.

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

Aún así, es importante recordar que la complejidad temporal es solo una estimación de 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 real de ejecución del algoritmo.

## 2.4 Suma máxima de subarreglo

A menudo existen varios algoritmos posibles para resolver un problema de manera que sus complejidades temporales sean diferentes. Esta sección discute un problema clásico que tiene una solución directa de  $O(n^3)$ . Sin embargo, al diseñar un algoritmo mejor, es posible resolver el problema en tiempo  $O(n^2)$  e incluso en tiempo  $O(n)$ .

Dado un arreglo de  $n$  números, nuestra tarea es calcular la **suma máxima de subarreglo**, es decir, la suma más grande posible de una secuencia de valores consecutivos en el arreglo<sup>2</sup>. 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
----	---	---	----	---	---	----	---

una subarreglo siguiente 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 de subarreglo siempre es al menos 0.

### Algoritmo 1

Una manera directa de resolver el problema es recorrer todos los subarreglos posibles, calcular la suma de los valores en cada subarreglo y mantener la suma máxima. El siguiente código implementa este algoritmo:

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

---

<sup>2</sup>El libro de J. Bentley *Programming Pearls* [8] popularizó el problema.

Las variables  $a$  y  $b$  fijan el primer y último índice del subarreglo, y se calcula la suma de los valores en la variable  $suma$ . La variable  $mejor$  contiene la suma máxima encontrada durante la búsqueda.

La complejidad temporal del algoritmo es  $O(n^3)$ , porque consiste en tres bucles anidados que recorren la entrada.

## Algoritmo 2

Es fácil hacer que el Algoritmo 1 sea más eficiente eliminando uno de sus bucles. Esto es posible al calcular la suma al mismo tiempo que el extremo derecho del subarreglo se mueve. El resultado es el siguiente código:

```
int mejor = 0;
for (int a = 0; a < n; a++) {
    int suma = 0;
    for (int b = a; b < n; b++) {
        suma += array[b];
        mejor = max(mejor, suma);
    }
}
cout << mejor << "\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 que significa que solo un bucle es suficiente. La idea es calcular, para cada posición del arreglo, la suma máxima de un subarreglo que termina en esa posición. Luego, la respuesta para el problema es el máximo de esas sumas.

Consideremos el subproblema de encontrar la suma máxima del subarreglo que termina en la posición  $k$ . Existen dos posibilidades:

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

En el segundo caso, dado que queremos encontrar un subarreglo con la suma máxima, el subarreglo que termina en la posición  $k - 1$  también debería tener la suma máxima. Así, podemos resolver el problema eficientemente calculando la suma máxima del subarreglo para cada posición final de izquierda a derecha.

El siguiente código implementa el algoritmo:

```
int mejor = 0, suma = 0;
for (int k = 0; k < n; k++) {
```

---

<sup>3</sup>En [8], este algoritmo lineal se atribuye a J. B. Kadane, y a veces se llama **algoritmo de Kadane**.

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

El algoritmo contiene solo un bucle 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 debe examinar todos los elementos del arreglo al menos una vez.

## Comparación de eficiencia

Es interesante estudiar cuán eficientes 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 se midió.

tamaño del arreglo $n$	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 tamaños de entrada más grandes muestran diferencias notables en los tiempos de ejecución de los algoritmos. El Algoritmo 1 se vuelve lento cuando  $n = 10^4$ , y el Algoritmo 2 se vuelve lento cuando  $n = 10^5$ . Solo el Algoritmo 3 es capaz de procesar incluso las entradas más grandes instantáneamente.

# Chapter 3

## Ordenamiento

**Ordenamiento** es un problema fundamental de diseño de algoritmos. Muchos algoritmos eficientes utilizan el ordenamiento como una subrutina, porque a menudo es más fácil procesar datos si los elementos están en orden.

Por ejemplo, el problema "¿contiene un arreglo dos elementos iguales?" es fácil de resolver usando el ordenamiento. Si el arreglo contiene dos elementos iguales, estos estarán uno al lado del otro después del ordenamiento, por lo que es fácil encontrarlos. Además, el problema "¿cuál es el elemento más frecuente en un arreglo?" se puede resolver de manera similar.

Hay muchos algoritmos para el ordenamiento, y estos son también buenos ejemplos de cómo aplicar diferentes técnicas de diseño de algoritmos. Los algoritmos generales de ordenamiento eficientes funcionan en tiempo  $O(n \log n)$ , y muchos algoritmos que utilizan el ordenamiento como una subrutina también tienen esta complejidad de tiempo.

### 3.1 Teoría de ordenamiento

El problema básico en el ordenamiento es el siguiente:

Dado un arreglo que contiene  $n$  elementos, tu tarea es ordenar los elementos en orden creciente.

Por ejemplo, el arreglo

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

será como sigue después del ordenamiento:

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

#### Algoritmos $O(n^2)$

Los algoritmos simples para ordenar un arreglo funcionan en tiempo  $O(n^2)$ . Tales algoritmos son cortos y generalmente constan de dos bucles anidados. Un famoso

algoritmo de ordenamiento en tiempo  $O(n^2)$  es **ordenamiento por burbuja** donde los elementos "burbujean" en el arreglo de acuerdo con sus valores.

El ordenamiento por burbuja consta de  $n$  rondas. En cada ronda, el algoritmo itera a través de los elementos del arreglo. Cuando se encuentran dos elementos consecutivos que no están en el orden correcto, el algoritmo los intercambia. El algoritmo se puede implementar de la siguiente manera:

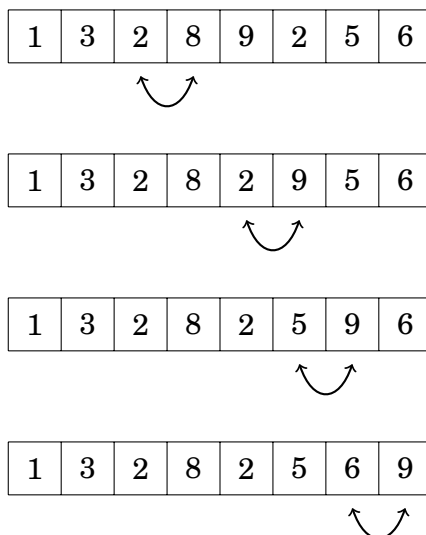
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n-1; j++) {  
        if (array[j] > array[j+1]) {  
            swap(array[j], array[j+1]);  
        }  
    }  
}
```

Después de la primera ronda del algoritmo, el elemento más grande estará en la posición correcta, y en general, después de  $k$  rondas, los  $k$  elementos más grandes estarán en las posiciones correctas. Por lo tanto, después de  $n$  rondas, todo el arreglo estará ordenado.

Por ejemplo, en el arreglo

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

la primera ronda de ordenamiento por burbuja intercambia elementos de la siguiente manera:



## Inversiones

La ordenación por burbuja es un ejemplo de un algoritmo de ordenación que siempre intercambia elementos *consecutivos* en la matriz. Resulta que la complejidad temporal de este tipo de algoritmo es *siempre* al menos  $O(n^2)$ , porque en el peor caso, se necesitan  $O(n^2)$  intercambios para ordenar la matriz.

Un concepto útil al analizar algoritmos de ordenación es una **inversión**: un par de elementos de la matriz ( $\text{array}[a], \text{array}[b]$ ) tal que  $a < b$  y  $\text{array}[a] > \text{array}[b]$ , es decir, los elementos están en el orden incorrecto. Por ejemplo, la matriz

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

tiene tres inversiones: (6, 3), (6, 5) y (9, 8). El número de inversiones indica cuánto trabajo se necesita para ordenar la matriz. Una matriz está completamente ordenada cuando no hay inversiones. Por otro lado, si los elementos de la matriz están en orden inverso, el número de inversiones es el mayor posible:

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

Intercambiar un par de elementos consecutivos que estén en el orden incorrecto elimina exactamente una inversión de la matriz. Por lo tanto, si un algoritmo de ordenación solo puede intercambiar elementos consecutivos, cada intercambio elimina como máximo una inversión, y la complejidad temporal del algoritmo es al menos  $O(n^2)$ .

## $O(n \log n)$ algoritmos

Es posible ordenar una matriz de forma eficiente en tiempo  $O(n \log n)$  usando algoritmos que no se limitan a intercambiar elementos consecutivos. Un algoritmo de este tipo es **merge sort**<sup>1</sup>, que se basa en la recursión.

Merge sort ordena una submatriz  $\text{array}[a \dots b]$  de la siguiente manera:

1. Si  $a = b$ , no hacer nada, porque la submatriz ya está ordenada.
2. Calcular la posición del elemento del medio:  $k = \lfloor (a + b)/2 \rfloor$ .
3. Ordenar recursivamente la submatriz  $\text{array}[a \dots k]$ .
4. Ordenar recursivamente la submatriz  $\text{array}[k + 1 \dots b]$ .
5. *Combinar* las submatrices ordenadas  $\text{array}[a \dots k]$  y  $\text{array}[k + 1 \dots b]$  en una submatriz ordenada  $\text{array}[a \dots b]$ .

Merge sort es un algoritmo eficiente, porque reduce a la mitad el tamaño de la submatriz en cada paso. La recursión consta de  $O(\log n)$  niveles, y el procesamiento de cada nivel lleva  $O(n)$  tiempo. Combinar las submatrices  $\text{array}[a \dots k]$  y  $\text{array}[k + 1 \dots b]$  es posible en tiempo lineal, porque ya están ordenadas.

Por ejemplo, considere la ordenación de la siguiente matriz:

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

La matriz se dividirá en dos submatrices de la siguiente manera:

---

<sup>1</sup>Según [47], la ordenación por mezcla fue inventada por J. von Neumann en 1945.

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

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

Entonces, las submatrices se ordenarán recursivamente de la siguiente manera:

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

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

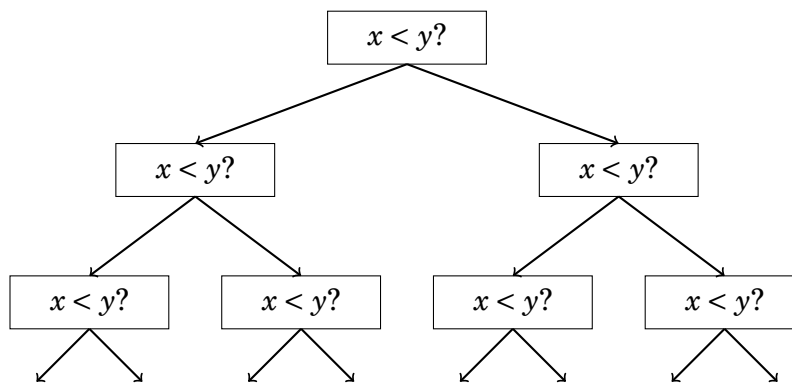
Finalmente, el algoritmo combina las ordenadas submatrices y crea la matriz ordenada final:

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

## Límite inferior de ordenación

¿Es posible ordenar una matriz más rápido que en tiempo  $O(n \log n)$ ? Resulta que esto *no* es posible cuando nos limitamos a algoritmos de ordenación que se basan en la comparación de elementos de la matriz.

El límite inferior para la complejidad temporal se puede probar considerando la ordenación como un proceso donde cada comparación de dos elementos proporciona más información sobre el contenido de la matriz. El proceso crea el siguiente árbol:



Aquí " $x < y$ ?" significa que algunos elementos  $x$  e  $y$  se comparan. Si  $x < y$ , el proceso continúa a la izquierda, y de lo contrario a la derecha. Los resultados del proceso son las posibles formas de ordenar la matriz, un total de  $n!$  maneras. Por esta razón, la altura del árbol debe ser al menos

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

Obtenemos un límite inferior para esta suma eligiendo los últimos  $n/2$  elementos y cambiando el valor de cada elemento a  $\log_2(n/2)$ . Esto produce una estimación

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

por lo que la altura del árbol y el mínimo número posible de pasos en un ordenamiento algoritmo en el peor de los casos es al menos  $n \log n$ .



## Ordenación por conteo

El límite inferior  $n \log n$  no se aplica a algoritmos que no comparan elementos de la matriz sino que utilizan alguna otra información. Un ejemplo de tal algoritmo es **ordenación por conteo** que ordena una matriz en tiempo  $O(n)$  asumiendo que cada elemento de la matriz es un entero entre  $0 \dots c$  y  $c = O(n)$ .

El algoritmo crea una matriz de *contabilidad*, cuyos índices son elementos de la matriz original. El algoritmo itera a través de la matriz original y calcula cuántas veces aparece cada elemento en la matriz.

Por ejemplo, la matriz

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

corresponde a la siguiente matriz de contabilidad:

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

Por ejemplo, el valor en la posición 3 en la matriz de contabilidad es 2, porque el elemento 3 aparece 2 veces en la matriz original.

La construcción de la matriz de contabilidad toma tiempo  $O(n)$ . Después de esto, la matriz ordenada se puede crear en tiempo  $O(n)$  porque el número de ocurrencias de cada elemento se puede recuperar de la matriz de contabilidad. Por lo tanto, la complejidad temporal total de la contabilidad ordenar es  $O(n)$ .

La ordenación por conteo es un algoritmo muy eficiente pero solo se puede utilizar cuando la constante  $c$  es lo suficientemente pequeña, para que los elementos de la matriz puedan usarse como índices en la matriz de contabilidad.

## 3.2 Ordenación en C++

Casi nunca es una buena idea usar un algoritmo de ordenación hecho en casa en un concurso, porque hay buenas implementaciones disponibles en lenguajes de programación. Por ejemplo, la biblioteca estándar de C++ contiene la función `sort` que se puede utilizar fácilmente para ordenar matrices y otras estructuras de datos.

Hay muchos beneficios en usar una función de biblioteca. Primero, ahorra tiempo porque no hay necesidad de implementar la función. En segundo lugar, la implementación de la biblioteca es ciertamente correcta y eficiente: no es probable que una función de ordenación hecha en casa sea mejor.

En esta sección veremos cómo usar la función `sort` de C++. El siguiente código ordena un vector en orden creciente:

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

Después de la ordenación, el contenido del vector será `[2,3,3,4,5,5,8]`. El orden de clasificación predeterminado es creciente, pero un orden inverso es posible de la siguiente manera:

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

Se puede ordenar una matriz ordinaria de la siguiente manera:

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

El siguiente código ordena la cadena s:

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

Ordenar una cadena significa que los caracteres de la cadena están ordenados. Por ejemplo, la cadena "monkey" se convierte en "ekmnoy".

## Operadores de comparación

La función `sort` requiere que se defina un **operador de comparación** para el tipo de datos de los elementos a ordenar. Al ordenar, este operador se utilizará cada vez que sea necesario averiguar el orden de dos elementos.

La mayoría de los tipos de datos de C++ tienen un operador de comparación integrado, y los elementos de esos tipos se pueden ordenar automáticamente. Por ejemplo, los números se ordenan según sus valores y las cadenas se ordenan en orden alfabético.

Los pares (`pair`) se ordenan principalmente según sus primeros elementos (`first`). Sin embargo, si los primeros elementos de dos pares son iguales, se ordenan según sus segundos elementos (`second`):

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

Después de esto, el orden de los pares es (1,2), (1,5) y (2,3).

De manera similar, las tuplas (`tuple`) se ordenan principalmente por el primer elemento, secundariamente por el segundo elemento, etc.<sup>2</sup>:

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

Después de esto, el orden de las tuplas es (1,5,3), (2,1,3) y (2,1,4).

## Estructuras definidas por el usuario

Las estructuras definidas por el usuario no tienen un operador de comparación automáticamente. El operador debe definirse dentro de la estructura como una función `operator<`, cuyo parámetro es otro elemento del mismo tipo. El operador

---

<sup>2</sup>Tenga en cuenta que en algunos compiladores más antiguos, la función `make_tuple` debe utilizarse para crear una tupla en lugar de llaves (por ejemplo, `make_tuple(2,1,4)` en lugar de `{2,1,4}`).

debe devolver true si el elemento es menor que el parámetro, y false en caso contrario.

Por ejemplo, la siguiente estructura P contiene las coordenadas x e y de un punto. El operador de comparación se define de modo que los puntos se ordenen principalmente por la coordenada x y secundariamente por la coordenada y.

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

## Funciones de comparación

También es posible dar una externa **función de comparación** a la función sort como una función de devolución de llamada. Por ejemplo, la siguiente función de comparación comp ordena las cadenas principalmente por longitud y secundariamente por orden alfabético:

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

Ahora un vector de cadenas se puede ordenar de la siguiente manera:

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

## 3.3 Búsqueda binaria

Un método general para buscar un elemento en un array es usar un bucle for que itera a través de los elementos del array. Por ejemplo, el siguiente código busca un elemento x en un array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x encontrado en el índice i
    }
}
```

La complejidad temporal de este enfoque es  $O(n)$ , porque en el peor de los casos, es necesario comprobar todos los elementos del array. Si el orden de los elementos es arbitrario, este es también el mejor enfoque posible, porque no

hay información adicional disponible sobre dónde en el array debemos buscar el elemento  $x$ .

Sin embargo, si el array está *ordenado*, la situación es diferente. En este caso, es posible realizar la búsqueda mucho más rápido, porque el orden de los elementos en el array guía la búsqueda. El siguiente algoritmo de **búsqueda binaria** busca eficientemente un elemento en un array ordenado en  $O(\log n)$  tiempo.

## Método 1

La forma usual de implementar la búsqueda binaria se asemeja a buscar una palabra en un diccionario. La búsqueda mantiene una región activa en la matriz, que inicialmente contiene todos los elementos de la matriz. Luego, se realiza una serie de pasos, cada uno de los cuales reduce a la mitad el tamaño de la región.

En cada paso, la búsqueda verifica el elemento del medio de la región activa. Si el elemento del medio es el elemento objetivo, la búsqueda termina. De lo contrario, la búsqueda continúa recursivamente a la mitad izquierda o derecha de la región, dependiendo del valor del elemento del medio.

La idea anterior se puede implementar de la siguiente manera:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x encontrado en el indice k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

En esta implementación, la región activa es  $a \dots b$ , y inicialmente la región es  $0 \dots n-1$ . El algoritmo reduce a la mitad el tamaño de la región en cada paso, por lo que la complejidad temporal es  $O(\log n)$ .

## Método 2

Un método alternativo para implementar la búsqueda binaria se basa en una forma eficiente de iterar a través de los elementos de la matriz. La idea es hacer saltos y reducir la velocidad cuando nos acercamos al elemento objetivo.

La búsqueda recorre la matriz de izquierda a derecha, y la longitud de salto inicial es  $n/2$ . En cada paso, la longitud del salto se reducirá a la mitad: primero  $n/4$ , luego  $n/8$ ,  $n/16$ , etc., hasta que finalmente la longitud sea 1. Después de los saltos, o bien se ha encontrado el elemento objetivo o sabemos que no aparece en la matriz.

El siguiente código implementa la idea anterior:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
```

```

    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x encontrado en el índice k
}

```

Durante la búsqueda, la variable  $b$  contiene la longitud del salto actual. La complejidad temporal del algoritmo es  $O(\log n)$ , porque el código en el bucle `while` se ejecuta como máximo dos veces para cada longitud de salto.

## Funciones de C++

La biblioteca estándar de C++ contiene las siguientes funciones que se basan en la búsqueda binaria y funcionan en tiempo logarítmico:

- `lower_bound` devuelve un puntero al primer elemento de la matriz cuyo valor es al menos  $x$ .
- `upper_bound` devuelve un puntero al primer elemento de la matriz cuyo valor es mayor que  $x$ .
- `equal_range` devuelve ambos punteros anteriores.

Las funciones asumen que la matriz está ordenada. Si no existe tal elemento, el puntero apunta a el elemento después del último elemento de la matriz. Por ejemplo, el siguiente código descubre si una matriz contiene un elemento con valor  $x$ :

```

auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x encontrado en el índice k
}

```

Luego, el siguiente código cuenta el número de elementos cuyo valor es  $x$ :

```

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

```

Usando `equal_range`, el código se vuelve más corto:

```

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

```

## Encontrar la solución más pequeña

Un uso importante de la búsqueda binaria es encontrar la posición donde cambia el valor de una *función*. Supongamos que queremos encontrar el valor más

pequeño  $k$  que sea una solución válida para un problema. Se nos da una función  $ok(x)$  que devuelve `true` si  $x$  es una solución válida y `false` de lo contrario. Además, sabemos que  $ok(x)$  es `false` cuando  $x < k$  y `true` cuando  $x \geq k$ . La situación se ve de la siguiente manera:

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

Ahora, el valor de  $k$  se puede encontrar usando la búsqueda binaria:

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

La búsqueda encuentra el valor más grande de  $x$  para el cual  $ok(x)$  es `false`. Por lo tanto, el siguiente valor  $k = x + 1$  es el valor más pequeño posible para el cual  $ok(k)$  es `true`. La longitud de salto inicial  $z$  tiene que ser lo suficientemente grande, por ejemplo, algún valor para el que sabemos de antemano que  $ok(z)$  es `true`.

El algoritmo llama a la función  $ok$   $O(\log z)$  veces, por lo que la complejidad temporal total depende de la función  $ok$ . Por ejemplo, si la función funciona en tiempo  $O(n)$ , la complejidad temporal total es  $O(n \log z)$ .

## Encontrar el valor máximo

La búsqueda binaria también se puede utilizar para encontrar el valor máximo de una función que es primero creciente y luego decreciente. Nuestra tarea es encontrar una posición  $k$  tal que

- $f(x) < f(x+1)$  cuando  $x < k$ , y
- $f(x) > f(x+1)$  cuando  $x \geq k$ .

La idea es usar la búsqueda binaria para encontrar el valor más grande de  $x$  para el cual  $f(x) < f(x+1)$ . Esto implica que  $k = x + 1$  porque  $f(x+1) > f(x+2)$ . El siguiente código implementa la búsqueda:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Tenga en cuenta que a diferencia de la búsqueda binaria ordinaria, aquí no se permite que los valores consecutivos de la función sean iguales. En este caso, no sería posible saber cómo continuar la búsqueda.





# Chapter 4

## Estructuras de datos

Una **estructura de datos** es una forma de almacenar datos en la memoria de una computadora. Es importante elegir una estructura de datos apropiada para un problema, porque cada estructura de datos tiene sus propias ventajas y desventajas. La pregunta crucial es: ¿qué operaciones son eficientes en la estructura de datos elegida?

Este capítulo presenta las estructuras de datos más importantes en la biblioteca estándar de C++. Es una buena idea usar la biblioteca estándar siempre que sea posible, porque ahorrará mucho tiempo. Más adelante en el libro, aprenderemos sobre estructuras de datos más sofisticadas que no están disponibles en la biblioteca estándar.

### 4.1 Arreglos dinámicos

Un **arreglo dinámico** es un arreglo cuyo tamaño se puede cambiar durante la ejecución del programa. El arreglo dinámico más popular en C++ es la estructura `vector`, que se puede usar casi como un arreglo ordinario.

El siguiente código crea un vector vacío y agrega tres elementos a él:

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

Después de esto, los elementos se pueden acceder como en un arreglo ordinario:

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

La función `size` devuelve el número de elementos en el vector. El siguiente código itera a través del vector e imprime todos los elementos en él:

```
for (int i = 0; i < v.size(); i++) {
```

```
    cout << v[i] << "\n";  
}
```

Una forma más corta de iterar a través de un vector es la siguiente:

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

La función `back` devuelve el último elemento en el vector, y la función `pop_back` elimina el último elemento:

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

El siguiente código crea un vector con cinco elementos:

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

Otra forma de crear un vector es dar el número de elementos y el valor inicial para cada elemento:

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

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

La implementación interna de un vector utiliza un arreglo ordinario. Si el tamaño del vector aumenta y el arreglo se vuelve demasiado pequeño, se asigna un nuevo arreglo y todos los elementos se mueven al nuevo arreglo. Sin embargo, esto no sucede a menudo y el tiempo de complejidad promedio de `push_back` es  $O(1)$ .

La estructura `string` también es un arreglo dinámico que se puede usar casi como un vector. Además, hay una sintaxis especial para cadenas que no está disponible en otras estructuras de datos. Las cadenas se pueden combinar usando el símbolo `+`. La función `substr(k,x)` devuelve la subcadena que comienza en la posición *k* y tiene longitud *x*, y la función `find(t)` encuentra la posición de la primera aparición de una subcadena *t*.

El siguiente código presenta algunas operaciones de cadena:

```
string a = "hatti";  
string b = a+a;  
cout << b << "\n"; // hattihatti
```

```
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

## 4.2 Estructuras de conjunto

Un **conjunto** es una estructura de datos que mantiene una colección de elementos. Las operaciones básicas de los conjuntos son la inserción de elementos, la búsqueda y la eliminación.

La biblioteca estándar de C++ contiene dos implementaciones de conjunto: La estructura `set` se basa en un árbol binario equilibrado y sus operaciones funcionan en tiempo  $O(\log n)$ . La estructura `unordered_set` usa hashing, y sus operaciones funcionan en tiempo  $O(1)$  en promedio.

La elección de qué implementación de conjunto usar es a menudo una cuestión de gusto. El beneficio de la estructura `set` es que mantiene el orden de los elementos y proporciona funciones que no están disponibles en `unordered_set`. Por otro lado, `unordered_set` puede ser más eficiente.

El siguiente código crea un conjunto que contiene enteros, y muestra algunas de las operaciones. La función `insert` agrega un elemento al conjunto, la función `count` devuelve el número de ocurrencias de un elemento en el conjunto, y la función `erase` elimina un elemento del conjunto.

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

Un conjunto se puede usar principalmente como un vector, pero no es posible acceder a los elementos usando la notación `[]`. El siguiente código crea un conjunto, imprime el número de elementos en él, y luego itera a través de todos los elementos:

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

Una propiedad importante de los conjuntos es que todos sus elementos son

*distintos*. Por lo tanto, la función `count` siempre devuelve ya sea 0 (el elemento no está en el conjunto) o 1 (el elemento está en el conjunto), y la función `insert` nunca agrega un elemento al conjunto si ya está ahí. El siguiente código ilustra esto:

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

C++ también contiene las estructuras `multiset` y `unordered_multiset` que de otra manera funcionan como `set` y `unordered_set` pero pueden contener múltiples instancias de un elemento. Por ejemplo, en el siguiente código todas las tres instancias del número 5 se agregan a un `multiset`:

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

La función `erase` elimina todas las instancias de un elemento de un `multiset`:

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

A menudo, solo se debe eliminar una instancia, lo que se puede hacer de la siguiente manera:

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

## 4.3 Estructuras de mapa

Un **mapa** es una matriz generalizada que consta de pares clave-valor. Si bien las claves en una matriz ordinaria son siempre los enteros consecutivos  $0, 1, \dots, n-1$ , donde  $n$  es el tamaño de la matriz, las claves en un mapa pueden ser de cualquier tipo de datos y no tienen que ser valores consecutivos.

La biblioteca estándar de C++ contiene dos mapas implementaciones que corresponden al conjunto implementaciones: la estructura `map` se basa en un balanceado árbol binario y acceder a elementos toma  $O(\log n)$  tiempo, mientras que la estructura `unordered_map` usa hashing y acceder a elementos toma  $O(1)$  tiempo en promedio.

El siguiente código crea un mapa donde las claves son cadenas y los valores son enteros:

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

Si se solicita el valor de una clave pero el mapa no lo contiene, la clave se agrega automáticamente al mapa con un valor predeterminado. Por ejemplo, en el siguiente código, la clave "aybaltu" con valor 0 se agrega al mapa.

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

La función `count` comprueba si una clave existe en un mapa:

```
if (m.count("aybaltu")) {
    // clave existe
}
```

El siguiente código imprime todas las claves y valores en un mapa:

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

## 4.4 Iteradores y rangos

Muchas funciones en la biblioteca estándar de C++ operan con iteradores. Un **iterador** es una variable que apunta a un elemento en una estructura de datos.

Los iteradores que se usan a menudo `begin` y `end` definen un rango que contiene todos los elementos en una estructura de datos. El iterador `begin` apunta a el primer elemento en la estructura de datos, y el iterador `end` apunta a la posición *después* del último elemento. La situación se ve de la siguiente manera:

```

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

Tenga en cuenta la asimetría en los iteradores: `s.begin()` apunta a un elemento en la estructura de datos, mientras que `s.end()` apunta fuera de la estructura de datos. Por lo tanto, el rango definido por los iteradores es *semia-bierto*.

### Trabajando con rangos

Los iteradores se utilizan en las funciones de la biblioteca estándar de C++ que reciben un rango de elementos en una estructura de datos. Por lo general,

queremos procesar todos los elementos en una estructura de datos, por lo que los iteradores `begin` y `end` se dan para la función.

Por ejemplo, el siguiente código ordena un vector usando la función `sort`, luego invierte el orden de los elementos usando la función `reverse`, y finalmente baraja el orden de los elementos usando la función `random_shuffle`.

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

Estas funciones también se pueden usar con una matriz ordinaria. En este caso, las funciones reciben punteros a la matriz en lugar de iteradores:

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

## Iteradores de conjunto

Los iteradores se utilizan a menudo para acceder a los elementos de un conjunto. El siguiente código crea un iterador `it` que apunta al elemento más pequeño de un conjunto:

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

Una forma más corta de escribir el código es la siguiente:

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

El elemento al que apunta un iterador se puede acceder utilizando el símbolo `*`. Por ejemplo, el siguiente código imprime el primer elemento del conjunto:

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

Los iteradores se pueden mover utilizando los operadores `++` (hacia adelante) y `--` (hacia atrás), lo que significa que el iterador se mueve al siguiente o elemento anterior del conjunto.

El siguiente código imprime todos los elementos en orden creciente:

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

El siguiente código imprime el elemento más grande del conjunto:

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

La función `find(x)` devuelve un iterador que apunta a un elemento cuyo valor es `x`. Sin embargo, si el conjunto no contiene `x`, el iterador será `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x no se encuentra
}
```

La función `lower_bound(x)` devuelve un iterador al elemento más pequeño del conjunto cuyo valor es *al menos* `x`, y la función `upper_bound(x)` devuelve un iterador al elemento más pequeño del conjunto cuyo valor es *mayor que* `x`. En ambas funciones, si no existe tal elemento, el valor de retorno es `end`. Estas

funciones no son compatibles con la estructura `unordered_set` que no mantiene el orden de los elementos.

Por ejemplo, el siguiente código encuentra el elemento más cercano a  $x$ :

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

El código asume que el conjunto no está vacío, y recorre todos los casos posibles utilizando un iterador `it`. Primero, el iterador apunta al elemento más pequeño cuyo valor es al menos  $x$ . Si `it` es igual a `begin`, el elemento correspondiente está más cerca de  $x$ . Si `it` es igual a `end`, el elemento más grande del conjunto está más cerca de  $x$ . Si ninguno de los casos anteriores se cumple, el elemento más cercano a  $x$  es o bien el elemento que corresponde a `it` o el elemento anterior.

## 4.5 Otras estructuras

### Bitset

Un **bitset** es una matriz cuyo valor de cada elemento es 0 o 1. Por ejemplo, el siguiente código crea un `bitset` que contiene 10 elementos:

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

La ventaja de usar bitsets es que requieren menos memoria que las matrices ordinarias, porque cada elemento de un `bitset` solo utiliza un bit de memoria. Por ejemplo, si se almacenan  $n$  bits en una matriz `int`, se utilizarán  $32n$  bits de memoria, pero un `bitset` correspondiente solo requiere  $n$  bits de memoria. Además, los valores de un `bitset` se pueden manipular de manera eficiente utilizando operadores bit a bit, lo que permite optimizar algoritmos utilizando bitsets.

El siguiente código muestra otra forma de crear el `bitset` anterior:

```
bitset<10> s(string("0010011010")); // de derecha a izquierda
```



```
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

La función `count` devuelve el número de unos en el `bitset`:

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

El siguiente código muestra ejemplos de uso de operaciones bit a bit:

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

## Deque

Un **deque** es una matriz dinámica cuyo tamaño se puede cambiar de forma eficiente en ambos extremos de la matriz. Al igual que un vector, un deque proporciona las funciones `push_back` y `pop_back`, pero también incluye las funciones `push_front` y `pop_front` que no están disponibles en un vector.

Un deque se puede utilizar de la siguiente manera:

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

La implementación interna de una cola de doble extremo es más compleja que la de un vector, y por esta razón, una cola de doble extremo es más lenta que un vector. Aun así, tanto agregar como eliminar elementos toma  $O(1)$  tiempo en promedio en ambos extremos.

## Pila

Una **pila** es una estructura de datos que proporciona dos operaciones de tiempo  $O(1)$ : agregar un elemento a la cima, y eliminar un elemento de la cima. Solo es posible acceder al elemento superior de una pila.

El siguiente código muestra cómo se puede usar una pila:

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

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

## Cola

Una **cola** también proporciona dos operaciones de tiempo  $O(1)$ : agregar un elemento al final de la cola, y eliminar el primer elemento de la cola. Solo es posible acceder al primero y al último elemento de una cola.

El siguiente código muestra cómo se puede usar una cola:

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

## Cola de prioridad

Una **cola de prioridad** mantiene un conjunto de elementos. Las operaciones compatibles son la inserción y, dependiendo del tipo de la cola, la recuperación y la eliminación de ya sea el elemento mínimo o máximo. La inserción y la eliminación toman  $O(\log n)$  tiempo, y la recuperación toma  $O(1)$  tiempo.

Si bien un conjunto ordenado admite de manera eficiente todas las operaciones de una cola de prioridad, la ventaja de usar una cola de prioridad es que tiene factores constantes más pequeños. Una cola de prioridad generalmente se implementa utilizando una estructura de montón que es mucho más simple que un árbol binario equilibrado utilizado en un conjunto ordenado.

De forma predeterminada, los elementos de una cola de prioridad de C++ están ordenados en orden decreciente, y es posible encontrar y eliminar el elemento más grande en la cola. El siguiente código ilustra esto:

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

Si queremos crear una cola de prioridad que admita la búsqueda y la eliminación del elemento más pequeño, podemos hacerlo de la siguiente manera:

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

## Estructuras de datos basadas en políticas

El compilador g++ también admite algunas estructuras de datos que no son parte de la biblioteca estándar de C++. Tales estructuras se llaman estructuras de datos *basadas en políticas*. Para usar estas estructuras, las siguientes líneas deben agregarse al código:

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

Después de esto, podemos definir una estructura de datos `indexed_set` que es como `set` pero se puede indexar como una matriz. La definición para valores `int` es la siguiente:

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

Ahora podemos crear un conjunto de la siguiente manera:

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

La especialidad de este conjunto es que tenemos acceso a los índices que los elementos tendrían en una matriz ordenada. La función `find_by_order` devuelve un iterador al elemento en una posición dada:

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

Y la función `order_of_key` devuelve la posición de un elemento dado:

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

Si el elemento no aparece en el conjunto, obtenemos la posición que el elemento tendría en el conjunto:

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

Ambas funciones funcionan en tiempo logarítmico.

## 4.6 Comparación con la clasificación

A menudo es posible resolver un problema utilizando estructuras de datos o clasificación. A veces hay diferencias notables en la eficiencia real de estos enfoques, que pueden estar ocultas en sus complejidades de tiempo.

Consideremos un problema donde se nos dan dos listas  $A$  y  $B$  que ambas contienen  $n$  elementos. Nuestra tarea es calcular el número de elementos que pertenecen a ambas listas. Por ejemplo, para las listas

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

la respuesta es 3 porque los números 2, 5 y 9 pertenecen a ambas listas.

Una solución sencilla al problema es recorrer todos los pares de elementos en  $O(n^2)$  tiempo, pero a continuación nos centraremos en algoritmos más eficientes.

### Algoritmo 1

Construimos un conjunto de los elementos que aparecen en  $A$ , y después de esto, iteramos a través de los elementos de  $B$  y verificamos para cada elemento si también pertenece a  $A$ . Esto es eficiente porque los elementos de  $A$  están en un conjunto. Usando la estructura `set`, la complejidad temporal del algoritmo es  $O(n \log n)$ .

### Algoritmo 2

No es necesario mantener un conjunto ordenado, así que en lugar de la estructura `set` también podemos usar la estructura `unordered_set`. Esta es una forma fácil de hacer el algoritmo más eficiente, porque solo tenemos que cambiar la estructura de datos subyacente. La complejidad temporal del nuevo algoritmo es  $O(n)$ .

### Algoritmo 3

En lugar de estructuras de datos, podemos usar la ordenación. Primero, ordenamos ambas listas  $A$  y  $B$ . Después de esto, iteramos a través de ambas listas al mismo tiempo y encontramos los elementos comunes. La complejidad temporal de la ordenación es  $O(n \log n)$ , y el resto del algoritmo funciona en tiempo  $O(n)$ , por lo que la complejidad temporal total es  $O(n \log n)$ .

## Comparación de eficiencia

La siguiente tabla muestra qué tan eficientes son los algoritmos anteriores cuando  $n$  varía y los elementos de las listas son enteros aleatorios entre  $1 \dots 10^9$ :

$n$	Algoritmo 1	Algoritmo 2	Algoritmo 3
$10^6$	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

Los algoritmos 1 y 2 son iguales excepto que usan diferentes estructuras de conjuntos. En este problema, esta elección tiene un efecto importante en el tiempo de ejecución, porque el Algoritmo 2 es 4–5 veces más rápido que el Algoritmo 1.

Sin embargo, el algoritmo más eficiente es el Algoritmo 3 que utiliza la ordenación. Solo usa la mitad del tiempo en comparación con el Algoritmo 2. Curiosamente, la complejidad temporal de ambos Algoritmo 1 y Algoritmo 3 es  $O(n \log n)$ , pero a pesar de esto, el Algoritmo 3 es diez veces más rápido. Esto se puede explicar por el hecho de que la ordenación es un procedimiento simple y se realiza solo una vez al principio del Algoritmo 3, y el resto del algoritmo funciona en tiempo lineal. Por otro lado, el Algoritmo 1 mantiene un complejo árbol binario balanceado durante todo el algoritmo.



# Chapter 5

## Búsqueda completa

**Búsqueda completa** es un método general que se puede utilizar para resolver casi cualquier problema de algoritmo. La idea es generar todas las posibles soluciones al problema utilizando la fuerza bruta, y luego seleccionar la mejor solución o contar la cantidad de soluciones, dependiendo del problema.

La búsqueda completa es una buena técnica si hay tiempo suficiente para recorrer todas las soluciones, porque la búsqueda suele ser fácil de implementar y siempre da la respuesta correcta. Si la búsqueda completa es demasiado lenta, es posible que se necesiten otras técnicas, como los algoritmos voraces o la programación dinámica.

### 5.1 Generación de subconjuntos

Primero consideramos el problema de generar todos los subconjuntos de un conjunto de  $n$  elementos. Por ejemplo, los subconjuntos de  $\{0, 1, 2\}$  son  $\emptyset$ ,  $\{0\}$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{0, 1\}$ ,  $\{0, 2\}$ ,  $\{1, 2\}$  y  $\{0, 1, 2\}$ . Hay dos métodos comunes para generar subconjuntos: podemos realizar una búsqueda recursiva o explotar la representación en bits de los enteros.

#### Método 1

Una forma elegante de recorrer todos los subconjuntos de un conjunto es usar la recursión. La siguiente función `search` genera los subconjuntos del conjunto  $\{0, 1, \dots, n-1\}$ . La función mantiene un vector `subset` que contendrá los elementos de cada subconjunto. La búsqueda comienza cuando la función se llama con el parámetro 0.

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

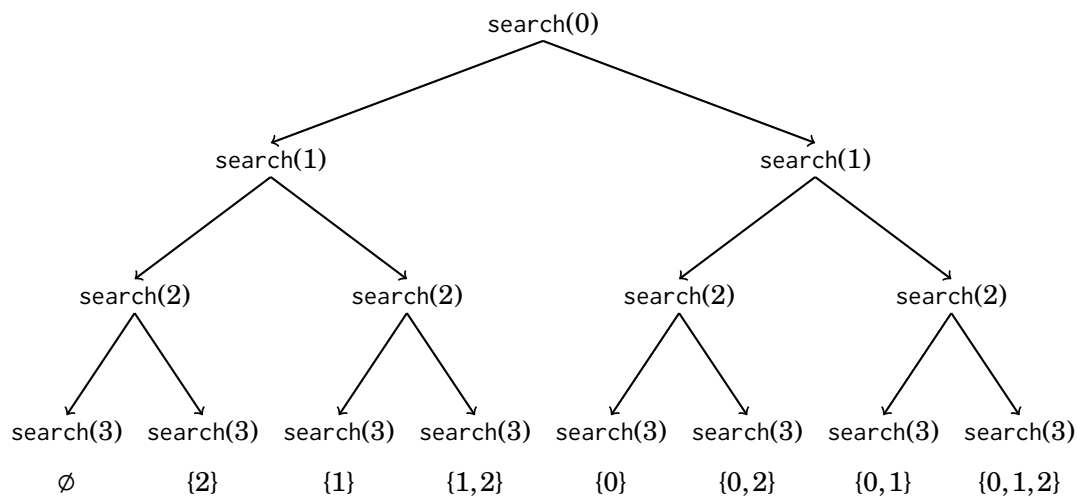
```

    }
}

```

Cuando la función `search` se llama con el parámetro  $k$ , decide si incluir el elemento  $k$  en el subconjunto o no, y en ambos casos, luego se llama a sí misma con el parámetro  $k + 1$ . Sin embargo, si  $k = n$ , la función nota que todos los elementos se han procesado y se ha generado un subconjunto.

El siguiente árbol ilustra las llamadas a funciones cuando  $n = 3$ . Siempre podemos elegir la rama izquierda ( $k$  no está incluido en el subconjunto) o la rama derecha ( $k$  está incluido en el subconjunto).



## Método 2

Otra forma de generar subconjuntos se basa en la representación en bits de los enteros. Cada subconjunto de un conjunto de  $n$  elementos se puede representar como una secuencia de  $n$  bits, que corresponde a un entero entre  $0 \dots 2^n - 1$ . Los unos en la secuencia de bits indican qué elementos están incluidos en el subconjunto.

La convención habitual es que el último bit corresponde al elemento 0, el penúltimo bit corresponde al elemento 1, y así sucesivamente. Por ejemplo, la representación en bits de 25 es 11001, que corresponde al subconjunto  $\{0, 3, 4\}$ .

El siguiente código recorre los subconjuntos de un conjunto de  $n$  elementos

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}

```

El siguiente código muestra cómo podemos encontrar los elementos de un subconjunto que corresponde a una secuencia de bits. Al procesar cada subconjunto, el código construye un vector que contiene los elementos en el subconjunto.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
}

```



```

    for (int i = 0; i < n; i++) {
        if (b & (1 << i)) subset.push_back(i);
    }
}

```

## 5.2 Generación de permutaciones

A continuación, consideramos el problema de generar todas las permutaciones de un conjunto de  $n$  elementos. Por ejemplo, las permutaciones de  $\{0, 1, 2\}$  son  $(0, 1, 2)$ ,  $(0, 2, 1)$ ,  $(1, 0, 2)$ ,  $(1, 2, 0)$ ,  $(2, 0, 1)$  y  $(2, 1, 0)$ . De nuevo, hay dos enfoques: podemos usar recursión o recorrer las permutaciones iterativamente.

### Método 1

Al igual que los subconjuntos, las permutaciones se pueden generar usando recursión. La siguiente función `search` recorre las permutaciones del conjunto  $\{0, 1, \dots, n-1\}$ . La función construye un vector `permutation` que contiene la permutación, y la búsqueda comienza cuando la función se llama sin parámetros.

```

void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}

```

Cada llamada a la función agrega un nuevo elemento a `permutation`. La matriz `chosen` indica qué elementos ya están incluidos en la permutación. Si el tamaño de `permutation` es igual al tamaño del conjunto, se ha generado una permutación.

### Método 2

Otro método para generar permutaciones es comenzar con la permutación  $\{0, 1, \dots, n-1\}$  y repetir el uso de una función que construye la siguiente permutación en orden creciente. La biblioteca estándar de C++ contiene la función `next_permutation` que se puede usar para esto:

```

vector<int> permutation;
for (int i = 0; i < n; i++) {

```

```

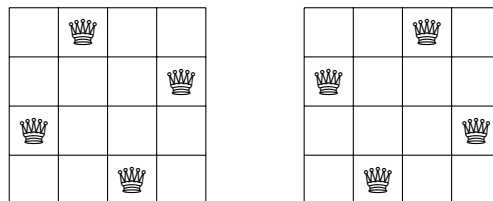
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));

```

## 5.3 Backtracking

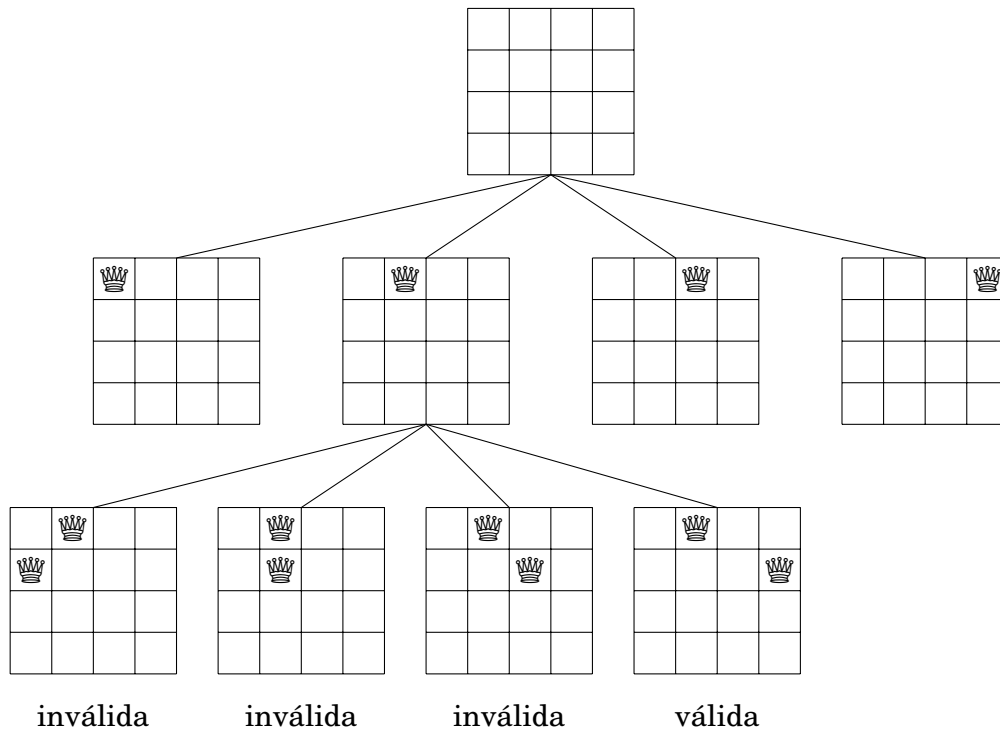
Un algoritmo de **backtracking** comienza con una solución vacía y extiende la solución paso a paso. La búsqueda recursivamente recorre todas las formas diferentes de cómo se puede construir una solución.

Como ejemplo, considere el problema de calcular el número de formas en que se pueden colocar  $n$  reinas en un tablero de ajedrez de  $n \times n$  de modo que ninguna de las  $n$  reinas se ataque entre sí. Por ejemplo, cuando  $n = 4$ , hay dos soluciones posibles:



El problema se puede resolver usando backtracking colocando reinas en el tablero fila por fila. Más precisamente, exactamente una reina se colocará en cada fila de modo que ninguna reina ataque a ninguna de las reinas colocadas antes. Se ha encontrado una solución cuando todas las  $n$  reinas se han colocado en el tablero.

Por ejemplo, cuando  $n = 4$ , algunas soluciones parciales generadas por el algoritmo de backtracking son las siguientes:



En el nivel inferior, las tres primeras configuraciones son ilegales, porque las reinas se atacan entre sí. Sin embargo, la cuarta configuración es válida y puede extenderse a una solución completa colocando dos reinas más en el tablero. Solo hay una forma de colocar las dos reinas restantes.

El algoritmo se puede implementar de la siguiente manera:

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

La búsqueda comienza llamando a `search(0)`. El tamaño del tablero es  $n \times n$ , y el código calcula el número de soluciones a `count`.

El código asume que las filas y columnas del tablero están numeradas de 0 a  $n - 1$ . Cuando se llama a la función `search` con el parámetro  $y$ , coloca una reina en la fila  $y$  y luego se llama a sí misma con el parámetro  $y + 1$ . Luego, si  $y = n$ , se ha encontrado una solución y la variable `count` se incrementa en uno.

El arreglo `column` lleva un registro de las columnas que contienen una reina, y los arreglos `diag1` y `diag2` llevan un registro de las diagonales. No está permitido agregar otra reina a una columna o diagonal que ya contiene una reina. Por

ejemplo, las columnas y diagonales de el tablero de  $4 \times 4$  se numeran de la siguiente manera:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

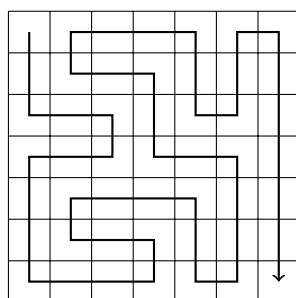
diag2

Sea  $q(n)$  el número de formas de colocar  $n$  reinas en un tablero de ajedrez de  $n \times n$ . El algoritmo de retroceso anterior nos dice que, por ejemplo,  $q(8) = 92$ . Cuando  $n$  aumenta, la búsqueda se vuelve rápidamente lenta, porque el número de soluciones aumenta exponencialmente. Por ejemplo, calcular  $q(16) = 14772512$  utilizando el algoritmo anterior ya lleva aproximadamente un minuto en una computadora moderna<sup>1</sup>.

## 5.4 Poda de la búsqueda

A menudo podemos optimizar el retroceso podando el árbol de búsqueda. La idea es agregar "inteligencia" al algoritmo para que se dé cuenta lo antes posible si una solución parcial no se puede extender a una solución completa. Estas optimizaciones pueden tener un tremendo efecto en la eficiencia de la búsqueda.

Consideremos el problema de calcular el número de caminos en una cuadrícula de  $n \times n$  desde la esquina superior izquierda hasta la esquina inferior derecha, de modo que el camino visite cada cuadrado exactamente una vez. Por ejemplo, en una cuadrícula de  $7 \times 7$ , hay 111712 caminos de este tipo. Uno de los caminos es el siguiente:



Nos centramos en el caso  $7 \times 7$ , porque su nivel de dificultad es apropiado para nuestras necesidades. Comenzamos con un algoritmo de retroceso directo, y luego lo optimizamos paso a paso usando observaciones de cómo se puede podar la búsqueda. Después de cada optimización, medimos el tiempo de ejecución del algoritmo y el número de llamadas recursivas, para que veamos claramente el efecto de cada optimización en la eficiencia de la búsqueda.

<sup>1</sup>No hay una forma conocida para calcular eficientemente valores mayores de  $q(n)$ . El récord actual es  $q(27) = 234907967154122528$ , calculado en 2016 [55].

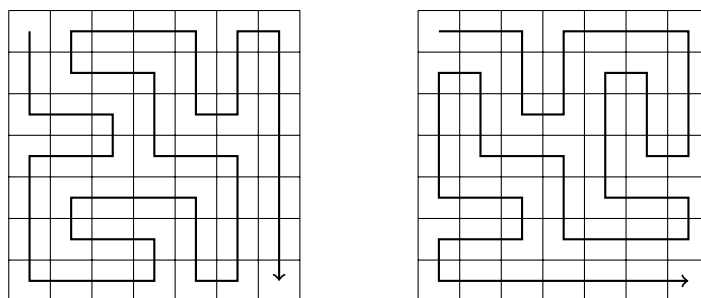
## Algoritmo básico

La primera versión del algoritmo no contiene ninguna optimización. Simplemente usamos retroceso para generar todos los caminos posibles desde la esquina superior izquierda hasta la esquina inferior derecha y contamos el número de estos caminos.

- tiempo de ejecución: 483 segundos
- número de llamadas recursivas: 76 mil millones

## Optimización 1

En cualquier solución, primero movemos un paso hacia abajo o hacia la derecha. Siempre hay dos caminos que son simétricos respecto a la diagonal de la cuadrícula después del primer paso. Por ejemplo, los siguientes caminos son simétricos:

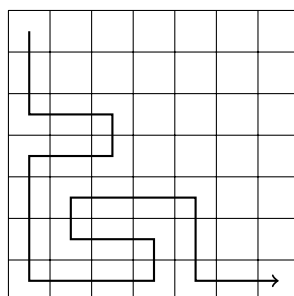


Por lo tanto, podemos decidir que siempre nos movemos primero un paso hacia abajo (o hacia la derecha), y finalmente multiplicamos el número de soluciones por dos.

- tiempo de ejecución: 244 segundos
- número de llamadas recursivas: 38 mil millones

## Optimización 2

Si el camino llega a la casilla inferior derecha antes de haber visitado todas las demás casillas de la cuadrícula, es claro que no será posible completar la solución. Un ejemplo de esto es el siguiente camino:

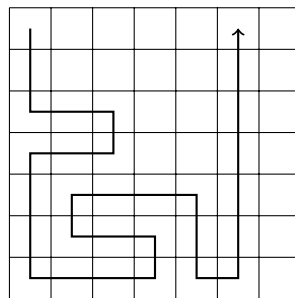


Usando esta observación, podemos terminar la búsqueda inmediatamente si llegamos a la casilla inferior derecha demasiado pronto.

- tiempo de ejecución: 119 segundos
- número de llamadas recursivas: 20 mil millones

### Optimización 3

Si el camino toca una pared y puede girar a la izquierda o a la derecha, la cuadrícula se divide en dos partes que contienen casillas no visitadas. Por ejemplo, en la siguiente situación, el camino puede girar a la izquierda o a la derecha:

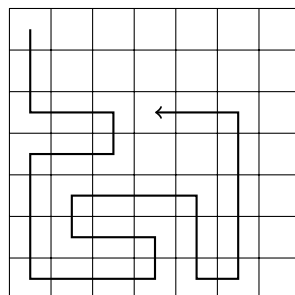


En este caso, ya no podemos visitar todas las casillas, por lo que podemos terminar la búsqueda. Esta optimización es muy útil:

- tiempo de ejecución: 1,8 segundos
- número de llamadas recursivas: 221 millones

### Optimización 4

La idea de la Optimización 3 se puede generalizar: si el camino no puede continuar hacia adelante pero puede girar a la izquierda o a la derecha, la cuadrícula se divide en dos partes que ambas contienen casillas no visitadas. Por ejemplo, considere el siguiente camino:



Está claro que ya no podemos visitar todas las casillas, por lo que podemos terminar la búsqueda. Después de esta optimización, la búsqueda es muy eficiente:

- tiempo de ejecución: 0.6 segundos

- número de llamadas recursivas: 69 millones

Ahora es un buen momento para dejar de optimizar el algoritmo y ver lo que hemos logrado. El tiempo de ejecución del algoritmo original era de 483 segundos, y ahora después de las optimizaciones, el tiempo de ejecución es de solo 0.6 segundos. Por lo tanto, el algoritmo se volvió casi 1000 veces más rápido después de las optimizaciones.

Este es un fenómeno habitual en el backtracking, porque el árbol de búsqueda suele ser grande e incluso observaciones simples pueden podar la búsqueda de forma eficaz. Son especialmente útiles las optimizaciones que se producen durante los primeros pasos del algoritmo, es decir, en la parte superior del árbol de búsqueda.

## 5.5 Encuentro en el medio

**Encuentro en el medio** es una técnica en la que el espacio de búsqueda se divide en dos partes de aproximadamente igual tamaño. Se realiza una búsqueda separada para ambas partes, y finalmente se combinan los resultados de las búsquedas.

La técnica se puede utilizar si hay una forma eficiente de combinar los resultados de las búsquedas. En tal situación, las dos búsquedas pueden requerir menos tiempo que una búsqueda grande. Típicamente, podemos convertir un factor de  $2^n$  en un factor de  $2^{n/2}$  usando la técnica de encuentro en el medio.

Como ejemplo, considere un problema en el que se nos da una lista de  $n$  números y un número  $x$ , y queremos saber si es posible elegir algunos números de la lista para que su suma sea  $x$ . Por ejemplo, dada la lista  $[2, 4, 5, 9]$  y  $x = 15$ , podemos elegir los números  $[2, 4, 9]$  para obtener  $2 + 4 + 9 = 15$ . Sin embargo, si  $x = 10$  para la misma lista, no es posible formar la suma.

Un algoritmo simple para el problema es recorrer todos los subconjuntos de los elementos y comprobar si la suma de alguno de los subconjuntos es  $x$ . El tiempo de ejecución de tal algoritmo es  $O(2^n)$ , porque hay  $2^n$  subconjuntos. Sin embargo, utilizando la técnica de encuentro en el medio, podemos lograr un algoritmo más eficiente de tiempo  $O(2^{n/2})^2$ . Tenga en cuenta que  $O(2^n)$  y  $O(2^{n/2})$  son diferentes complejidades porque  $2^{n/2}$  es igual a  $\sqrt{2^n}$ .

La idea es dividir la lista en dos listas  $A$  y  $B$  de modo que ambas listas contengan aproximadamente la mitad de los números. La primera búsqueda genera todos los subconjuntos de  $A$  y almacena sus sumas en una lista  $S_A$ . De manera correspondiente, la segunda búsqueda crea una lista  $S_B$  de  $B$ . Después de esto, basta con comprobar si es posible elegir un elemento de  $S_A$  y otro elemento de  $S_B$  de modo que su suma sea  $x$ . Esto es posible exactamente cuando hay una forma de formar la suma  $x$  usando los números de la lista original.

Por ejemplo, suponga que la lista es  $[2, 4, 5, 9]$  y  $x = 15$ . Primero, dividimos la lista en  $A = [2, 4]$  y  $B = [5, 9]$ . Después de esto, creamos listas  $S_A = [0, 2, 4, 6]$

---

<sup>2</sup>Esta idea fue introducida en 1974 por E. Horowitz y S. Sahni [39].

y  $S_B = [0, 5, 9, 14]$ . En este caso, la suma  $x = 15$  es posible de formar, porque  $S_A$  contiene la suma 6,  $S_B$  contiene la suma 9, y  $6 + 9 = 15$ . Esto corresponde a la solución  $[2, 4, 9]$ .

Podemos implementar el algoritmo de modo que su complejidad temporal sea  $O(2^{n/2})$ . Primero, generamos listas *ordenadas*  $S_A$  y  $S_B$ , lo que se puede hacer en tiempo  $O(2^{n/2})$  utilizando una técnica similar a la fusión. Después de esto, dado que las listas están ordenadas, podemos comprobar en tiempo  $O(2^{n/2})$  si la suma  $x$  se puede crear a partir de  $S_A$  y  $S_B$ .



# Chapter 6

## Algoritmos voraces

Un **algoritmo voraz** construye una solución al problema tomando siempre la decisión que parece mejor en ese momento. Un algoritmo voraz nunca revoca sus decisiones, sino que construye directamente la solución final. Por esta razón, los algoritmos voraces suelen ser muy eficientes.

La dificultad en el diseño de algoritmos voraces es encontrar una estrategia voraz que siempre produzca una solución óptima al problema. Las decisiones localmente óptimas en un algoritmo voraz también deberían ser globalmente óptimas. A menudo es difícil argumentar que un algoritmo voraz funciona.

### 6.1 Problema de la moneda

Como primer ejemplo, consideramos un problema en el que se nos da un conjunto de monedas y nuestra tarea es formar una suma de dinero  $n$  usando las monedas. Los valores de las monedas son  $\text{coins} = \{c_1, c_2, \dots, c_k\}$ , y cada moneda se puede usar tantas veces como queramos. ¿Cuál es el número mínimo de monedas necesarias?

Por ejemplo, si las monedas son las monedas de euro (en céntimos)

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

y  $n = 520$ , necesitamos al menos cuatro monedas. La solución óptima es seleccionar monedas  $200 + 200 + 100 + 20$  cuya suma es 520.

#### Algoritmo voraz

Un algoritmo voraz simple para el problema siempre selecciona la moneda más grande posible, hasta que se ha construido la suma de dinero requerida. Este algoritmo funciona en el caso de ejemplo, porque primero seleccionamos dos monedas de 200 céntimos, luego una moneda de 100 céntimos y finalmente una moneda de 20 céntimos. ¿Pero este algoritmo siempre funciona?

Resulta que si las monedas son las monedas de euro, el algoritmo voraz *siempre* funciona, es decir, siempre produce una solución con el menor número posible de monedas. La corrección del algoritmo se puede mostrar como sigue:

Primero, cada moneda 1, 5, 10, 50 y 100 aparece como máximo una vez en una solución óptima, porque si la solución contendría dos de esas monedas, podríamos reemplazarlas por una moneda y obtener una mejor solución. Por ejemplo, si la solución contendría monedas 5 + 5, podríamos reemplazarlas por la moneda 10.

De la misma manera, las monedas 2 y 20 aparecen como máximo dos veces en una solución óptima, porque podríamos reemplazar monedas 2 + 2 + 2 por monedas 5 + 1 y monedas 20 + 20 + 20 por monedas 50 + 10. Además, una solución óptima no puede contener monedas 2 + 2 + 1 o 20 + 20 + 10, porque podríamos reemplazarlas por monedas 5 y 50.

Usando estas observaciones, podemos mostrar para cada moneda  $x$  que no es posible construir óptimamente una suma  $x$  o cualquier suma mayor usando solo monedas que son más pequeñas que  $x$ . Por ejemplo, si  $x = 100$ , la suma óptima más grande usando las monedas más pequeñas es  $50 + 20 + 20 + 5 + 2 + 2 = 99$ . Por lo tanto, el algoritmo voraz que siempre selecciona la moneda más grande produce la solución óptima.

Este ejemplo muestra que puede ser difícil argumentar que un algoritmo voraz funciona, incluso si el algoritmo en sí es simple.

## Caso general

En el caso general, el conjunto de monedas puede contener cualquier moneda y el algoritmo voraz *no* produce necesariamente una solución óptima.

Podemos probar que un algoritmo voraz no funciona mostrando un contraejemplo donde el algoritmo da una respuesta incorrecta. En este problema podemos encontrar fácilmente un contraejemplo: si las monedas son  $\{1, 3, 4\}$  y la suma objetivo es 6, el algoritmo voraz produce la solución  $4 + 1 + 1$  mientras que la solución óptima es  $3 + 3$ .

No se sabe si el problema general de la moneda se puede resolver usando algún algoritmo voraz<sup>1</sup>. Sin embargo, como veremos en el Capítulo 7, en algunos casos, el problema general se puede resolver de manera eficiente utilizando un algoritmo de programación dinámica que siempre da la respuesta correcta.

## 6.2 Planificación

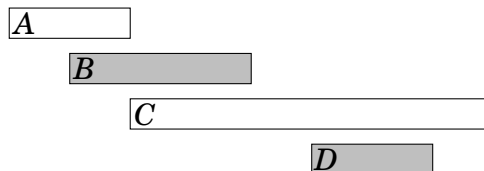
Muchos problemas de planificación se pueden resolver usando algoritmos voraces. Un problema clásico es el siguiente: Dado  $n$  eventos con sus tiempos de inicio y fin, encuentra un horario que incluya tantos eventos como sea posible. No es posible seleccionar un evento parcialmente. Por ejemplo, considere los siguientes eventos:

---

<sup>1</sup>Sin embargo, es posible *comprobar* en tiempo polinomial si el algoritmo voraz presentado en este capítulo funciona para un conjunto dado de monedas [53].

evento	tiempo de inicio	tiempo de fin
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

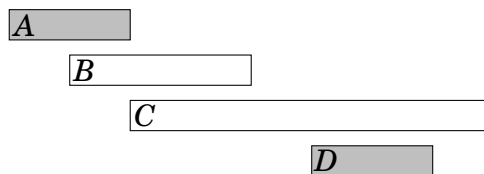
En este caso, el número máximo de eventos es dos. Por ejemplo, podemos seleccionar los eventos *B* y *D* de la siguiente manera:



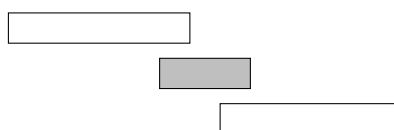
Es posible inventar varios algoritmos voraces para el problema, pero ¿cuál de ellos funciona en todos los casos?

## Algoritmo 1

La primera idea es seleccionar eventos tan *cortos* como sea posible. En el caso del ejemplo, este algoritmo selecciona los siguientes eventos:



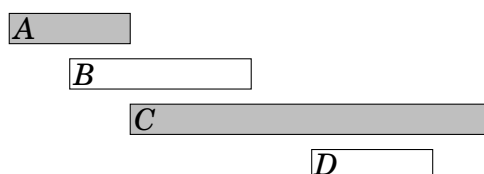
Sin embargo, seleccionar eventos cortos no siempre es una estrategia correcta. Por ejemplo, el algoritmo falla en el siguiente caso:



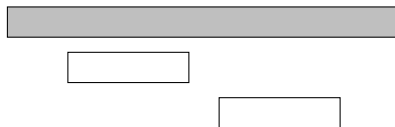
Si seleccionamos el evento corto, solo podemos seleccionar un evento. Sin embargo, sería posible seleccionar ambos eventos largos.

## Algoritmo 2

Otra idea es siempre seleccionar el siguiente posible evento que *comience* lo más *temprano* posible. Este algoritmo selecciona los siguientes eventos:



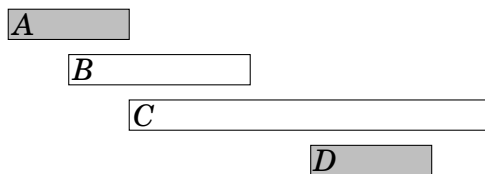
Sin embargo, podemos encontrar un contraejemplo también para este algoritmo. Por ejemplo, en el siguiente caso, el algoritmo solo selecciona un evento:



Si seleccionamos el primer evento, no es posible seleccionar ningún otro evento. Sin embargo, sería posible seleccionar los otros dos eventos.

### Algoritmo 3

La tercera idea es siempre seleccionar el siguiente evento posible que *termine* lo más *temprano* posible. Este algoritmo selecciona los siguientes eventos:



Resulta que este algoritmo *siempre* produce una solución óptima. La razón de esto es que siempre es una opción óptima seleccionar primero un evento que termine lo más temprano posible. Después de esto, es una opción óptima seleccionar el siguiente evento usando la misma estrategia, etc., hasta que no podamos seleccionar más eventos.

Una forma de argumentar que el algoritmo funciona es considerar qué sucede si primero seleccionamos un evento que termina más tarde que el evento que termina lo más temprano posible. Ahora, tendremos como máximo un número igual de opciones sobre cómo podemos seleccionar el siguiente evento. Por lo tanto, seleccionar un evento que termina más tarde nunca puede producir una mejor solución, y el algoritmo voraz es correcto.

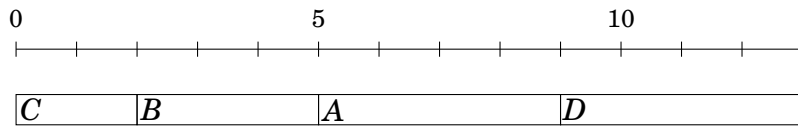
## 6.3 Tareas y plazos

Consideremos ahora un problema donde se nos dan  $n$  tareas con duraciones y plazos y nuestra tarea es elegir un orden para realizar las tareas. Para cada tarea, ganamos  $d - x$  puntos donde  $d$  es el plazo de la tarea y  $x$  es el momento en que terminamos la tarea. ¿Cuál es la mayor puntuación total posible que podemos obtener?

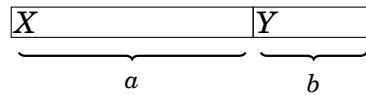
Por ejemplo, supongamos que las tareas son las siguientes:

tarea	duración	plazo
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

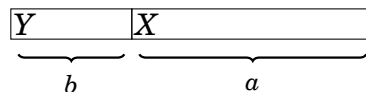
En este caso, un programa óptimo para las tareas es el siguiente:



En esta solución,  $C$  produce 5 puntos,  $B$  produce 0 puntos,  $A$  produce  $-7$  puntos y  $D$  produce  $-8$  puntos, por lo que la puntuación total es  $-10$ . Sorprendentemente, la solución óptima al problema no depende en absoluto de las fechas límite, sino que una estrategia codiciosa correcta es simplemente realizar las tareas *ordenadas por sus duraciones* en orden creciente. La razón de esto es que si alguna vez realizamos dos tareas una después de la otra de modo que la primera tarea dure más que la segunda tarea, podemos obtener una mejor solución si intercambiamos las tareas. Por ejemplo, considere el siguiente horario:



Aquí  $a > b$ , por lo que debemos intercambiar las tareas:



Ahora  $X$  da  $b$  puntos menos e  $Y$  da  $a$  puntos más, por lo que la puntuación total aumenta en  $a - b > 0$ . En una solución óptima, para dos tareas consecutivas, debe cumplirse que la tarea más corta venga antes de la tarea más larga. Por lo tanto, las tareas deben realizarse ordenadas por sus duraciones.

## 6.4 Minimizar sumas

A continuación, consideramos un problema en el que se nos dan  $n$  números  $a_1, a_2, \dots, a_n$  y nuestra tarea es encontrar un valor  $x$  que minimice la suma

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

Nos centramos en los casos  $c = 1$  y  $c = 2$ .

### Caso $c = 1$

En este caso, debemos minimizar la suma

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

Por ejemplo, si los números son  $[1, 2, 9, 2, 6]$ , la mejor solución es seleccionar  $x = 2$  que produce la suma

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

En el caso general, la mejor opción para  $x$  es la *mediana* de los números, es decir, el número del medio después de la ordenación. Por ejemplo, la lista  $[1, 2, 9, 2, 6]$  se convierte en  $[1, 2, 2, 6, 9]$  después de la ordenación, por lo que la mediana es 2.

La mediana es una opción óptima, porque si  $x$  es menor que la mediana, la suma se hace más pequeña al aumentar  $x$ , y si  $x$  es mayor que la mediana, la suma se hace más pequeña al disminuir  $x$ . Por lo tanto, la solución óptima es que  $x$  sea la mediana. Si  $n$  es par y hay dos medianas, tanto las medianas como todos los valores entre ellas son opciones óptimas.

## Caso $c = 2$

En este caso, debemos minimizar la suma

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

Por ejemplo, si los números son  $[1, 2, 9, 2, 6]$ , la mejor solución es seleccionar  $x = 4$  que produce la suma

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

En el caso general, la mejor opción para  $x$  es la *media* de los números. En el ejemplo, la media es  $(1 + 2 + 9 + 2 + 6)/5 = 4$ . Este resultado se puede derivar presentando la suma de la siguiente manera:

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

La última parte no depende de  $x$ , por lo que podemos ignorarla. Las partes restantes forman una función  $nx^2 - 2xs$  donde  $s = a_1 + a_2 + \cdots + a_n$ . Esta es una parábola que se abre hacia arriba con raíces  $x = 0$  y  $x = 2s/n$ , y el valor mínimo es la media de las raíces  $x = s/n$ , es decir, la media de los números  $a_1, a_2, \dots, a_n$ .

## 6.5 Compresión de datos

Un **código binario** asigna a cada carácter de una cadena una **palabra de código** que consiste en bits. Podemos *comprimir* la cadena usando el código binario reemplazando cada carácter por la palabra de código correspondiente. Por ejemplo, el siguiente código binario asigna palabras de código para los caracteres A–D:

carácter	palabra de código
A	00
B	01
C	10
D	11

Este es un código de **longitud constante** lo que significa que la longitud de cada palabra de código es la misma. Por ejemplo, podemos comprimir la cadena AABACDACA como sigue:

000001001011001000

Usando este código, la longitud de la cadena comprimida es de 18 bits. Sin embargo, podemos comprimir la cadena mejor si usamos un código de **longitud variable** donde las palabras de código pueden tener diferentes longitudes. Entonces podemos dar palabras de código cortas para los caracteres que aparecen a menudo y palabras de código largas para los caracteres que aparecen raramente. Resulta que un código **óptimo** para la cadena anterior es el siguiente:

carácter	palabra de código
A	0
B	110
C	10
D	111

Un código óptimo produce una cadena comprimida que es lo más corta posible. En este caso, la cadena comprimida usando el código óptimo es

001100101110100,

por lo que solo se necesitan 15 bits en lugar de 18 bits. Por lo tanto, gracias a un mejor código fue posible ahorrar 3 bits en la cadena comprimida.

Requerimos que ninguna palabra de código sea un prefijo de otra palabra de código. Por ejemplo, no está permitido que un código contenga ambas palabras de código 10 y 1011. La razón de esto es que queremos poder generar la cadena original a partir de la cadena comprimida. Si una palabra de código pudiera ser un prefijo de otra palabra de código, esto no siempre sería posible. Por ejemplo, el siguiente código no es *válido*:

carácter	palabra de código
A	10
B	11
C	1011
D	111

Usando este código, no sería posible saber si la cadena comprimida 1011 corresponde a la cadena AB o la cadena C.

## Codificación de Huffman

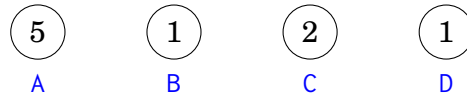
**Codificación de Huffman**<sup>2</sup> es un algoritmo voraz que construye un código óptimo para comprimir una cadena dada. El algoritmo construye un árbol binario basado en las frecuencias de los caracteres en la cadena, y la palabra de código de cada carácter se puede leer siguiendo un camino desde la raíz hasta el nodo correspondiente. Un movimiento hacia la izquierda corresponde al bit 0, y un movimiento hacia la derecha corresponde al bit 1.

Inicialmente, cada carácter de la cadena es representado por un nodo cuyo peso es el número de veces que el carácter aparece en la cadena. Luego, en cada

<sup>2</sup>D. A. Huffman descubrió este método al resolver una tarea de un curso universitario y publicó el algoritmo en 1952 [40].

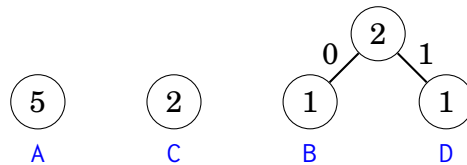
paso, dos nodos con pesos mínimos se combinan creando un nuevo nodo cuyo peso es la suma de los pesos de los nodos originales. El proceso continúa hasta que todos los nodos se han combinado.

A continuación, veremos cómo la codificación de Huffman crea el código óptimo para la cadena AABACDACA. Inicialmente, hay cuatro nodos que corresponden a los caracteres de la cadena:

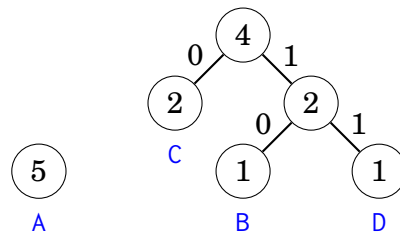


El nodo que representa el carácter A tiene peso 5 porque el carácter A aparece 5 veces en la cadena. Los otros pesos se han calculado de la misma manera.

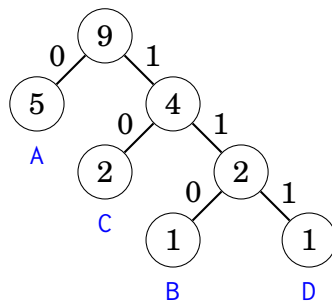
El primer paso es combinar los nodos que corresponden a los caracteres B y D, ambos con peso 1. El resultado es:



Después de esto, los nodos con peso 2 se combinan:



Finalmente, los dos nodos restantes se combinan:



Ahora todos los nodos están en el árbol, por lo que el código está listo. Las siguientes palabras de código se pueden leer del árbol:

carácter	palabra de código
A	0
B	110
C	10
D	111



# Chapter 7

## Programación dinámica

**Programación dinámica** es una técnica que combina la corrección de la búsqueda completa y la eficiencia de los algoritmos voraces. La programación dinámica se puede aplicar si el problema se puede dividir en subproblemas superpuestos que se pueden resolver independientemente.

Hay dos usos para la programación dinámica:

- **Encontrar una solución óptima:** Queremos encontrar una solución que sea lo más grande o lo más pequeña posible.
- **Contar el número de soluciones:** Queremos calcular el número total de soluciones posibles.

Primero veremos cómo la programación dinámica puede usarse para encontrar una solución óptima, y luego usaremos la misma idea para contar las soluciones.

Comprender la programación dinámica es un hito en la carrera de todo programador competitivo. Si bien la idea básica es simple, el desafío es cómo aplicar la programación dinámica a diferentes problemas. Este capítulo presenta un conjunto de problemas clásicos que son un buen punto de partida.

### 7.1 Problema de la moneda

Primero nos centramos en un problema que ya hemos visto en el Capítulo 6: Dado un conjunto de valores de monedas  $\text{coins} = \{c_1, c_2, \dots, c_k\}$  y una suma objetivo de dinero  $n$ , nuestra tarea es formar la suma  $n$  usando la menor cantidad de monedas posible.

En el Capítulo 6, resolvimos el problema usando un algoritmo voraz que siempre elige la más grande moneda posible. El algoritmo voraz funciona, por ejemplo, cuando las monedas son las monedas de euro, pero en el caso general el algoritmo voraz no produce necesariamente una solución óptima.

Ahora es el momento de resolver el problema de manera eficiente usando programación dinámica, para que el algoritmo funcione para cualquier conjunto de monedas. El algoritmo de programación dinámica se basa en una función recursiva que recorre todas las posibilidades de cómo formar la suma, como un algoritmo de fuerza bruta. Sin embargo, la programación dinámica algoritmo es

eficiente porque utiliza *memorización* y calcula la respuesta a cada subproblema solo una vez.

## Formulación recursiva

La idea en la programación dinámica es formular el problema de forma recursiva para que la solución al problema se pueda calcular a partir de soluciones a problemas más pequeños. En el problema de la moneda, un problema recursivo natural es el siguiente: ¿cuál es el menor número de monedas requerido para formar una suma  $x$ ?

Sea  $\text{solve}(x)$  denotar el mínimo número de monedas requeridas para una suma  $x$ . Los valores de la función dependen de los valores de las monedas. Por ejemplo, si  $\text{coins} = \{1, 3, 4\}$ , los primeros valores de la función son los siguientes:

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

Por ejemplo,  $\text{solve}(10) = 3$ , porque se necesitan al menos 3 monedas para formar la suma 10. La solución óptima es  $3 + 3 + 4 = 10$ .

La propiedad esencial de  $\text{solve}$  es que sus valores se pueden calcular recursivamente a partir de sus valores más pequeños. La idea es centrarse en la *primera* moneda que elegimos para la suma. Por ejemplo, en el escenario anterior, la primera moneda puede ser 1, 3 o 4. Si primero elegimos la moneda 1, la tarea restante es formar la suma 9 usando el mínimo número de monedas, que es un subproblema del problema original. Por supuesto, lo mismo se aplica a las monedas 3 y 4. Por lo tanto, podemos usar la siguiente fórmula recursiva para calcular el mínimo número de monedas:

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

El caso base de la recursión es  $\text{solve}(0) = 0$ , porque no se necesitan monedas para formar una suma vacía. Por ejemplo,

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

Ahora estamos listos para dar una función recursiva general que calcula el

mínimo número de monedas necesarias para formar una suma  $x$ :

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

Primero, si  $x < 0$ , el valor es  $\infty$ , porque es imposible formar una suma negativa de dinero. Luego, si  $x = 0$ , el valor es 0, porque no se necesitan monedas para formar una suma vacía. Finalmente, si  $x > 0$ , la variable  $c$  pasa por todas las posibilidades de cómo elegir la primera moneda de la suma.

Una vez que se ha encontrado una función recursiva que resuelve el problema, podemos implementar directamente una solución en C++ (la constante INF denota infinito):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Aún así, esta función no es eficiente, porque puede haber un número exponencial de formas de construir la suma. Sin embargo, a continuación veremos cómo hacer que la función sea eficiente utilizando una técnica llamada memorización.

## Usando memorización

La idea de la programación dinámica es usar **memorización** para calcular eficientemente los valores de una función recursiva. Esto significa que los valores de la función se almacenan en una matriz después de calcularlos. Para cada parámetro, el valor de la función se calcula recursivamente solo una vez, y después de esto, el valor se puede recuperar directamente de la matriz.

En este problema, usamos matrices

```
bool ready[N];
int value[N];
```

donde `ready[x]` indica si el valor de `solve(x)` ya se ha calculado, y si es así, `value[x]` contiene este valor. La constante  $N$  se ha elegido de modo que todos los valores necesarios quepan en las matrices.

Ahora la función se puede implementar eficientemente de la siguiente manera:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
```

```

    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}

```

La función maneja los casos base  $x < 0$  y  $x = 0$  como antes. Luego, la función verifica desde `ready[x]` si `solve(x)` ya se ha almacenado en `value[x]`, y si es así, la función la devuelve directamente. De lo contrario, la función calcula el valor de `solve(x)` recursivamente y lo almacena en `value[x]`.

Esta función funciona eficientemente, porque la respuesta para cada parámetro  $x$  se calcula recursivamente solo una vez. Una vez que un valor de `solve(x)` se ha almacenado en `value[x]`, se puede recuperar eficientemente cada vez que la función se vuelva a llamar con el parámetro  $x$ . La complejidad temporal del algoritmo es  $O(nk)$ , donde  $n$  es la suma objetivo y  $k$  es la cantidad de monedas.

Tenga en cuenta que también podemos *iterativamente* construir la matriz `value` utilizando un bucle que simplemente calcula todos los valores de `solve` para los parámetros  $0 \dots n$ :

```

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

```

De hecho, la mayoría de los programadores competitivos prefieren esta implementación, porque es más corta y tiene factores constantes más bajos. De ahora en adelante, también usamos implementaciones iterativas en nuestros ejemplos. Aún así, a menudo es más fácil pensar en soluciones de programación dinámica en términos de funciones recursivas.

## Construyendo una solución

A veces se nos pide que encontremos el valor de una solución óptima y que demos un ejemplo de cómo se puede construir una solución de este tipo. En el problema de la moneda, por ejemplo, podemos declarar otra matriz que indique para cada suma de dinero la primera moneda en una solución óptima:

```

int first[N];

```

Luego, podemos modificar el algoritmo de la siguiente manera:

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

Después de esto, el siguiente código se puede usar para imprimir las monedas que aparecen en una solución óptima para la suma  $n$ :

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

## Contando el número de soluciones

Consideremos ahora otra versión del problema de la moneda donde nuestra tarea es calcular el número total de formas de producir una suma  $x$  usando las monedas. Por ejemplo, si  $\text{coins} = \{1, 3, 4\}$  y  $x = 5$ , hay un total de 6 formas:

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

Nuevamente, podemos resolver el problema recursivamente. Sea  $\text{solve}(x)$  denotar el número de formas en que podemos formar la suma  $x$ . Por ejemplo, si  $\text{coins} = \{1, 3, 4\}$ , entonces  $\text{solve}(5) = 6$  y la fórmula recursiva es

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

Entonces, la función recursiva general es la siguiente:

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

Si  $x < 0$ , el valor es 0, porque no hay soluciones. Si  $x = 0$ , el valor es 1, porque solo hay una forma de formar una suma vacía. De lo contrario, calculamos la suma de todos los valores de la forma  $\text{solve}(x-c)$  donde  $c$  está en  $\text{coins}$ .

El siguiente código construye un arreglo `count` tal que `count[x]` es igual al valor de `solve(x)` para  $0 \leq x \leq n$ :

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x-c >= 0) {
            count[x] += count[x-c];
        }
    }
}
```

A menudo, el número de soluciones es tan grande que no es necesario calcular el número exacto, pero es suficiente dar la respuesta módulo  $m$  donde, por ejemplo,  $m = 10^9 + 7$ . Esto se puede hacer cambiando el código para que todos los cálculos se realicen módulo  $m$ . En el código anterior, basta con agregar la línea

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

después de la línea

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

Ahora hemos discutido todas las ideas básicas de la programación dinámica. Dado que la programación dinámica se puede utilizar en muchas situaciones diferentes, ahora veremos un conjunto de problemas que muestran más ejemplos sobre las posibilidades de la programación dinámica.


## 7.2 Subsecuencia creciente más larga

Nuestro primer problema es encontrar la **subsecuencia creciente más larga** en un arreglo de  $n$  elementos. Esta es una secuencia de longitud máxima de elementos del arreglo que va de izquierda a derecha, y cada elemento de la secuencia es mayor que el elemento anterior. Por ejemplo, en el arreglo

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

la subsecuencia creciente más larga contiene 4 elementos:

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



Sea  $\text{length}(k)$  denotar la longitud de la subsecuencia creciente más larga que termina en la posición  $k$ . Así, si calculamos todos los valores de  $\text{length}(k)$  donde

$0 \leq k \leq n - 1$ , descubriremos la longitud de la subsecuencia creciente más larga. Por ejemplo, los valores de la función para el arreglo anterior son los siguientes:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

Por ejemplo,  $\text{length}(6) = 4$ , porque la subsecuencia creciente más larga que termina en la posición 6 consta de 4 elementos.

Para calcular un valor de  $\text{length}(k)$ , debemos encontrar una posición  $i < k$  para la cual  $\text{array}[i] < \text{array}[k]$  y  $\text{length}(i)$  sea lo más grande posible. Entonces sabemos que  $\text{length}(k) = \text{length}(i) + 1$ , porque esta es una forma óptima de agregar  $\text{array}[k]$  a una subsecuencia. Sin embargo, si no existe tal posición  $i$ , entonces  $\text{length}(k) = 1$ , lo que significa que la subsecuencia solo contiene  $\text{array}[k]$ .

Dado que todos los valores de la función se pueden calcular a partir de sus valores más pequeños, podemos usar la programación dinámica. En el siguiente código, los valores de la función se almacenarán en un arreglo `length`.

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

Este código funciona en tiempo  $O(n^2)$ , porque consiste en dos bucles anidados. Sin embargo, también es posible implementar el cálculo de programación dinámica de forma más eficiente en tiempo  $O(n \log n)$ . ¿Puedes encontrar una manera de hacer esto?

## 7.3 Caminos en una cuadrícula

Nuestro próximo problema es encontrar un camino desde la esquina superior izquierda hasta la esquina inferior derecha de una cuadrícula de  $n \times n$ , de modo que solo nos movamos hacia abajo y hacia la derecha. Cada cuadrado contiene un entero positivo, y el camino debe construirse de modo que la suma de los valores a lo largo del camino sea lo más grande posible.

La siguiente imagen muestra un camino óptimo en una cuadrícula:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

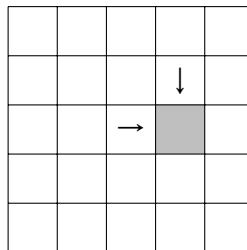
La suma de los valores en el camino es 67, y esta es la suma más grande posible en un camino desde la esquina superior izquierda hasta la esquina inferior derecha.

Asuma que las filas y columnas de la cuadrícula están numeradas de 1 a  $n$ , y  $\text{value}[y][x]$  es igual al valor del cuadrado  $(y, x)$ . Sea  $\text{sum}(y, x)$  la suma máxima en un camino desde la esquina superior izquierda hasta el cuadrado  $(y, x)$ . Ahora  $\text{sum}(n, n)$  nos dice la suma máxima desde la esquina superior izquierda hasta la esquina inferior derecha. Por ejemplo, en la cuadrícula de arriba,  $\text{sum}(5, 5) = 67$ .

Podemos calcular recursivamente las sumas de la siguiente manera:

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

La fórmula recursiva se basa en la observación de que un camino que termina en el cuadrado  $(y, x)$  puede venir ya sea del cuadrado  $(y, x - 1)$  o del cuadrado  $(y - 1, x)$ :



Por lo tanto, seleccionamos la dirección que maximiza la suma. Asumamos que  $\text{sum}(y, x) = 0$  si  $y = 0$  o  $x = 0$  (porque no existen tales caminos), por lo que la fórmula recursiva también funciona cuando  $y = 1$  o  $x = 1$ .

Dado que la función  $\text{sum}$  tiene dos parámetros, la matriz de programación dinámica también tiene dos dimensiones. Por ejemplo, podemos usar una matriz

```
int sum[N][N];
```

y calcular las sumas de la siguiente manera:

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

La complejidad temporal del algoritmo es  $O(n^2)$ .



## 7.4 Problemas de la mochila

El término **mochila** se refiere a problemas donde se da un conjunto de objetos, y se deben encontrar subconjuntos con algunas propiedades. Los problemas de la mochila a menudo se pueden resolver utilizando programación dinámica.

En esta sección, nos centraremos en el siguiente problema: Dado una lista de pesos  $[w_1, w_2, \dots, w_n]$ , determinar todas las sumas que se pueden construir utilizando los pesos. Por ejemplo, si los pesos son  $[1, 3, 3, 5]$ , las siguientes sumas son posibles:

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

En este caso, todas las sumas entre  $0 \dots 12$  son posibles, excepto 2 y 10. Por ejemplo, la suma 7 es posible porque podemos seleccionar los pesos  $[1, 3, 3]$ .

Para resolver el problema, nos centramos en subproblemas donde solo usamos los primeros  $k$  pesos para construir sumas. Sea  $\text{possible}(x, k) = \text{true}$  si podemos construir una suma  $x$  usando los primeros  $k$  pesos, y de lo contrario  $\text{possible}(x, k) = \text{false}$ . Los valores de la función se pueden calcular recursivamente como sigue:

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

La fórmula se basa en el hecho de que podemos usar o no usar el peso  $w_k$  en la suma. Si usamos  $w_k$ , la tarea restante es formar la suma  $x - w_k$  usando los primeros  $k - 1$  pesos, y si no usamos  $w_k$ , la tarea restante es formar la suma  $x$  usando los primeros  $k - 1$  pesos. Como casos base,

$$\text{possible}(x, 0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

porque si no se usan pesos, solo podemos formar la suma 0.

La siguiente tabla muestra todos los valores de la función para los pesos  $[1, 3, 3, 5]$  (el símbolo "X" indica los valores verdaderos):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

Después de calcular esos valores,  $\text{possible}(x, n)$  nos dice si podemos construir una suma  $x$  usando *todos* los pesos.

Sea  $W$  la suma total de los pesos. La siguiente solución de programación dinámica de tiempo  $O(nW)$  corresponde a la función recursiva:

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

Sin embargo, aquí hay una mejor implementación que solo usa una matriz unidimensional `possible[x]` que indica si podemos construir un subconjunto con suma  $x$ . El truco es actualizar la matriz de derecha a izquierda para cada nuevo peso:

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

Tenga en cuenta que la idea general presentada aquí se puede utilizar en muchos problemas de mochila. Por ejemplo, si se nos dan objetos con pesos y valores, podemos determinar para cada suma de peso la suma de valor máximo de un subconjunto.

## 7.5 Distancia de edición

La **distancia de edición** o **distancia de Levenshtein**<sup>1</sup> es el número mínimo de operaciones de edición necesarias para transformar una cadena en otra cadena. Las operaciones de edición permitidas son las siguientes:

- insertar un carácter (por ejemplo,  $ABC \rightarrow ABCA$ )
- eliminar un carácter (por ejemplo,  $ABC \rightarrow AC$ )
- modificar un carácter (por ejemplo,  $ABC \rightarrow ADC$ )

Por ejemplo, la distancia de edición entre LOVE y MOVIE es 2, porque primero podemos realizar la operación  $LOVE \rightarrow MOVE$  (modificar) y luego la operación  $MOVE \rightarrow MOVIE$  (insertar). Este es el menor número posible de operaciones, porque está claro que solo una operación no es suficiente.

Suponga que se nos da una cadena  $x$  de longitud  $n$  y una cadena  $y$  de longitud  $m$ , y queremos calcular la distancia de edición entre  $x$  y  $y$ . Para resolver el problema, definimos una función  $distance(a, b)$  que da la distancia de edición entre prefijos  $x[0 \dots a]$  y  $y[0 \dots b]$ . Por lo tanto, usando esta función, la distancia de

<sup>1</sup>La distancia recibe el nombre de V. I. Levenshtein quien la estudió en relación con los códigos binarios [49].

edición entre  $x$  y  $y$  es igual a  $\text{distance}(n-1, m-1)$ . Podemos calcular los valores de  $\text{distance}$  de la siguiente manera:

$$\begin{aligned} \text{distance}(a, b) = \min(&\text{distance}(a, b-1) + 1, \\ &\text{distance}(a-1, b) + 1, \\ &\text{distance}(a-1, b-1) + \text{cost}(a, b)). \end{aligned}$$

Aquí  $\text{cost}(a, b) = 0$  si  $x[a] = y[b]$ , y de lo contrario  $\text{cost}(a, b) = 1$ . La fórmula considera las siguientes formas de editar la cadena  $x$ :

- $\text{distance}(a, b-1)$ : insertar un carácter al final de  $x$
- $\text{distance}(a-1, b)$ : eliminar el último carácter de  $x$
- $\text{distance}(a-1, b-1)$ : coincidir o modificar el último carácter de  $x$

En los dos primeros casos, se necesita una operación de edición (insertar o eliminar). En el último caso, si  $x[a] = y[b]$ , podemos hacer coincidir los últimos caracteres sin editar, y de lo contrario se necesita una operación de edición (modificar).

La siguiente tabla muestra los valores de  $\text{distance}$  en el caso de ejemplo:

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

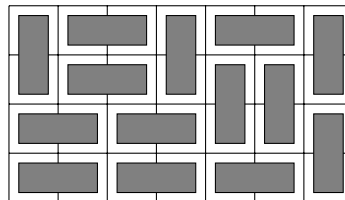
La esquina inferior derecha de la tabla nos dice que la distancia de edición entre LOVE y MOVIE es 2. La tabla también muestra cómo construir la secuencia más corta de operaciones de edición. En este caso, la ruta es la siguiente:

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

Los últimos caracteres de LOVE y MOVIE son iguales, por lo que la distancia de edición entre ellos es igual a la distancia de edición entre LOV y MOVI. Podemos usar una operación de edición para eliminar el carácter I de MOVI. Por lo tanto, la distancia de edición es uno mayor que la distancia de edición entre LOV y MOV, etc.

## 7.6 Contando mosaicos

A veces, los estados de una solución de programación dinámica son más complejos que las combinaciones fijas de números. Como ejemplo, considere el problema de calcular el número de maneras distintas de llenar una cuadrícula de  $n \times m$  usando baldosas de tamaño  $1 \times 2$  y  $2 \times 1$ . Por ejemplo, una solución válida para la cuadrícula de  $4 \times 7$  es



y el número total de soluciones es 781.

El problema se puede resolver usando programación dinámica recorriendo la cuadrícula fila por fila. Cada fila en una solución se puede representar como una cadena que contiene  $m$  caracteres del conjunto  $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$ . Por ejemplo, la solución anterior consta de cuatro filas que corresponden a las siguientes cadenas:

- $\sqcap \sqsubset \sqcap \sqsubset \sqsubset \sqcap$
- $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$

Sea  $\text{count}(k, x)$  el número de maneras de construir una solución para las filas  $1 \dots k$  de la cuadrícula tal que la cadena  $x$  corresponde a la fila  $k$ . Es posible usar programación dinámica aquí, porque el estado de una fila está limitado solo por el estado de la fila anterior.

Una solución es válida si la fila 1 no contiene el carácter  $\sqcup$ , la fila  $n$  no contiene el carácter  $\sqcap$ , y todas las filas consecutivas son *compatibles*. Por ejemplo, las filas  $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup$  y  $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$  son compatibles, mientras que las filas  $\sqcap \sqsubset \sqcap \sqsubset \sqsubset \sqcap$  y  $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$  no son compatibles.

Dado que una fila consta de  $m$  caracteres y hay cuatro opciones para cada carácter, el número de distintas filas es como máximo  $4^m$ . Por lo tanto, la complejidad temporal de la solución es  $O(n4^{2m})$  porque podemos recorrer las  $O(4^m)$  posibles estados para cada fila, y para cada estado, hay  $O(4^m)$  posibles estados para la fila anterior. En la práctica, es una buena idea rotar la cuadrícula para que el lado más corto tenga longitud  $m$ , porque el factor  $4^{2m}$  domina la complejidad temporal.

Es posible hacer que la solución sea más eficiente usando una representación más compacta para las filas. Resulta que es suficiente saber cuáles columnas de la fila anterior contienen el cuadrado superior de una baldosa vertical. Por lo tanto, podemos representar una fila usando solo caracteres  $\sqcap$  y  $\sqcup$ , donde  $\sqcup$  es una combinación de caracteres  $\sqcup$ ,  $\sqsubset$  y  $\sqsupset$ . Usando esta representación, solo hay  $2^m$  filas distintas y la complejidad temporal es  $O(n2^{2m})$ .

Como nota final, también existe una sorprendente fórmula directa para calcular el número de teselaciones<sup>2</sup>:

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

Esta fórmula es muy eficiente, porque calcula el número de teselaciones en  $O(nm)$  tiempo, pero dado que la respuesta es un producto de números reales, un problema al usar la fórmula es cómo almacenar los resultados intermedios con precisión.

---

<sup>2</sup>Sorprendentemente, esta fórmula fue descubierta en 1961 por dos equipos de investigación [43, 67] que trabajaron independientemente.



# Chapter 8

## Análisis amortizado

La complejidad temporal de un algoritmo a menudo es fácil de analizar simplemente examinando la estructura del algoritmo: qué bucles contiene el algoritmo y cuántas veces se ejecutan los bucles. Sin embargo, a veces un análisis directo no da una imagen real de la eficiencia del algoritmo.

**Análisis amortizado** se puede utilizar para analizar algoritmos que contienen operaciones cuya complejidad temporal varía. La idea es estimar el tiempo total utilizado para todas esas operaciones durante la ejecución del algoritmo, en lugar de centrarse en operaciones individuales.

### 8.1 Método de dos punteros

En el **método de dos punteros**, se utilizan dos punteros para iterar a través de los valores de la matriz. Ambos punteros pueden moverse solo en una dirección, lo que garantiza que el algoritmo funcione de manera eficiente. A continuación, analizaremos dos problemas que se pueden resolver utilizando el método de dos punteros.

#### Suma de subarreglos

Como primer ejemplo, considere un problema en el que se nos da un arreglo de  $n$  enteros positivos y una suma objetivo  $x$ , y queremos encontrar un subarreglo cuya suma sea  $x$  o informar que no existe tal subarreglo.

Por ejemplo, el arreglo

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

contiene un subarreglo cuya suma es 8:

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

Este problema se puede resolver en tiempo  $O(n)$  utilizando el método de dos punteros. La idea es mantener punteros que apunten al primer y último valor de un subarreglo. En cada turno, el puntero izquierdo se mueve un paso hacia la

derecha, y el puntero derecho se mueve hacia la derecha siempre que la suma del subarreglo resultante sea como máximo  $x$ . Si la suma se convierte exactamente en  $x$ , se ha encontrado una solución.

Como ejemplo, considere el siguiente arreglo y una suma objetivo  $x = 8$ :

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

El subarreglo inicial contiene los valores 1, 3 y 2 cuya suma es 6:

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

↑            ↑

Luego, el puntero izquierdo se mueve un paso hacia la derecha. El puntero derecho no se mueve, porque de lo contrario la suma del subarreglo excedería  $x$ .

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

↑            ↑

Nuevamente, el puntero izquierdo se mueve un paso hacia la derecha, y esta vez el puntero derecho se mueve tres pasos hacia la derecha. La suma del subarreglo es  $2 + 5 + 1 = 8$ , por lo que se ha encontrado un subarreglo cuya suma es  $x$ .

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

          ↑            ↑

El tiempo de ejecución del algoritmo depende de la cantidad de pasos que se mueve el puntero derecho. Si bien no existe un límite superior útil sobre cuántos pasos el puntero puede moverse en un *solo* turno, sabemos que el puntero se mueve *un total de*  $O(n)$  pasos durante el algoritmo, porque solo se mueve hacia la derecha.

Dado que tanto el puntero izquierdo como el derecho se mueven  $O(n)$  pasos durante el algoritmo, el algoritmo funciona en tiempo  $O(n)$ .

## Problema 2SUM

Otro problema que se puede resolver utilizando el método de los dos punteros es el siguiente problema, también conocido como el problema de **2SUM**: dada una matriz de  $n$  números y una suma objetivo  $x$ , encontrar dos valores de la matriz tales que su suma sea  $x$ , o informar que no existen tales valores.

Para resolver el problema, primero ordenamos los valores de la matriz en orden creciente. Después de eso, iteramos a través de la matriz usando dos punteros. El puntero izquierdo comienza en el primer valor y se mueve un paso a la derecha en cada vuelta. El puntero derecho comienza en el último valor y siempre se mueve hacia la izquierda hasta que la suma de los valores izquierdo y derecho es como máximo  $x$ . Si la suma es exactamente  $x$ , se ha encontrado una solución.

Por ejemplo, considere la siguiente matriz y una suma objetivo  $x = 12$ :



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

Las posiciones iniciales de los punteros son las siguientes. La suma de los valores es  $1 + 10 = 11$  que es menor que  $x$ .

1	4	5	6	7	9	9	10
↑							↑

Luego, el puntero izquierdo se mueve un paso a la derecha. El puntero derecho se mueve tres pasos a la izquierda, y la suma se convierte en  $4 + 7 = 11$ .

1	4	5	6	7	9	9	10
	↑			↑			

Después de esto, el puntero izquierdo se mueve un paso a la derecha de nuevo. El puntero derecho no se mueve, y una solución  $5 + 7 = 12$  ha sido encontrada.

1	4	5	6	7	9	9	10
		↑		↑			

El tiempo de ejecución del algoritmo es  $O(n \log n)$ , porque primero ordena la matriz en  $O(n \log n)$  tiempo, y luego ambos punteros se mueven  $O(n)$  pasos.

Tenga en cuenta que es posible resolver el problema de otra manera en  $O(n \log n)$  tiempo usando la búsqueda binaria. En tal solución, iteramos a través de la matriz y para cada valor de la matriz, tratamos de encontrar otro valor que produzca la suma  $x$ . Esto se puede hacer realizando  $n$  búsquedas binarias, cada una de las cuales tarda  $O(\log n)$  tiempo.

Un problema más difícil es el problema de **3SUM** que pide encontrar *tres* valores de la matriz cuya suma es  $x$ . Utilizando la idea del algoritmo anterior, este problema se puede resolver en  $O(n^2)$  tiempo<sup>1</sup>. ¿Puedes ver cómo?

## 8.2 Elementos más pequeños cercanos

El análisis amortizado se utiliza a menudo para estimar el número de operaciones realizadas en una estructura de datos. Las operaciones pueden estar distribuidas de forma desigual, de modo que la mayoría de las operaciones ocurren durante una cierta fase del algoritmo, pero el total número de operaciones es limitado.

Como ejemplo, considere el problema de encontrar para cada elemento de la matriz el **elemento más pequeño cercano**, es decir, el primer elemento más pequeño que precede al elemento en la matriz. Es posible que no exista tal elemento, en cuyo caso el algoritmo debe reportar esto. A continuación veremos cómo el problema puede ser resuelto eficientemente utilizando una estructura de pila.

<sup>1</sup>Durante mucho tiempo, se pensó que resolver el problema de 3SUM más eficientemente que en  $O(n^2)$  tiempo no sería posible. Sin embargo, en 2014, resultó [30] que este no es el caso.

Recorremos la matriz de izquierda a derecha y mantenemos una pila de elementos de la matriz. En cada posición de la matriz, eliminamos elementos de la pila hasta que el elemento superior sea menor que el elemento actual, o la pila esté vacía. Luego, informamos que el elemento superior es el elemento más pequeño cercano del elemento actual, o si la pila está vacía, no existe tal elemento. Finalmente, agregamos el elemento actual a la pila. Como ejemplo, considere la siguiente matriz:

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

Primero, los elementos 1, 3 y 4 se agregan a la pila, porque cada elemento es mayor que el elemento anterior. Por lo tanto, el elemento más pequeño cercano de 4 es 3, y el elemento más pequeño cercano de 3 es 1.

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

1 → 3 → 4

El siguiente elemento 2 es menor que los dos elementos superiores en la pila. Por lo tanto, los elementos 3 y 4 se eliminan de la pila, y luego el elemento 2 se agrega a la pila. Su elemento más pequeño cercano es 1:

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

1 —————→ 2

Luego, el elemento 5 es mayor que el elemento 2, por lo que se agregará a la pila, y su elemento más pequeño cercano es 2:

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

1 —————→ 2 → 5

Después de esto, el elemento 5 se elimina de la pila y los elementos 3 y 4 se agregan a la pila:

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

1 —————→ 2 —————→ 3 → 4

Finalmente, todos los elementos excepto 1 se eliminan de la pila y el último elemento 2 se agrega a la pila:

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

1 —————→ 2

La eficiencia del algoritmo depende de la cantidad total de operaciones de pila. Si el elemento actual es mayor que el elemento superior en la pila, se agrega directamente a la pila, lo cual es eficiente. Sin embargo, a veces la pila puede contener varios elementos más grandes y lleva tiempo eliminarlos. Aún así, cada elemento se agrega *exactamente una vez* a la pila y se elimina *como máximo una vez* de la pila. Por lo tanto, cada elemento provoca  $O(1)$  operaciones de pila, y el algoritmo funciona en tiempo  $O(n)$ .

### 8.3 Mínimo de ventana deslizante

Una **ventana deslizante** es una submatriz de tamaño constante que se mueve de izquierda a derecha a través de la matriz. En cada posición de la ventana, queremos calcular alguna información sobre los elementos dentro de la ventana. En esta sección, nos centramos en el problema de mantener el **mínimo de la ventana deslizante**, lo que significa que deberíamos reportar el valor más pequeño dentro de cada ventana.

El mínimo de la ventana deslizante se puede calcular usando una idea similar a la que usamos para calcular los elementos más cercanos más pequeños. Mantenemos una cola donde cada elemento es más grande que el elemento anterior, y el primer elemento siempre corresponde al elemento mínimo dentro de la ventana. Después de cada movimiento de la ventana, eliminamos elementos del final de la cola hasta que el último elemento de la cola sea más pequeño que el nuevo elemento de la ventana, o la cola se vacíe. También eliminamos el primer elemento de la cola si ya no está dentro de la ventana. Finalmente, añadimos el nuevo elemento de la ventana al final de la cola.

Como ejemplo, considere la siguiente matriz:

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

Supongamos que el tamaño de la ventana deslizante es 4. En la primera posición de la ventana, el valor más pequeño es 1:

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

1	→	4	→	5
---	---	---	---	---

Luego la ventana se mueve un paso a la derecha. El nuevo elemento 3 es más pequeño que los elementos 4 y 5 en la cola, por lo que los elementos 4 y 5 se eliminan de la cola y el elemento 3 se añade a la cola. El valor más pequeño sigue siendo 1.

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

1	→	3
---	---	---

Después de esto, la ventana se mueve de nuevo, y el elemento más pequeño 1 ya no pertenece a la ventana. Por lo tanto, se elimina de la cola y el más pequeño el valor ahora es 3. También el nuevo elemento 4 se añade a la cola.

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

3	→	4
---	---	---

El siguiente nuevo elemento 1 es más pequeño que todos los elementos en la cola. Por lo tanto, todos los elementos se eliminan de la cola y solo contendrá el elemento 1:

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

1

Finalmente la ventana alcanza su última posición. El elemento 2 se añade a la cola, pero el valor más pequeño dentro de la ventana sigue siendo 1.

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

1 → 2

Dado que cada elemento del arreglo se agrega a la cola exactamente una vez y se elimina de la cola como máximo una vez, el algoritmo funciona en tiempo  $O(n)$ .

# Chapter 9

## Consultas de rango

En este capítulo, discutimos las estructuras de datos que nos permiten procesar consultas de rango de manera eficiente. En una **consulta de rango**, nuestra tarea es calcular un valor basado en una submatriz de una matriz. Las consultas de rango típicas son:

- $\text{sum}_q(a, b)$ : calcular la suma de los valores en el rango  $[a, b]$
- $\text{min}_q(a, b)$ : encontrar el valor mínimo en el rango  $[a, b]$
- $\text{max}_q(a, b)$ : encontrar el valor máximo en el rango  $[a, b]$

Por ejemplo, considere el rango  $[3, 6]$  en la siguiente matriz:

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

En este caso,  $\text{sum}_q(3, 6) = 14$ ,  $\text{min}_q(3, 6) = 1$  y  $\text{max}_q(3, 6) = 6$ .

Una forma simple de procesar consultas de rango es usar un bucle que recorra todos los valores de la matriz en el rango. Por ejemplo, la siguiente función puede ser usada para procesar consultas de suma en una matriz:

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

Esta función funciona en tiempo  $O(n)$ , donde  $n$  es el tamaño de la matriz. Por lo tanto, podemos procesar  $q$  consultas en  $O(nq)$  tiempo usando la función. Sin embargo, si tanto  $n$  como  $q$  son grandes, este enfoque es lento. Afortunadamente, resulta que hay maneras de procesar consultas de rango mucho más eficientemente.

## 9.1 Consultas de matriz estática

Primero nos enfocamos en una situación donde la matriz es *estática*, es decir, los valores de la matriz nunca se actualizan entre las consultas. En este caso, es suficiente construir una estructura de datos estática que nos diga la respuesta para cualquier consulta posible.

### Consultas de suma

Podemos procesar fácilmente consultas de suma en una matriz estática construyendo una **matriz de suma de prefijos**. Cada valor en la matriz de suma de prefijos es igual a la suma de los valores en la matriz original hasta esa posición, es decir, el valor en la posición  $k$  es  $\text{sum}_q(0, k)$ . La matriz de suma de prefijos se puede construir en tiempo  $O(n)$ .

Por ejemplo, considere la siguiente matriz:

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

La matriz de suma de prefijos correspondiente es la siguiente:

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

Dado que la matriz de suma de prefijos contiene todos los valores de  $\text{sum}_q(0, k)$ , podemos calcular cualquier valor de  $\text{sum}_q(a, b)$  en tiempo  $O(1)$  de la siguiente manera:

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

Definiendo  $\text{sum}_q(0, -1) = 0$ , la fórmula anterior también se cumple cuando  $a = 0$ .

Por ejemplo, considere el rango  $[3, 6]$ :

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

En este caso  $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$ . Esta suma se puede calcular a partir de dos valores del arreglo de suma de prefijo:

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

Por lo tanto,  $\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$ .

También es posible generalizar esta idea a dimensiones superiores. Por ejemplo, podemos construir un arreglo de suma de prefijo bidimensional que se puede usar para calcular la suma de cualquier subarreglo rectangular en tiempo  $O(1)$ . Cada suma en tal arreglo corresponde a un subarreglo que comienza en la esquina superior izquierda del arreglo.

La siguiente imagen ilustra la idea:

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

La suma del subarreglo gris se puede calcular usando la fórmula

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

donde  $S(X)$  denota la suma de valores en un subarreglo rectangular desde la esquina superior izquierda hasta la posición de  $X$ .

## Consultas mínimas

Las consultas mínimas son más difíciles de procesar que las consultas de suma. Aún así, existe un método de preprocesamiento bastante simple de  $O(n \log n)$  tiempo después del cual podemos responder cualquier consulta mínima en tiempo  $O(1)$ <sup>1</sup>. Tenga en cuenta que dado que las consultas mínimas y máximas pueden procesarse de forma similar, podemos centrarnos en las consultas mínimas.

La idea es precalcular todos los valores de  $\min_q(a, b)$  donde  $b - a + 1$  (la longitud del rango) es una potencia de dos. Por ejemplo, para el arreglo

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

se calculan los siguientes valores:

$a$	$b$	$\min_q(a, b)$	$a$	$b$	$\min_q(a, b)$	$a$	$b$	$\min_q(a, b)$
0	0	1	0	1	1	0	3	1
1	1	3	1	2	3	1	4	3
2	2	4	2	3	4	2	5	1
3	3	8	3	4	6	3	6	1
4	4	6	4	5	1	4	7	1
5	5	1	5	6	1	0	7	1
6	6	4	6	7	2			
7	7	2						

<sup>1</sup>Esta técnica fue introducida en [7] y a veces se llama el método de la **tabla dispersa**. También hay técnicas más sofisticadas [22] donde el tiempo de preprocesamiento es solo  $O(n)$ , pero tales algoritmos no son necesarios en la programación competitiva.

El número de valores precalculados es  $O(n \log n)$ , porque hay  $O(\log n)$  longitudes de rango que son potencias de dos. Los valores se pueden calcular de manera eficiente usando la fórmula recursiva

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

donde  $b - a + 1$  es una potencia de dos y  $w = (b - a + 1)/2$ . Calcular todos esos valores toma  $O(n \log n)$  tiempo.

Después de esto, cualquier valor de  $\min_q(a, b)$  se puede calcular en tiempo  $O(1)$  como un mínimo de dos valores precalculados. Sea  $k$  la mayor potencia de dos que no excede  $b - a + 1$ . Podemos calcular el valor de  $\min_q(a, b)$  usando la fórmula

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

En la fórmula anterior, el rango  $[a, b]$  está representado como la unión de los rangos  $[a, a + k - 1]$  y  $[b - k + 1, b]$ , ambos de longitud  $k$ .

Como ejemplo, considere el rango  $[1, 6]$ :

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

La longitud del rango es 6, y la mayor potencia de dos que no excede 6 es 4. Por lo tanto, el rango  $[1, 6]$  es la unión de los rangos  $[1, 4]$  y  $[3, 6]$ :

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

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

Dado que  $\min_q(1, 4) = 3$  y  $\min_q(3, 6) = 1$ , concluimos que  $\min_q(1, 6) = 1$ .

## 9.2 Árbol indexado binario

Un **árbol indexado binario** o un **árbol de Fenwick**<sup>2</sup> se puede ver como una variante dinámica de una matriz de suma de prefijos. Soporta dos operaciones en tiempo  $O(\log n)$  en una matriz: procesar una consulta de suma de rango y actualizar un valor.

La ventaja de un árbol indexado binario es que nos permite actualizar de manera eficiente los valores de la matriz entre consultas de suma. Esto no sería posible usando una matriz de suma de prefijos, porque después de cada actualización, sería necesario construir la matriz de suma de prefijos completa nuevamente en tiempo  $O(n)$ .

<sup>2</sup>La estructura del árbol indexado binario fue presentada por P. M. Fenwick en 1994 [21].



## Estructura

Incluso si el nombre de la estructura es un *árbol* indexado binario, generalmente se representa como una matriz. En esta sección, asumimos que todas las matrices tienen índice uno, porque facilita la implementación.

Sea  $p(k)$  la mayor potencia de dos que divide  $k$ . Almacenamos un árbol indexado binario como una matriz  $tree$  tal que

$$tree[k] = \text{sum}_q(k - p(k) + 1, k),$$

es decir, cada posición  $k$  contiene la suma de los valores en un rango de la matriz original cuya longitud es  $p(k)$  y que termina en la posición  $k$ . Por ejemplo, dado que  $p(6) = 2$ ,  $tree[6]$  contiene el valor de  $\text{sum}_q(5, 6)$ .

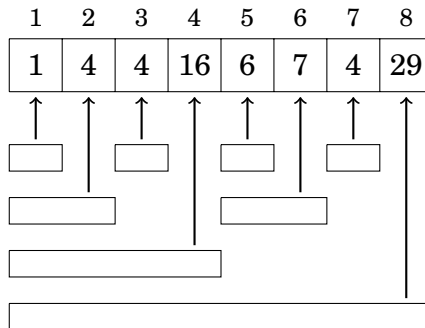
Por ejemplo, considere la siguiente matriz:

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

El árbol indexado binario correspondiente es el siguiente:

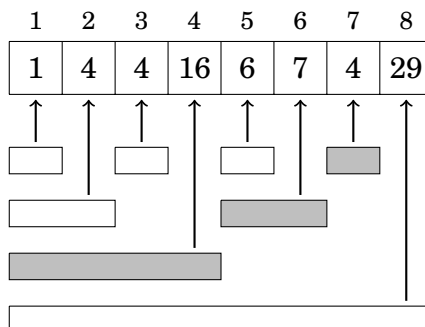
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

La siguiente imagen muestra más claramente cómo cada valor en el árbol indexado binario corresponde a un rango en la matriz original:



Usando un árbol indexado binario, cualquier valor de  $\text{sum}_q(1, k)$  se puede calcular en tiempo  $O(\log n)$ , porque un rango  $[1, k]$  siempre se puede dividir en  $O(\log n)$  rangos cuyas sumas están almacenadas en el árbol.

Por ejemplo, el rango  $[1, 7]$  consiste en los siguientes rangos:



Por lo tanto, podemos calcular la suma correspondiente de la siguiente manera:

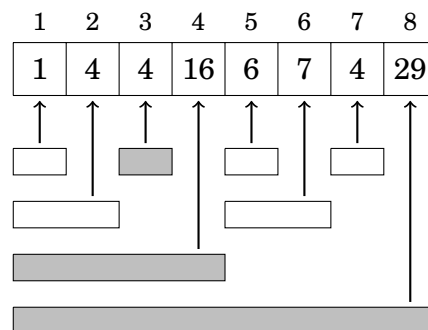
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

Para calcular el valor de  $\text{sum}_q(a, b)$  donde  $a > 1$ , podemos usar el mismo truco que usamos con las matrices de suma de prefijo:

$$\text{sum}_q(a, b) = \text{sum}_q(1, b) - \text{sum}_q(1, a - 1).$$

Dado que podemos calcular tanto  $\text{sum}_q(1, b)$  como  $\text{sum}_q(1, a - 1)$  en tiempo  $O(\log n)$ , la complejidad de tiempo total es  $O(\log n)$ .

Entonces, después de actualizar un valor en la matriz original, varios valores en el árbol indexado binario deberían actualizarse. Por ejemplo, si el valor en la posición 3 cambia, las sumas de los siguientes rangos cambian:



Dado que cada elemento de la matriz pertenece a  $O(\log n)$  rangos en el árbol indexado binario, es suficiente actualizar  $O(\log n)$  valores en el árbol.

## Implementación

Las operaciones de un árbol indexado binario se pueden implementar eficientemente utilizando operaciones de bits. El hecho clave que se necesita es que podemos calcular cualquier valor de  $p(k)$  utilizando la fórmula

$$p(k) = k \& -k.$$

La siguiente función calcula el valor de  $\text{sum}_q(1, k)$ :

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k&-k;
    }
    return s;
}
```

La siguiente función incrementa el valor del arreglo en la posición  $k$  por  $x$  ( $x$  puede ser positivo o negativo):

```

void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}

```

La complejidad temporal de ambas funciones es  $O(\log n)$ , porque las funciones acceden a  $O(\log n)$  valores en el árbol indexado binario, y cada movimiento a la siguiente posición toma  $O(1)$  tiempo.

## 9.3 Árbol de segmentos

Un **árbol de segmentos**<sup>3</sup> es una estructura de datos que admite dos operaciones: procesamiento de una consulta de rango y actualización de un valor del arreglo. Los árboles de segmentos pueden admitir consultas de suma, consultas de mínimo y máximo y muchas otras consultas para que ambas operaciones funcionen en  $O(\log n)$  tiempo.

En comparación con un árbol indexado binario, la ventaja de un árbol de segmentos es que es una estructura de datos más general. Mientras que los árboles indexados binarios solo admiten consultas de suma<sup>4</sup>, los árboles de segmentos también admiten otras consultas. Por otro lado, un árbol de segmentos requiere más memoria y es un poco más difícil de implementar.

### Estructura

Un árbol de segmentos es un árbol binario tal que los nodos en el nivel inferior del árbol corresponden a los elementos del arreglo, y los otros nodos contienen información necesaria para procesar consultas de rango.

En esta sección, asumimos que el tamaño del arreglo es una potencia de dos y se utiliza un índice basado en cero, porque es conveniente construir un árbol de segmentos para tal arreglo. Si el tamaño del arreglo no es una potencia de dos, siempre podemos agregar elementos adicionales a él.

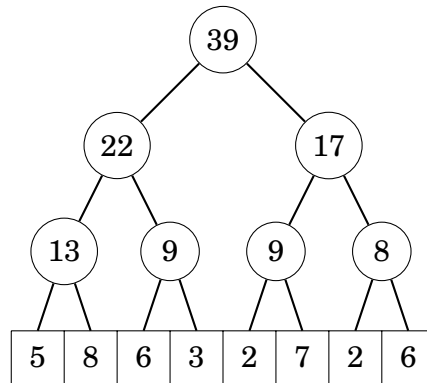
Primero discutiremos los árboles de segmentos que admiten consultas de suma. Como ejemplo, considere el siguiente arreglo:

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

El árbol de segmentos correspondiente es el siguiente:

<sup>3</sup>La implementación ascendente en este capítulo corresponde a la de [62]. Estructuras similares se usaron a finales de la década de 1970 para resolver problemas geométricos [9].

<sup>4</sup>De hecho, utilizando *dos* árboles indexados binarios es posible admitir consultas de mínimo [16], pero esto es más complicado que usar un árbol de segmentos.

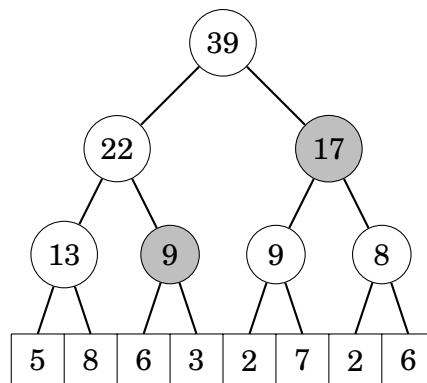


Cada nodo interno del árbol corresponde a un rango del arreglo cuyo tamaño es una potencia de dos. En el árbol anterior, el valor de cada interno nodo es la suma de los valores del arreglo correspondientes, y se puede calcular como la suma de los valores de su nodo hijo izquierdo y derecho.

Resulta que cualquier rango  $[a, b]$  se puede dividir en  $O(\log n)$  rangos cuyos valores se almacenan en nodos del árbol. Por ejemplo, considere el rango  $[2, 7]$ :

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

Aquí  $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$ . En este caso, los dos siguientes nodos del árbol corresponden al rango:

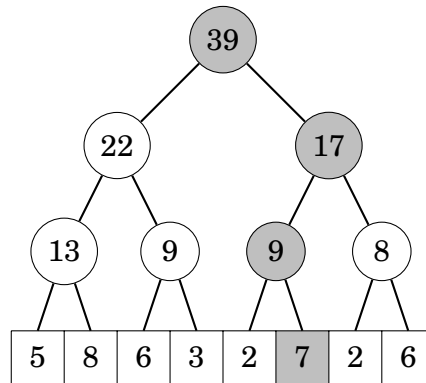


Por lo tanto, otra forma de calcular la suma es  $9 + 17 = 26$ .

Cuando la suma se calcula usando nodos ubicados lo más alto posible en el árbol, como máximo dos nodos en cada nivel del árbol son necesarios. Por lo tanto, el número total de nodos es  $O(\log n)$ .

Después de una actualización de la matriz, deberíamos actualizar todos los nodos cuyo valor depende del valor actualizado. Esto se puede hacer recorriendo la ruta desde el elemento de la matriz actualizado hasta el nodo superior y actualizando los nodos a lo largo de la ruta.

La siguiente imagen muestra qué nodos del árbol cambian si el valor de la matriz 7 cambia:

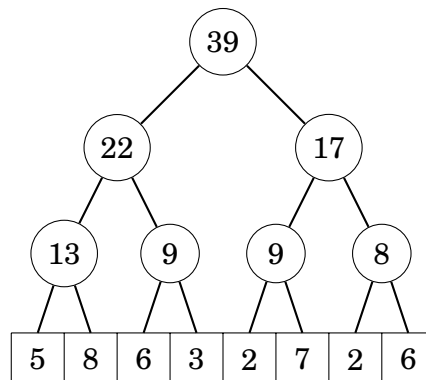


La ruta de abajo hacia arriba siempre consiste en  $O(\log n)$  nodos, por lo que cada actualización cambia  $O(\log n)$  nodos en el árbol.

## Implementación

Almacenamos un árbol de segmentos como una matriz de  $2n$  elementos donde  $n$  es el tamaño de la matriz original y una potencia de dos. Los nodos del árbol se almacenan de arriba hacia abajo:  $\text{tree}[1]$  es el nodo superior,  $\text{tree}[2]$  y  $\text{tree}[3]$  son sus hijos, y así sucesivamente. Finalmente, los valores de  $\text{tree}[n]$  a  $\text{tree}[2n - 1]$  corresponden a los valores de la matriz original en el nivel inferior del árbol.

Por ejemplo, el árbol de segmentos



se almacena de la siguiente manera:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Usando esta representación, el padre de  $\text{tree}[k]$  es  $\text{tree}[\lfloor k/2 \rfloor]$ , y sus hijos son  $\text{tree}[2k]$  y  $\text{tree}[2k + 1]$ . Tenga en cuenta que esto implica que la posición de un nodo es par si es un hijo izquierdo e impar si es un hijo derecho.

La siguiente función calcula el valor de  $\text{sum}_q(a, b)$ :

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
```

```

    if (a%2 == 1) s += tree[a++];
    if (b%2 == 0) s += tree[b--];
    a /= 2; b /= 2;
}
return s;
}

```

La función mantiene un rango que es inicialmente  $[a + n, b + n]$ . Luego, en cada paso, el rango se mueve un nivel más alto en el árbol, y antes de eso, los valores de los nodos que no pertenecen al rango superior se agregan a la suma.

La siguiente función aumenta el valor de la matriz en la posición  $k$  en  $x$ :

```

void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}

```

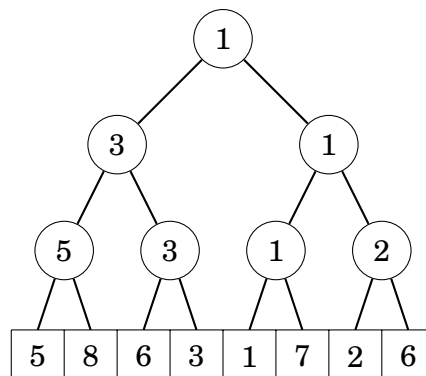
Primero, la función actualiza el valor en el nivel inferior del árbol. Después de esto, la función actualiza los valores de todos los nodos internos del árbol, hasta que llega al nodo superior del árbol.

Ambas funciones anteriores funcionan en  $O(\log n)$  tiempo, porque un árbol de segmentos de  $n$  elementos consta de  $O(\log n)$  niveles, y las funciones se mueven un nivel más alto en el árbol en cada paso.

## Otras consultas

Los árboles de segmentos pueden admitir todas las consultas de rango donde es posible dividir un rango en dos partes, calcular la respuesta por separado para ambas partes y luego combinar las respuestas de manera eficiente. Ejemplos de tales consultas son mínimo y máximo, máximo común divisor, y operaciones bit a bit y, o y xor.

Por ejemplo, el siguiente árbol de segmentos admite consultas mínimas:

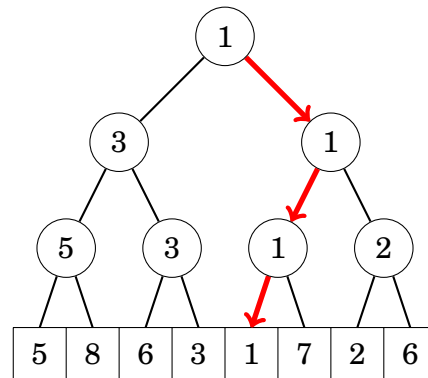


En este caso, cada nodo de árbol contiene el valor más pequeño en el correspondiente rango de la matriz. El nodo superior del árbol contiene el más pequeño valor

en toda la matriz. Las operaciones se pueden implementar como anteriormente, pero en lugar de sumas, se calculan los mínimos.

La estructura de un árbol de segmentos también nos permite usar la búsqueda binaria para localizar elementos de la matriz. Por ejemplo, si el árbol admite consultas mínimas, podemos encontrar la posición de un elemento con el valor más pequeño en  $O(\log n)$  tiempo.

Por ejemplo, en el árbol anterior, un elemento con el valor más pequeño 1 se puede encontrar atravesando un camino hacia abajo desde el nodo superior:



## 9.4 Técnicas adicionales

### Compresión de índice

Una limitación en las estructuras de datos que se basan en una matriz es que los elementos están indexados usando enteros consecutivos. Surgen dificultades cuando se necesitan índices grandes. Por ejemplo, si queremos usar el índice  $10^9$ , la matriz debe contener  $10^9$  elementos que requerirían demasiada memoria.

Sin embargo, a menudo podemos evitar esta limitación usando **compresión de índice**, donde los índices originales son reemplazados por índices 1, 2, 3, etc. Esto se puede hacer si conocemos todos los índices necesitados durante el algoritmo de antemano.

La idea es reemplazar cada índice original  $x$  con  $c(x)$  donde  $c$  es una función que comprime los índices. Requerimos que el orden de los índices no cambie, así que si  $a < b$ , entonces  $c(a) < c(b)$ . Esto nos permite realizar consultas convenientemente incluso si los índices están comprimidos.

Por ejemplo, si los índices originales son 555,  $10^9$  y 8, los nuevos índices son:

$$\begin{aligned} c(8) &= 1 \\ c(555) &= 2 \\ c(10^9) &= 3 \end{aligned}$$

### Actualizaciones de rango

Hasta ahora, hemos implementado estructuras de datos que admiten consultas de rango y actualizaciones de valores únicos. Consideremos ahora una situación

opuesta, donde deberíamos actualizar rangos y recuperar valores únicos. Nos enfocamos en una operación que aumenta todos los elementos en un rango  $[a, b]$  por  $x$ .

Sorprendentemente, podemos usar las estructuras de datos presentadas en este capítulo también en esta situación. Para hacer esto, construimos una **matriz de diferencias** cuyos valores indican las diferencias entre valores consecutivos en la matriz original. Por lo tanto, la matriz original es la matriz de suma de prefijo de la matriz de diferencias. Por ejemplo, considere la siguiente matriz:

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

La matriz de diferencias para la matriz anterior es la siguiente:

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

Por ejemplo, el valor 2 en la posición 6 del array original corresponde a la suma  $3 - 2 + 4 - 3 = 2$  en el array de diferencias.

La ventaja del array de diferencias es que podemos actualizar un rango en el array original cambiando solo dos elementos en el array de diferencias. Por ejemplo, si queremos aumentar los valores del array original entre las posiciones 1 y 4 en 5, basta con aumentar el valor del array de diferencias en la posición 1 en 5 y disminuir el valor en la posición 5 en 5. El resultado es el siguiente:

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

Más generalmente, para aumentar los valores en el rango  $[a, b]$  en  $x$ , aumentamos el valor en la posición  $a$  en  $x$  y disminuimos el valor en la posición  $b + 1$  en  $x$ . Por lo tanto, solo es necesario actualizar valores únicos y procesar consultas de suma, por lo que podemos usar un árbol indexado binario o un árbol de segmentos.

Un problema más difícil es soportar ambas consultas de rango y actualizaciones de rango. En el Capítulo 28 veremos que incluso esto es posible.



# Manipulación de bits

## 10.1 Representación de bits

103

Por ejemplo, la representación de bits de el número `int -43` es

**1111111111111111111111111010101.**

En una representación sin signo, solo no negativos los números se pueden usar, pero el límite superior para los valores es mayor. Una variable sin signo de  $n$  bits puede contener cualquier entero entre 0 y  $2^n - 1$ . Por ejemplo, en C++, una variable `unsigned int` puede contener cualquier entero entre 0 y  $2^{32} - 1$ .

Hay una conexión entre el representaciones: un número con signo  $-x$  es igual a un número sin signo  $2^n - x$ . Por ejemplo, el siguiente código muestra que el número con signo  $x = -43$  es igual al sin signo número  $y = 2^{32} - 43$ :

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Si un número es mayor que el límite superior de la representación de bits, el número se desbordará. En una representación con signo, el siguiente número después de  $2^{n-1} - 1$  es  $-2^{n-1}$ , y en una representación sin signo, el siguiente número después de  $2^n - 1$  es 0. Por ejemplo, considera el siguiente código:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Inicialmente, el valor de  $x$  es  $2^{31}-1$ . Este es el valor más grande que se puede almacenar en una variable `int`, por lo que el siguiente número después de  $2^{31}-1$  es  $-2^{31}$ .

## 10.2 Operaciones de bits

## Operación y

La operación  $x \& y$  produce un número que tiene unos en las posiciones donde ambos  $x$  e  $y$  tienen unos. Por ejemplo,  $22 \& 26 = 18$ , porque

$$\begin{array}{rcl} & 10110 & (22) \\ \& & 11010 & (26) \\ \hline = & 10010 & (18) \end{array}$$

Usando la operación  $\&$ , podemos verificar si un número  $x$  es par porque  $x \& 1 = 0$  si  $x$  es par, y  $x \& 1 = 1$  si  $x$  es impar. De manera más general,  $x$  es divisible por  $2^k$  exactamente cuando  $x \& (2^k - 1) = 0$ .

## Operación o

La operación **o**  $x \mid y$  produce un número que tiene unos en las posiciones donde al menos uno de  $x$  e  $y$  tienen unos. Por ejemplo,  $22 \mid 26 = 30$ , porque

$$\begin{array}{r} 10110 \quad (22) \\ \mid 11010 \quad (26) \\ \hline = 11110 \quad (30) \end{array}$$

## Operación xor

La operación **xor**  $x \wedge y$  produce un número que tiene unos en las posiciones donde exactamente uno de  $x$  e  $y$  tienen unos. Por ejemplo,  $22 \wedge 26 = 12$ , porque

$$\begin{array}{r} 10110 \quad (22) \\ \wedge 11010 \quad (26) \\ \hline = 01100 \quad (12) \end{array}$$

## Operación no

La operación **no**  $\sim x$  produce un número donde todos los bits de  $x$  han sido invertidos. La fórmula  $\sim x = -x - 1$  se cumple, por ejemplo,  $\sim 29 = -30$ .

El resultado de la operación no a nivel de bits depende de la longitud de la representación de bits, porque la operación invierte todos los bits. Por ejemplo, si los números son de 32 bits números int, el resultado es el siguiente:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000011101 \\ \sim x & = & -30 \quad 111111111111111111111111110010 \end{array}$$

## Desplazamientos de bits

El desplazamiento de bits a la izquierda  $x \ll k$  agrega  $k$  bits cero al número, y el desplazamiento de bits a la derecha  $x \gg k$  elimina los  $k$  últimos bits del número. Por ejemplo,  $14 \ll 2 = 56$ , porque 14 y 56 corresponden a 1110 y 111000. De manera similar,  $49 \gg 3 = 6$ , porque 49 y 6 corresponden a 110001 y 110.

Tenga en cuenta que  $x \ll k$  corresponde a multiplicar  $x$  por  $2^k$ , y  $x \gg k$  corresponde a dividir  $x$  por  $2^k$  redondeado hacia abajo a un entero.

## Aplicaciones

Un número de la forma  $1 \ll k$  tiene un bit uno en la posición  $k$  y todos los demás bits son cero, por lo que podemos usar esos números para acceder a bits únicos de números. En particular, el bit  $k$ -ésimo de un número es uno exactamente cuando  $x \& (1 \ll k)$  no es cero. El siguiente código imprime la representación de bits de un número int  $x$ :

```
for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}
```

También es posible modificar bits únicos de números usando ideas similares. Por ejemplo, la fórmula  $x \mid (1 \ll k)$  establece el bit  $k$ -ésimo de  $x$  a uno, la fórmula  $x \& \sim(1 \ll k)$  establece el bit  $k$ -ésimo de  $x$  a cero, y la fórmula  $x \wedge (1 \ll k)$  invierte el bit  $k$ -ésimo de  $x$ .

La fórmula  $x \& (x - 1)$  establece el último bit uno de  $x$  a cero, y la fórmula  $x \& -x$  establece todos los bits uno a cero, excepto el último bit uno. La fórmula  $x \mid (x - 1)$  invierte todos los bits después del último bit uno. También tenga en cuenta que un número positivo  $x$  es una potencia de dos exactamente cuando  $x \& (x - 1) = 0$ .

## Funciones adicionales

El compilador g++ proporciona las siguientes funciones para contar bits:

- `__builtin_clz(x)`: el número de ceros al principio del número
- `__builtin_ctz(x)`: el número de ceros al final del número
- `__builtin_popcount(x)`: el número de unos en el número
- `__builtin_parity(x)`: la paridad (par o impar) del número de unos

Las funciones se pueden usar de la siguiente manera:

```
int x = 5328; // 0000000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Si bien las funciones anteriores solo admiten números `int`, también hay versiones `long` `long` de las funciones disponibles con el sufijo `ll`.

## 10.3 Representar conjuntos

Cada subconjunto de un conjunto  $\{0, 1, 2, \dots, n - 1\}$  se puede representar como un entero de  $n$  bits cuyos bits uno indican qué elementos pertenecen al subconjunto. Esta es una forma eficiente de representar conjuntos, porque cada elemento solo requiere un bit de memoria, y las operaciones de conjunto se pueden implementar como operaciones de bits.

Por ejemplo, dado que `int` es un tipo de 32 bits, un número `int` puede representar cualquier subconjunto del conjunto  $\{0, 1, 2, \dots, 31\}$ . La representación de bits del conjunto  $\{1, 3, 4, 8\}$  es

000000000000000000000000100011010,

que corresponde al número  $2^8 + 2^4 + 2^3 + 2^1 = 282$ .

## Implementación de conjuntos

El siguiente código declara un `int` variable `x` que puede contener un subconjunto de  $\{0, 1, 2, \dots, 31\}$ . Después de esto, el código agrega los elementos 1, 3, 4 y 8 al conjunto e imprime el tamaño del conjunto.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Luego, el siguiente código imprime todos los elementos que pertenecen al conjunto:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// salida: 1 3 4 8
```

## Operaciones de conjunto

Las operaciones de conjunto se pueden implementar de la siguiente manera como operaciones de bits:

	sintaxis de conjunto	sintaxis de bits
intersección	$a \cap b$	$a \& b$
unión	$a \cup b$	$a   b$
complemento	$\bar{a}$	$\sim a$
diferencia	$a \setminus b$	$a \& (\sim b)$

Por ejemplo, el siguiente código primero construye los conjuntos  $x = \{1, 3, 4, 8\}$  e  $y = \{3, 6, 8, 9\}$ , y luego construye el conjunto  $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$ :

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

## Iterar a través de subconjuntos

El siguiente código recorre los subconjuntos de  $\{0, 1, \dots, n-1\}$ :

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

El siguiente código recorre los subconjuntos con exactamente  $k$  elementos:

```
for (int b = 0; b < (1<<n); b++) {  
    if (__builtin_popcount(b) == k) {  
        // process subset b  
    }  
}
```

El siguiente código recorre los subconjuntos de un conjunto  $x$ :

```
int b = 0;  
do {  
    // process subset b  
} while (b=(b-x)&x);
```

## 10.4 Optimizaciones de bits

Muchos algoritmos pueden ser optimizados usando operaciones de bits. Tales optimizaciones no cambian la complejidad temporal del algoritmo, pero pueden tener un gran impacto en el tiempo de ejecución real del código. En esta sección discutimos ejemplos de tales situaciones.

### Distancias de Hamming

La **distancia de Hamming**  $\text{hamming}(a, b)$  entre dos cadenas  $a$  y  $b$  de igual longitud es el número de posiciones en las que las cadenas difieren. Por ejemplo,

$$\text{hamming}(01101, 11001) = 2.$$

Considere el siguiente problema: Dada una lista de  $n$  cadenas de bits, cada una de longitud  $k$ , calcular la distancia de Hamming mínima entre dos cadenas en la lista. Por ejemplo, la respuesta para  $[00111, 01101, 11110]$  es 2, porque

- $\text{hamming}(00111, 01101) = 2$ ,
- $\text{hamming}(00111, 11110) = 3$ , y
- $\text{hamming}(01101, 11110) = 3$ .

Una forma sencilla de resolver el problema es recorrer todos los pares de cadenas y calcular sus distancias de Hamming, lo que da un algoritmo de tiempo  $O(n^2k)$ . La siguiente función se puede utilizar para calcular las distancias:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

Sin embargo, si  $k$  es pequeño, podemos optimizar el código almacenando las cadenas de bits como enteros y calculando las distancias de Hamming usando operaciones de bits. En particular, si  $k \leq 32$ , podemos simplemente almacenar las cadenas como valores `int` y usar la siguiente función para calcular las distancias:

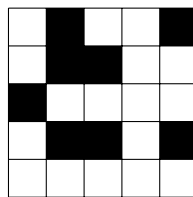
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

En la función anterior, la operación `xor` construye una cadena de bits que tiene unos en las posiciones en las que  $a$  y  $b$  difieren. Luego, el número de bits se calcula usando la función `__builtin_popcount`.

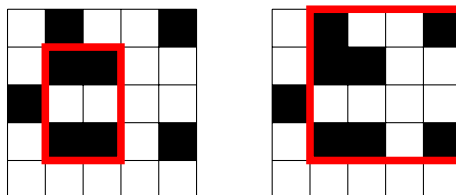
Para comparar las implementaciones, generamos una lista de 10000 cadenas de bits aleatorias de longitud 30. Usando el primer enfoque, la búsqueda tardó 13.5 segundos, y después de la optimización de bits, solo tardó 0.5 segundos. Por lo tanto, el código optimizado de bits fue casi 30 veces más rápido que el código original.

## Contando subrejillas

Como otro ejemplo, considere el siguiente problema: Dada una rejilla  $n \times n$  cuyas casillas son negras (1) o blancas (0), calcular el número de subrejillas cuyas esquinas son negras. Por ejemplo, la rejilla



contiene dos subrejillas:



Hay un algoritmo de tiempo  $O(n^3)$  para resolver el problema: recorrer todos los  $O(n^2)$  pares de filas y para cada par  $(a, b)$  calcular el número de columnas que

contienen una casilla negra en ambas filas en  $O(n)$  tiempo. El siguiente código asume que `color[y][x]` denota el color en la fila  $y$  y la columna  $x$ :

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Entonces, esas columnas representan  $\text{count}(\text{count} - 1)/2$  subrejillas con esquinas negras, porque podemos elegir dos de ellas para formar una subrejilla. Para optimizar este algoritmo, dividimos la cuadrícula en bloques de columnas de manera que cada bloque consista en  $N$  columnas consecutivas. Luego, cada fila se almacena como una lista de números de  $N$  bits que describen los colores de los cuadrados. Ahora podemos procesar  $N$  columnas al mismo tiempo usando operaciones bit a bit. En el siguiente código, `color[y][k]` representa un bloque de  $N$  colores como bits.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

El algoritmo resultante funciona en  $O(n^3/N)$  tiempo.

Generamos una cuadrícula aleatoria de tamaño  $2500 \times 2500$  y comparamos la implementación original y la optimizada con bits. Mientras que el código original tardó 29.6 segundos, la versión optimizada con bits solo tardó 3.1 segundos con  $N = 32$  (int numbers) y 1.7 segundos con  $N = 64$  (long long numbers).

## 10.5 Programación dinámica

Las operaciones bit a bit proporcionan una forma eficiente y conveniente de implementar algoritmos de programación dinámica cuyos estados contienen subconjuntos de elementos, porque estos estados se pueden almacenar como enteros. A continuación, discutimos ejemplos de cómo combinar operaciones bit a bit y programación dinámica.

### Selección óptima

Como primer ejemplo, considere el siguiente problema: Se nos dan los precios de  $k$  productos durante  $n$  días, y queremos comprar cada producto exactamente una vez. Sin embargo, solo podemos comprar como máximo un producto en un día. ¿Cuál es el precio total mínimo? Por ejemplo, considere el siguiente escenario ( $k = 3$  y  $n = 8$ ):



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

En este escenario, el precio total mínimo es 5:

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

Sea  $\text{price}[x][d]$  el precio del producto  $x$  en el día  $d$ . Por ejemplo, en el escenario anterior  $\text{price}[2][3] = 7$ . Luego, sea  $\text{total}(S, d)$  el precio total mínimo para comprar un subconjunto  $S$  de productos hasta el día  $d$ . Usando esta función, la solución al problema es  $\text{total}(\{0 \dots k-1\}, n-1)$ .

Primero,  $\text{total}(\emptyset, d) = 0$ , porque no cuesta nada comprar un conjunto vacío, y  $\text{total}(\{x\}, 0) = \text{price}[x][0]$ , porque hay una forma de comprar un producto el primer día. Luego, se puede usar la siguiente recurrencia:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

Esto significa que no compramos ningún producto el día  $d$  o compramos un producto  $x$  que pertenece a  $S$ . En el último caso, eliminamos  $x$  de  $S$  y agregamos el precio de  $x$  al precio total.

El siguiente paso es calcular los valores de la función usando programación dinámica. Para almacenar los valores de la función, declaramos una matriz

```
int total[1<<K][N];
```

donde  $K$  y  $N$  son constantes lo suficientemente grandes. La primera dimensión de la matriz corresponde a una bit representación de un subconjunto.

Primero, los casos donde  $d = 0$  se pueden procesar de la siguiente manera:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Luego, la recurrencia se traduce en el siguiente código:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s & (1<<x)) {
                total[s][d] = min(total[s][d],
```

```

total[s^(1<<x)][d-1]+price[x][d]];
    }
  }
}

```

La complejidad temporal del algoritmo es  $O(n2^k k)$ .

## De permutaciones a subconjuntos

Usando la programación dinámica, a menudo es posible cambiar una iteración sobre permutaciones a una iteración sobre subconjuntos<sup>1</sup>. El beneficio de esto es que  $n!$ , el número de permutaciones, es mucho mayor que  $2^n$ , el número de subconjuntos. Por ejemplo, si  $n = 20$ , entonces  $n! \approx 2.4 \cdot 10^{18}$  y  $2^n \approx 10^6$ . Por lo tanto, para ciertos valores de  $n$ , podemos recorrer los subconjuntos de manera eficiente, pero no las permutaciones.

Como ejemplo, considere el siguiente problema: Hay un ascensor con peso máximo  $x$ , y  $n$  personas con pesos conocidos que quieren ir desde la planta baja hasta la planta superior. ¿Cuál es el número mínimo de viajes necesarios si las personas entran al ascensor en un orden óptimo?

Por ejemplo, suponga que  $x = 10$ ,  $n = 5$  y los pesos son los siguientes:

persona	peso
0	2
1	3
2	3
3	5
4	6

En este caso, el número mínimo de viajes es 2. Un orden óptimo es  $\{0, 2, 3, 1, 4\}$ , que divide a las personas en dos viajes: primero  $\{0, 2, 3\}$  (peso total 10), y luego  $\{1, 4\}$  (peso total 9).

El problema se puede resolver fácilmente en tiempo  $O(n!n)$  probando todas las permutaciones posibles de  $n$  personas. Sin embargo, podemos usar la programación dinámica para obtener un algoritmo más eficiente de tiempo  $O(2^n n)$ . La idea es calcular para cada subconjunto de personas dos valores: el número mínimo de viajes necesarios y el peso mínimo de las personas que viajan en el último grupo.

Sea  $\text{weight}[p]$  el peso de la persona  $p$ . Definimos dos funciones:  $\text{rides}(S)$  es el número mínimo de viajes para un subconjunto  $S$ , y  $\text{last}(S)$  es el peso mínimo del último viaje. Por ejemplo, en el escenario anterior

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{y} \quad \text{last}(\{1, 3, 4\}) = 5,$$

porque los viajes óptimos son  $\{1, 4\}$  y  $\{3\}$ , y el segundo viaje tiene un peso de 5. Por supuesto, nuestro objetivo final es calcular el valor de  $\text{rides}(\{0 \dots n-1\})$ .

<sup>1</sup>Esta técnica fue introducida en 1962 por M. Held y R. M. Karp [34].

Podemos calcular los valores de las funciones recursivamente y luego aplicar la programación dinámica. La idea es recorrer todas las personas que pertenecen a  $S$  y elegir de manera óptima a la última persona  $p$  que ingresa al ascensor. Cada una de estas elecciones produce un subproblema para un subconjunto más pequeño de personas. Si  $\text{last}(S \setminus p) + \text{weight}[p] \leq x$ , podemos agregar  $p$  al último viaje. De lo contrario, tenemos que reservar un nuevo viaje que inicialmente solo contiene  $p$ .

Para implementar la programación dinámica, declaramos un array

```
pair<int,int> best[1<N];
```

que contiene para cada subconjunto  $S$  un par  $(\text{rides}(S), \text{last}(S))$ . Establecemos el valor para un grupo vacío de la siguiente manera:

```
best[0] = {1,0};
```

Luego, podemos rellenar el array de la siguiente manera:

```
for (int s = 1; s < (1<n); s++) {
    // valor inicial: se necesitan n+1 viajes
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<p)) {
            auto option = best[s^(1<p)];
            if (option.second+weight[p] <= x) {
                // agregar p a un viaje existente
                option.second += weight[p];
            } else {
                // reservar un nuevo viaje para p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Tenga en cuenta que el bucle anterior garantiza que para dos subconjuntos  $S_1$  y  $S_2$  tales que  $S_1 \subset S_2$ , procesamos  $S_1$  antes que  $S_2$ . Por lo tanto, los valores de la programación dinámica se calculan en el orden correcto.

## Contar subconjuntos

Nuestro último problema en este capítulo es el siguiente: Sea  $X = \{0 \dots n-1\}$ , y a cada subconjunto  $S \subset X$  se le asigna un entero  $\text{value}[S]$ . Nuestra tarea es calcular para cada  $S$

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

es decir, la suma de los valores de los subconjuntos de  $S$ .

Por ejemplo, suponga que  $n = 3$  y los valores son los siguientes:

- $\text{value}[\emptyset] = 3$
- $\text{value}[\{0\}] = 1$
- $\text{value}[\{1\}] = 4$
- $\text{value}[\{0, 1\}] = 5$
- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0, 2\}] = 1$
- $\text{value}[\{1, 2\}] = 3$
- $\text{value}[\{0, 1, 2\}] = 3$

En este caso, por ejemplo,

$$\begin{aligned} \text{sum}(\{0, 2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

Debido a que hay un total de  $2^n$  subconjuntos, una posible solución es recorrer todos los pares de subconjuntos en tiempo  $O(2^{2n})$ . Sin embargo, usando la programación dinámica, nosotros podemos resolver el problema en tiempo  $O(2^n n)$ . La idea es concentrarse en las sumas donde los elementos que se pueden eliminar de  $S$  están restringidos. Sea  $\text{partial}(S, k)$  la suma de los valores de los subconjuntos de  $S$  con la restricción de que solo los elementos  $0 \dots k$  pueden ser eliminados de  $S$ . Por ejemplo,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

porque solo podemos eliminar los elementos  $0 \dots 1$ . Podemos calcular los valores de  $\text{sum}$  usando los valores de  $\text{partial}$ , porque

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Los casos base para la función son

$$\text{partial}(S, -1) = \text{value}[S],$$

porque en este caso no se pueden eliminar elementos de  $S$ . Entonces, en el caso general podemos usar la siguiente recurrencia:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Aquí nos centramos en el elemento  $k$ . Si  $k \in S$ , tenemos dos opciones: podemos mantener  $k$  en  $S$  o eliminarlo de  $S$ .

Hay una forma particularmente inteligente de implementar el cálculo de sumas. Podemos declarar una matriz

```
int sum[1<<N];
```

que contendrá la suma de cada subconjunto. La matriz se inicializa de la siguiente manera:

```
for (int s = 0; s < (1<<n); s++) {  
    sum[s] = value[s];  
}
```

Luego, podemos llenar la matriz de la siguiente manera:

```
for (int k = 0; k < n; k++) {  
    for (int s = 0; s < (1<<n); s++) {  
        if (s & (1<<k)) sum[s] += sum[s ^ (1<<k)];  
    }  
}
```

Este código calcula los valores de  $\text{partial}(S, k)$  para  $k = 0 \dots n - 1$  en la matriz `sum`. Dado que  $\text{partial}(S, k)$  siempre se basa en  $\text{partial}(S, k - 1)$ , podemos reutilizar la matriz `sum`, lo que produce una implementación muy eficiente.



# **Part II**

## **Algoritmos gráficos**





# Chapter 11

## Conceptos básicos de grafos

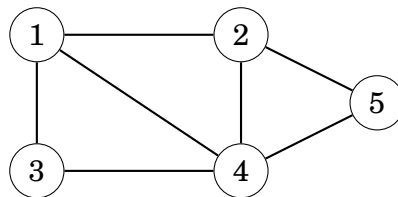
Muchos problemas de programación pueden resolverse modelando el problema como un problema de grafo y utilizando un algoritmo de grafo apropiado. Un ejemplo típico de un grafo es una red de carreteras y ciudades en un país. A veces, sin embargo, el grafo está oculto en el problema y puede ser difícil detectarlo.

Esta parte del libro discute los algoritmos de grafos, especialmente centrándose en temas que son importantes en la programación competitiva. En este capítulo, revisaremos conceptos relacionados con los grafos, y estudiaremos diferentes formas de representar grafos en algoritmos.

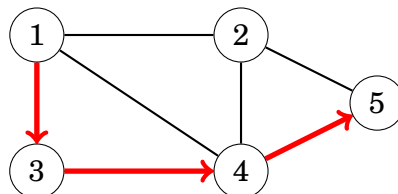
### 11.1 Terminología de grafos

Un **grafo** consiste en **nodos** y **aristas**. En este libro, la variable  $n$  denota el número de nodos en un grafo, y la variable  $m$  denota el número de aristas. Los nodos están numerados utilizando enteros  $1, 2, \dots, n$ .

Por ejemplo, el siguiente grafo consta de 5 nodos y 7 aristas:



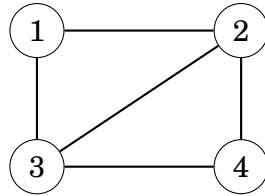
Un **camino** va del nodo  $a$  al nodo  $b$  a través de aristas del grafo. La **longitud** de un camino es el número de aristas en él. Por ejemplo, el grafo anterior contiene un camino  $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  de longitud 3 desde el nodo 1 hasta el nodo 5:



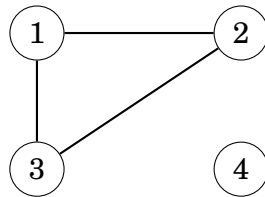
Un camino es un **ciclo** si el primer y último nodo es el mismo. Por ejemplo, el grafo anterior contiene un ciclo  $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ . Un camino es **simple** si cada nodo aparece como máximo una vez en el camino.

## Conectividad

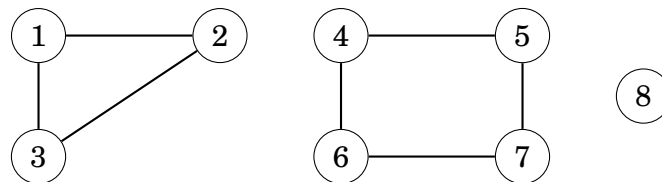
Un grafo es **conectado** si hay un camino entre dos nodos cualesquiera. Por ejemplo, el siguiente grafo está conectado:



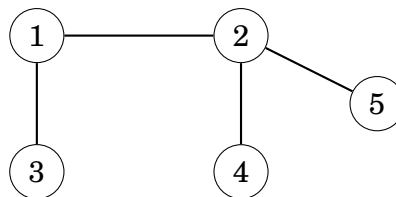
El siguiente grafo no está conectado, porque no es posible llegar desde el nodo 4 a ningún otro nodo:



Las partes conectadas de un grafo son llamadas sus **componentes**. Por ejemplo, el siguiente grafo contiene tres componentes:  $\{1, 2, 3\}$ ,  $\{4, 5, 6, 7\}$  y  $\{8\}$ .

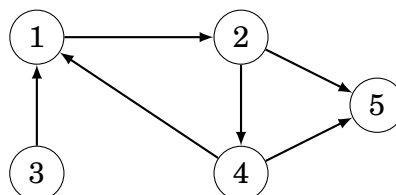


Un **árbol** es un gráfico conectado que consta de  $n$  nodos y  $n - 1$  aristas. Existe una ruta única entre dos nodos cualesquiera de un árbol. Por ejemplo, el siguiente gráfico es un árbol:



## Direcciones de las aristas

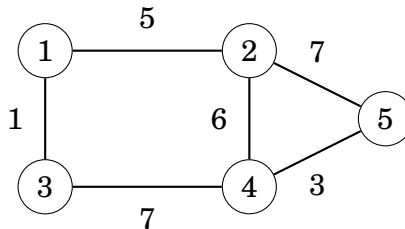
Un gráfico es **dirigido** si las aristas se pueden recorrer en una sola dirección. Por ejemplo, el siguiente gráfico está dirigido:



El gráfico anterior contiene una ruta  $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$  desde el nodo 3 hasta el nodo 5, pero no hay una ruta desde el nodo 5 hasta el nodo 3.

## Pesos de las aristas

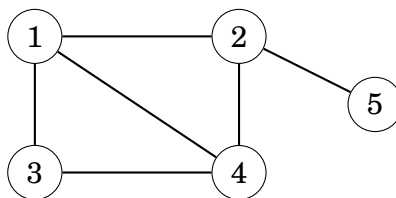
En un gráfico **ponderado**, cada arista se asigna un **peso**. Los pesos a menudo se interpretan como longitudes de las aristas. Por ejemplo, el siguiente gráfico está ponderado:



La longitud de una ruta en un gráfico ponderado es la suma de los pesos de las aristas en la ruta. Por ejemplo, en el gráfico anterior, la longitud de la ruta  $1 \rightarrow 2 \rightarrow 5$  es 12, y la longitud de la ruta  $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  es 11. La última ruta es la ruta **más corta** desde el nodo 1 hasta el nodo 5.

## Vecinos y grados

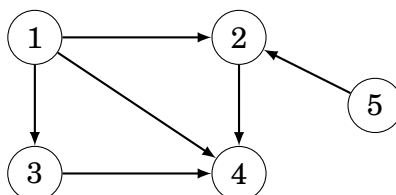
Dos nodos son **vecinos** o **adyacentes** si hay una arista entre ellos. El **grado** de un nodo es el número de sus vecinos. Por ejemplo, en el siguiente gráfico, los vecinos del nodo 2 son 1, 4 y 5, por lo que su grado es 3.



La suma de grados en un gráfico es siempre  $2m$ , donde  $m$  es el número de aristas, porque cada arista aumenta el grado de exactamente dos nodos en uno. Por esta razón, la suma de grados es siempre par.

Un gráfico es **regular** si el grado de cada nodo es una constante  $d$ . Un gráfico es **completo** si el grado de cada nodo es  $n - 1$ , es decir, el gráfico contiene todas las aristas posibles entre los nodos.

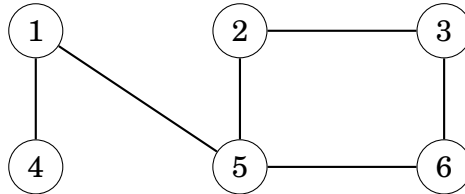
En un gráfico dirigido, el **grado de entrada** de un nodo es el número de aristas que terminan en el nodo, y el **grado de salida** de un nodo es el número de aristas que comienzan en el nodo. Por ejemplo, en el siguiente gráfico, el grado de entrada del nodo 2 es 2, y el grado de salida del nodo 2 es 1.



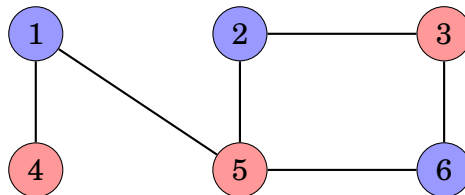
## Coloraciones

En una **coloración** de un grafo, cada nodo se asigna un color de modo que ningún nodo adyacente tenga el mismo color.

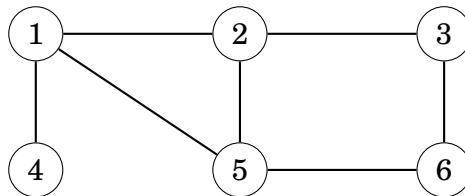
Un grafo es **bipartito** si es posible colorearlo utilizando dos colores. Resulta que un grafo es bipartito exactamente cuando no contiene un ciclo con un número impar de aristas. Por ejemplo, el grafo



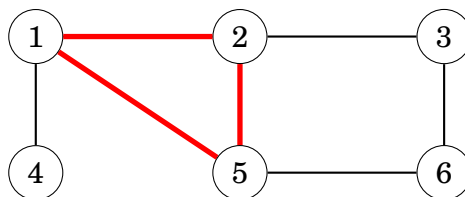
es bipartito, porque puede colorearse de la siguiente manera:



Sin embargo, el grafo

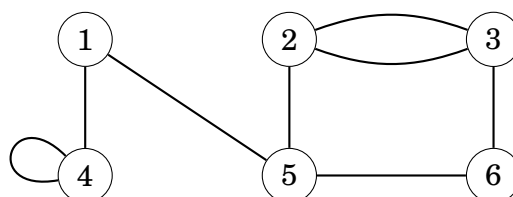


no es bipartito, porque no es posible colorear el siguiente ciclo de tres nodos usando dos colores:



## Simplicidad

Un grafo es **simple** si ninguna arista comienza y termina en el mismo nodo, y no hay múltiples aristas entre dos nodos. A menudo asumimos que los grafos son simples. Por ejemplo, el siguiente grafo *no* es simple:



## 11.2 Representación de grafos

Hay varias formas de representar grafos en algoritmos. La elección de una estructura de datos depende del tamaño del grafo y de la forma en que el algoritmo lo procesa. A continuación, veremos tres representaciones comunes.

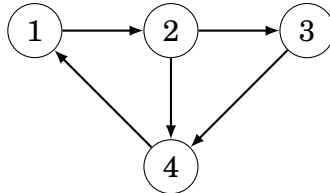
### Representación de lista de adyacencia

En la representación de lista de adyacencia, cada nodo  $x$  en el grafo se asigna una **lista de adyacencia** que consiste en los nodos a los que hay una arista desde  $x$ . Las listas de adyacencia son las más populares formas de representar grafos, y la mayoría de los algoritmos se pueden implementar de manera eficiente utilizando ellos.

Una forma conveniente de almacenar las listas de adyacencia es declarar una matriz de vectores de la siguiente manera:

```
vector<int> adj[N];
```

La constante  $N$  se elige de modo que todas las listas de adyacencia se puedan almacenar. Por ejemplo, el grafo



se puede almacenar de la siguiente manera:

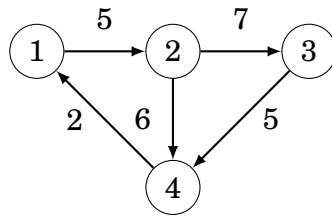
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Si el gráfico no está dirigido, se puede almacenar de manera similar, pero cada borde se agrega en ambas direcciones.

Para un gráfico ponderado, la estructura se puede ampliar de la siguiente manera:

```
vector<pair<int,int>> adj[N];
```

En este caso, la lista de adyacencia del nodo  $a$  contiene el par  $(b, w)$  siempre que haya un borde del nodo  $a$  al nodo  $b$  con peso  $w$ . Por ejemplo, el gráfico



se puede almacenar de la siguiente manera:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

La ventaja de usar listas de adyacencia es que podemos encontrar de manera eficiente los nodos a los que podemos movernos desde un nodo dado a través de un borde. Por ejemplo, el siguiente bucle recorre todos los nodos a los que podemos movernos desde el nodo  $s$ :

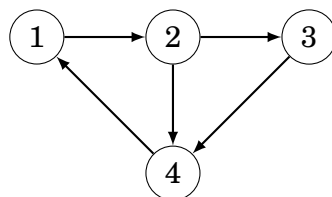
```
for (auto u : adj[s]) {
    // procesar nodo u
}
```

## Representación de matriz de adyacencia

Una **matriz de adyacencia** es una matriz bidimensional que indica qué bordes contiene el gráfico. Podemos verificar de manera eficiente desde una matriz de adyacencia si hay un borde entre dos nodos. La matriz se puede almacenar como una matriz

```
int adj[N][N];
```

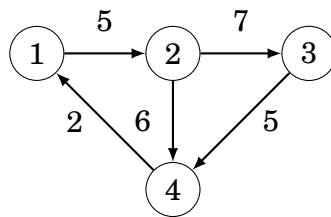
donde cada valor  $adj[a][b]$  indica si el gráfico contiene un borde de nodo  $a$  a nodo  $b$ . Si el borde está incluido en el gráfico, entonces  $adj[a][b] = 1$ , y de lo contrario  $adj[a][b] = 0$ . Por ejemplo, el gráfico



se puede representar de la siguiente manera:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Si el gráfico está ponderado, la representación de la matriz de adyacencia se puede ampliar para que la matriz contenga el peso del borde si el borde existe. Usando esta representación, el gráfico



corresponde a la siguiente matriz:

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

El inconveniente de la representación de la matriz de adyacencia es que la matriz contiene  $n^2$  elementos, y por lo general la mayoría de ellos son cero. Por esta razón, la representación no se puede utilizar si el gráfico es grande.

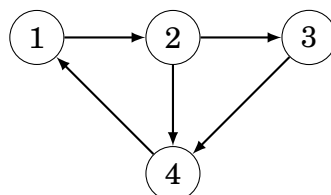
## Representación de la lista de aristas

Una **lista de aristas** contiene todas las aristas de un gráfico en algún orden. Esta es una forma conveniente de representar un gráfico si el algoritmo procesa todas las aristas del gráfico y no es necesario encontrar aristas que comiencen en un nodo dado.

La lista de aristas se puede almacenar en un vector

```
vector<pair<int,int>> edges;
```

donde cada par  $(a, b)$  denota que hay una arista del nodo  $a$  al nodo  $b$ . Por lo tanto, el gráfico



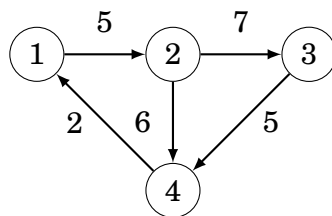
se puede representar de la siguiente manera:

```
edges.push_back({1,2});  
edges.push_back({2,3});  
edges.push_back({2,4});  
edges.push_back({3,4});  
edges.push_back({4,1});
```

Si el gráfico está ponderado, la estructura puede extenderse de la siguiente manera:

```
vector<tuple<int,int,int>> edges;
```

Cada elemento de esta lista es de la forma  $(a,b,w)$ , lo que significa que hay una arista del nodo  $a$  al nodo  $b$  con peso  $w$ . Por ejemplo, el gráfico



se puede representar de la siguiente manera<sup>1</sup>:

```
edges.push_back({1,2,5});  
edges.push_back({2,3,7});  
edges.push_back({2,4,6});  
edges.push_back({3,4,5});  
edges.push_back({4,1,2});
```

---

<sup>1</sup>En algunos compiladores antiguos, la función `make_tuple` debe utilizarse en lugar de las llaves (por ejemplo, `make_tuple(1,2,5)` en lugar de `{1,2,5}`).



# Chapter 12

## Recorrido de grafos

Este capítulo discute dos algoritmos de grafos fundamentales: búsqueda en profundidad y búsqueda en amplitud. Ambos algoritmos se dan un nodo de inicio en el gráfico, y visitan todos los nodos que se pueden alcanzar desde el nodo de inicio. La diferencia en los algoritmos es el orden en el que visitan los nodos.

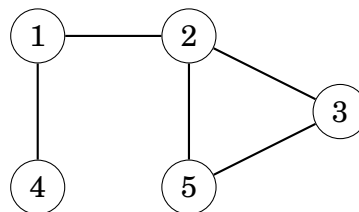
### 12.1 Búsqueda en profundidad

**Búsqueda en profundidad** (DFS) es una técnica de recorrido de grafos sencilla. El algoritmo comienza en un nodo de inicio, y procede a todos los demás nodos que son alcanzables desde el nodo de inicio usando los bordes del grafo.

La búsqueda en profundidad siempre sigue un solo camino en el grafo mientras encuentre nuevos nodos. Después de esto, regresa a los nodos anteriores y comienza a explorar otras partes del grafo. El algoritmo lleva un registro de los nodos visitados, para que procese cada nodo solo una vez.

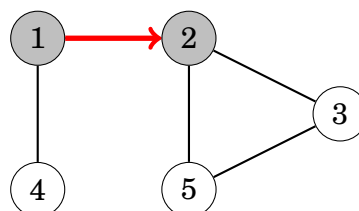
#### Ejemplo

Consideremos cómo la búsqueda en profundidad procesa el siguiente gráfico:

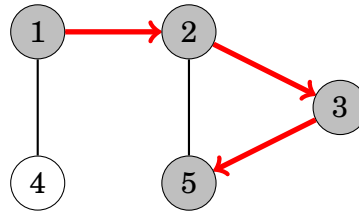


Podemos comenzar la búsqueda en cualquier nodo del gráfico; ahora comenzaremos la búsqueda en el nodo 1.

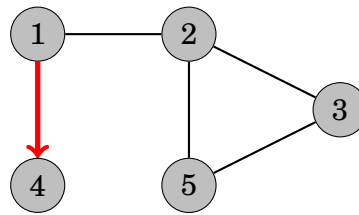
La búsqueda primero procede al nodo 2:



Después de esto, se visitarán los nodos 3 y 5:



Los vecinos del nodo 5 son 2 y 3, pero la búsqueda ya ha visitado ambos, por lo que es hora de volver a los nodos anteriores. También los vecinos de los nodos 3 y 2 han sido visitados, por lo que luego nos movemos desde el nodo 1 al nodo 4:



Después de esto, la búsqueda termina porque ha visitado todos los nodos.

La complejidad temporal de la búsqueda en profundidad es  $O(n + m)$  donde  $n$  es el número de nodos y  $m$  es el número de bordes, porque el algoritmo procesa cada nodo y borde una vez.

## Implementación

La búsqueda en profundidad se puede implementar convenientemente utilizando recursión. La siguiente función `dfs` comienza una búsqueda en profundidad en un nodo dado. La función asume que el gráfico está almacenado como listas de adyacencia en una matriz

```
vector<int> adj[N];
```

y también mantiene una matriz

```
bool visited[N];
```

que lleva un registro de los nodos visitados. Inicialmente, cada valor de la matriz es `false`, y cuando la búsqueda llega al nodo `s`, el valor de `visited[s]` se convierte en `true`. La función se puede implementar de la siguiente manera:

```
void dfs(int s) {  
    if (visited[s]) return;  
    visited[s] = true;  
    // procesar nodo s  
    for (auto u: adj[s]) {  
        dfs(u);  
    }  
}
```

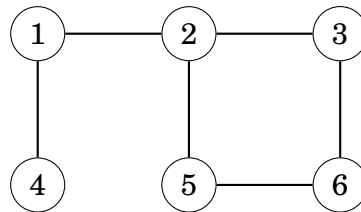
## 12.2 Búsqueda en amplitud

**Búsqueda en amplitud** (BFS) visita los nodos en orden creciente de su distancia desde el nodo de inicio. Por lo tanto, podemos calcular la distancia desde el nodo de inicio a todos los demás nodos utilizando la búsqueda en amplitud. Sin embargo, la búsqueda en amplitud es más difícil de implementar que la búsqueda en profundidad.

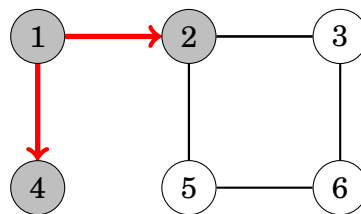
La búsqueda en amplitud recorre los nodos un nivel después de otro. Primero, la búsqueda explora los nodos cuyo distancia del nodo de inicio es 1, luego los nodos cuya distancia es 2, y así sucesivamente. Este proceso continúa hasta que todos los nodos han sido visitados.

### Ejemplo

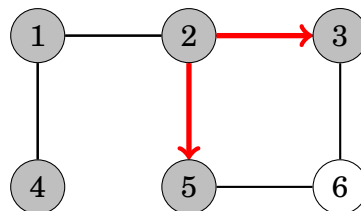
Consideremos cómo la búsqueda en amplitud procesa el siguiente gráfico:



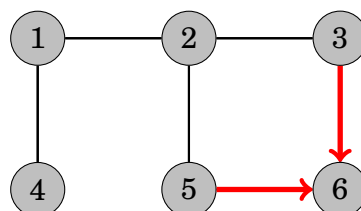
Supongamos que la búsqueda comienza en el nodo 1. Primero, procesamos todos los nodos que se pueden alcanzar desde el nodo 1 usando un solo borde:



Después de esto, procedemos a los nodos 3 y 5:



Finalmente, visitamos el nodo 6:



Ahora hemos calculado las distancias desde el nodo de inicio a todos los nodos del gráfico. Las distancias son las siguientes:

nodo	distancia
1	0
2	1
3	2
4	1
5	2
6	3

Al igual que en la búsqueda en profundidad, la complejidad temporal de la búsqueda en amplitud es  $O(n + m)$ , donde  $n$  es el número de nodos y  $m$  es el número de aristas.

## Implementación

La búsqueda en amplitud es más difícil de implementar que la búsqueda en profundidad, porque el algoritmo visita nodos en diferentes partes del gráfico. Una implementación típica se basa en una cola que contiene nodos. En cada paso, el siguiente nodo en la cola será procesado.

El siguiente código asume que el gráfico se almacena como listas de adyacencia y mantiene las siguientes estructuras de datos:

```
queue<int> q;  
bool visited[N];  
int distance[N];
```

La cola  $q$  contiene nodos a procesar en orden creciente de su distancia. Los nuevos nodos siempre se agregan al final de la cola, y el nodo al principio de la cola es el siguiente nodo a procesar. La matriz `visited` indica qué nodos la búsqueda ya ha visitado, y la matriz `distance` contendrá la distancias desde el nodo de inicio a todos los nodos del gráfico. La búsqueda se puede implementar de la siguiente manera, comenzando en el nodo  $x$ :

```
visited[x] = true;  
distance[x] = 0;  
q.push(x);  
while (!q.empty()) {  
    int s = q.front(); q.pop();  
    // process node s  
    for (auto u : adj[s]) {  
        if (visited[u]) continue;  
        visited[u] = true;  
        distance[u] = distance[s]+1;  
        q.push(u);  
    }  
}
```

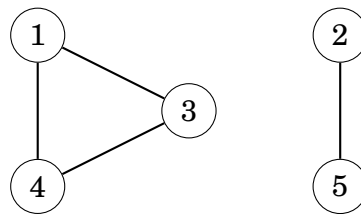
## 12.3 Aplicaciones

Usando los algoritmos de recorrido de grafos, podemos verificar muchas propiedades de los grafos. Por lo general, tanto la búsqueda en profundidad como la búsqueda en anchura se pueden utilizar, pero en la práctica, la búsqueda en profundidad es una mejor opción, porque es más fácil de implementar. En las siguientes aplicaciones asumiremos que el grafo no está dirigido.

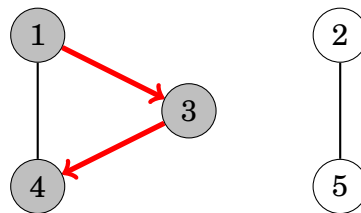
### Verificación de conectividad

Un grafo está conectado si hay un camino entre dos nodos cualesquiera del grafo. Por lo tanto, podemos verificar si un grafo está conectado comenzando en un nodo arbitrario y averiguando si podemos llegar a todos los demás nodos.

Por ejemplo, en el grafo



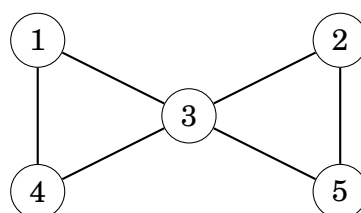
una búsqueda en profundidad desde el nodo 1 visita los siguientes nodos:



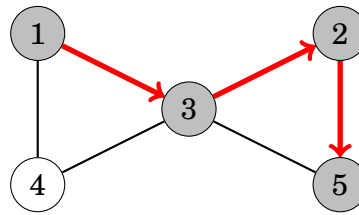
Como la búsqueda no visitó todos los nodos, podemos concluir que el grafo no está conectado. De manera similar, también podemos encontrar todos los componentes conectados de un grafo iterando a través de los nodos y siempre comenzando una nueva búsqueda en profundidad si el nodo actual no pertenece a ningún componente todavía.

### Encontrar ciclos

Un grafo contiene un ciclo si durante un recorrido del grafo, encontramos un nodo cuyo vecino (diferente del nodo anterior en la ruta actual) ya ha sido visitado. Por ejemplo, el grafo



contiene dos ciclos y podemos encontrar uno de ellos de la siguiente manera:



Después de movernos del nodo 2 al nodo 5, notamos que el vecino 3 del nodo 5 ya ha sido visitado. Por lo tanto, el grafo contiene un ciclo que pasa por el nodo 3, por ejemplo,  $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$ .

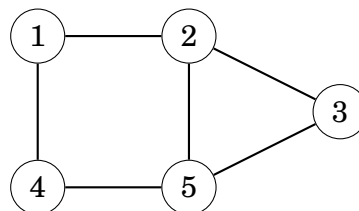
Otra forma de averiguar si un grafo contiene un ciclo es simplemente calcular el número de nodos y aristas en cada componente. Si un componente contiene  $c$  nodos y no ciclo, debe contener exactamente  $c - 1$  aristas (por lo que tiene que ser un árbol). Si hay  $c$  o más aristas, el componente seguramente contiene un ciclo.

## Verificación de bipartición

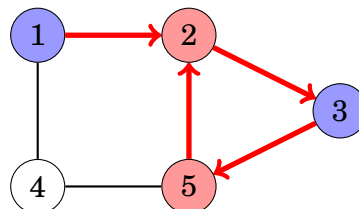
Un grafo es bipartito si sus nodos se pueden colorear usando dos colores de modo que no haya nodos adyacentes con el mismo color. Es sorprendentemente fácil comprobar si un grafo es bipartito utilizando algoritmos de recorrido del grafo.

La idea es colorear el nodo inicial de azul, todos sus vecinos de rojo, todos sus vecinos de azul, y así sucesivamente. Si en algún momento de la búsqueda notamos que dos nodos adyacentes tienen el mismo color, esto significa que el grafo no es bipartito. De lo contrario, el grafo es bipartito y se ha encontrado una coloración.

Por ejemplo, el gráfico



no es bipartito, porque una búsqueda desde el nodo 1 procede de la siguiente manera:



Observamos que el color de ambos nodos 2 y 5 es rojo, mientras que son nodos adyacentes en el gráfico. Por lo tanto, el gráfico no es bipartito.

Este algoritmo siempre funciona, porque cuando hay solo dos colores disponibles, el color del nodo de inicio en un componente determina los colores de todos los demás nodos en el componente. No importa si el nodo de inicio es rojo o azul.

Tenga en cuenta que en el caso general, es difícil averiguar si los nodos en un gráfico se pueden colorear usando  $k$  colores de modo que ningún nodo adyacente tenga el mismo color. Incluso cuando  $k = 3$ , no se conoce ningún algoritmo eficiente pero el problema es NP-difícil.





# Chapter 13

## Camino más cortos

Encontrar un camino más corto entre dos nodos de un grafo es un problema importante que tiene muchas aplicaciones prácticas. Por ejemplo, un problema natural relacionado con una red de carreteras es calcular la longitud más corta posible de una ruta entre dos ciudades, dadas las longitudes de las carreteras.

En un grafo no ponderado, la longitud de un camino es igual al número de sus aristas, y podemos simplemente usar la búsqueda en amplitud para encontrar un camino más corto. Sin embargo, en este capítulo nos centramos en grafos ponderados donde se necesitan algoritmos más sofisticados para encontrar caminos más cortos.

### 13.1 Algoritmo de Bellman–Ford

El **algoritmo de Bellman–Ford**<sup>1</sup> encuentra caminos más cortos desde un nodo de inicio hasta todos los nodos del grafo. El algoritmo puede procesar todo tipo de grafos, siempre que el grafo no contenga un ciclo con longitud negativa. Si el grafo contiene un ciclo negativo, el algoritmo puede detectarlo.

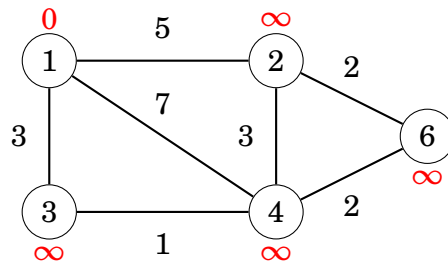
El algoritmo realiza un seguimiento de las distancias desde el nodo de inicio hasta todos los nodos del grafo. Inicialmente, la distancia al nodo de inicio es 0 y la distancia a todos los demás nodos es infinita. El algoritmo reduce las distancias encontrando aristas que acortan los caminos hasta que ya no es posible reducir ninguna distancia.

### Ejemplo

Consideremos cómo funciona el algoritmo de Bellman–Ford en el siguiente grafo:

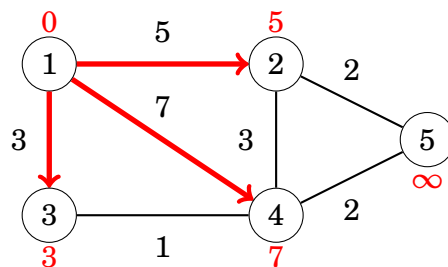
---

<sup>1</sup>El algoritmo lleva el nombre de R. E. Bellman y L. R. Ford quienes lo publicaron independientemente en 1958 y 1956, respectivamente [5, 24].

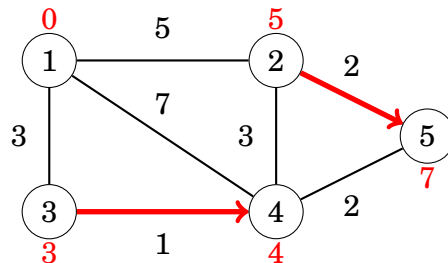


A cada nodo del grafo se le asigna una distancia. Inicialmente, la distancia al nodo de inicio es 0, y la distancia a todos los demás nodos es infinita.

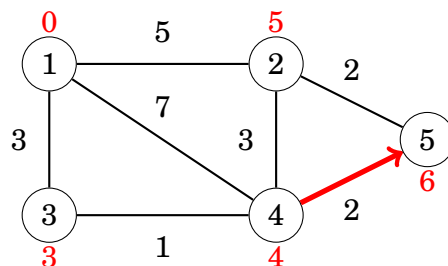
El algoritmo busca aristas que reduzcan distancias. Primero, todas las aristas desde el nodo 1 reducen distancias:



Después de esto, las aristas 2 → 5 y 3 → 4 reducen distancias:

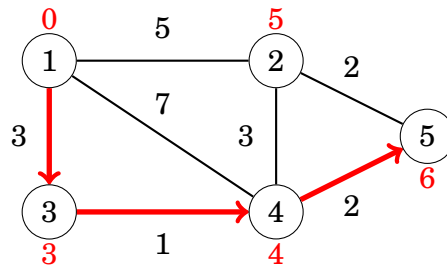


Finalmente, hay un cambio más:



Después de esto, ninguna arista puede reducir ninguna distancia. Esto significa que las distancias son finales, y hemos calculado exitosamente las distancias más cortas desde el nodo inicial a todos los nodos del grafo.

Por ejemplo, la distancia más corta 3 desde el nodo 1 al nodo 5 corresponde a la siguiente ruta:



## Implementación

La siguiente implementación del algoritmo de Bellman–Ford determina las distancias más cortas desde un nodo  $x$  a todos los nodos del grafo. El código asume que el grafo está almacenado como una lista de aristas `edges` que consiste en tuplas de la forma  $(a, b, w)$ , lo que significa que hay una arista desde el nodo  $a$  al nodo  $b$  con peso  $w$ .

El algoritmo consta de  $n - 1$  rondas, y en cada ronda el algoritmo recorre todas las aristas del grafo e intenta reducir las distancias. El algoritmo construye una matriz `distance` que contendrá las distancias desde  $x$  a todos los nodos del grafo. La constante `INF` denota una distancia infinita.

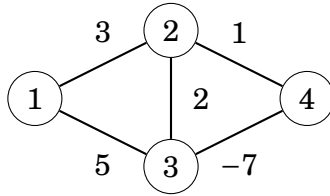
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

La complejidad temporal del algoritmo es  $O(nm)$ , porque el algoritmo consta de  $n - 1$  rondas y itera a través de todas las  $m$  aristas durante una ronda. Si no hay ciclos negativos en el grafo, todas las distancias son finales después de  $n - 1$  rondas, porque cada ruta más corta puede contener como máximo  $n - 1$  aristas.

En la práctica, las distancias finales generalmente se pueden encontrar más rápido que en  $n - 1$  rondas. Por lo tanto, una forma posible de hacer que el algoritmo sea más eficiente es detener el algoritmo si ninguna distancia se puede reducir durante una ronda.

## Ciclos negativos

El algoritmo de Bellman–Ford también se puede usar para comprobar si el grafo contiene un ciclo con longitud negativa. Por ejemplo, el grafo



contiene un ciclo negativo  $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$  con longitud  $-4$ .

Si el grafo contiene un ciclo negativo, podemos acortar infinitamente muchas veces cualquier camino que contenga el ciclo repitiendo el ciclo una y otra vez. Por lo tanto, el concepto de camino más corto no tiene sentido en esta situación.

Un ciclo negativo se puede detectar utilizando el algoritmo de Bellman–Ford mediante la ejecución del algoritmo durante  $n$  rondas. Si la última ronda reduce cualquier distancia, el grafo contiene un ciclo negativo. Tenga en cuenta que este algoritmo se puede usar para buscar un ciclo negativo en todo el grafo independientemente del nodo de inicio.

## Algoritmo SPFA

El **algoritmo SPFA** ("Shortest Path Faster Algorithm") [20] es una variante del algoritmo de Bellman–Ford, que a menudo es más eficiente que el algoritmo original. El algoritmo SPFA no recorre todos los bordes en cada ronda, sino que, en cambio, elige los bordes que se examinarán de una manera más inteligente.

El algoritmo mantiene una cola de nodos que podrían usarse para reducir las distancias. Primero, el algoritmo agrega el nodo de inicio  $x$  a la cola. Luego, el algoritmo siempre procesa el primer nodo en la cola, y cuando un borde  $a \rightarrow b$  reduce una distancia, el nodo  $b$  se agrega a la cola.

La eficiencia del algoritmo SPFA depende de la estructura del grafo: el algoritmo suele ser eficiente, pero su complejidad temporal en el peor de los casos sigue siendo  $O(nm)$  y es posible crear entradas que hagan que el algoritmo sea tan lento como el algoritmo original de Bellman–Ford.

## 13.2 Algoritmo de Dijkstra

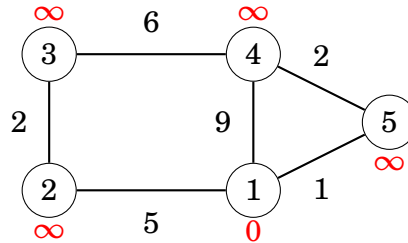
El **algoritmo de Dijkstra**<sup>2</sup> encuentra los caminos más cortos desde el nodo de inicio hasta todos los nodos del grafo, como el algoritmo de Bellman–Ford. La ventaja del algoritmo de Dijkstra es que es más eficiente y se puede usar para procesar grafos grandes. Sin embargo, el algoritmo requiere que haya bordes sin peso negativo en el grafo.

Al igual que el algoritmo de Bellman–Ford, el algoritmo de Dijkstra mantiene las distancias a los nodos y las reduce durante la búsqueda. El algoritmo de Dijkstra es eficiente porque solo procesa cada borde en el grafo una vez, utilizando el hecho de que no hay bordes negativos.

<sup>2</sup>E. W. Dijkstra publicó el algoritmo en 1959 [14]; sin embargo, su documento original no menciona cómo implementar el algoritmo de manera eficiente.

## Ejemplo

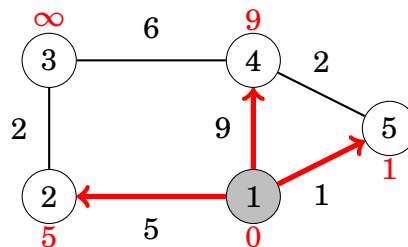
Consideremos cómo funciona el algoritmo de Dijkstra en el siguiente grafo cuando el nodo de inicio es el nodo 1:



Al igual que en el algoritmo de Bellman–Ford, inicialmente la distancia al nodo de inicio es 0 y la distancia a todos los demás nodos es infinita.

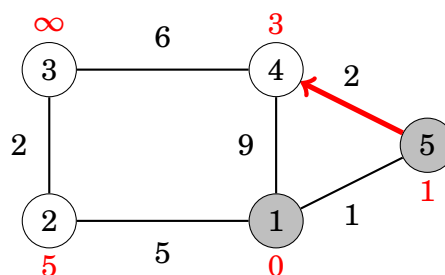
En cada paso, el algoritmo de Dijkstra selecciona un nodo que aún no se ha procesado y cuya distancia es lo más pequeña posible. El primer nodo de este tipo es el nodo 1 con distancia 0.

Cuando se selecciona un nodo, el algoritmo recorre todos los bordes que comienzan en el nodo y reduce las distancias usándolos:

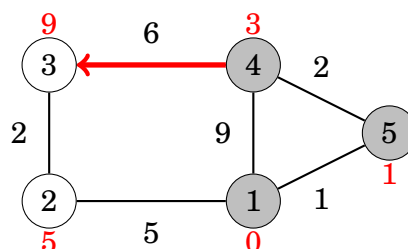


En este caso, los bordes del nodo 1 redujeron las distancias de los nodos 2, 4 y 5, cuyas distancias ahora son 5, 9 y 1.

El siguiente nodo que se procesará es el nodo 5 con distancia 1. Esto reduce la distancia al nodo 4 de 9 a 3:

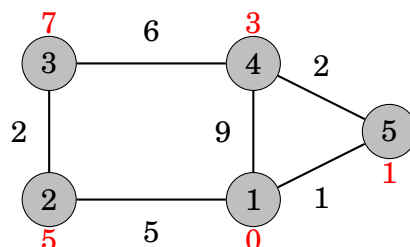


Después de esto, el siguiente nodo es el nodo 4, que reduce la distancia al nodo 3 a 9:



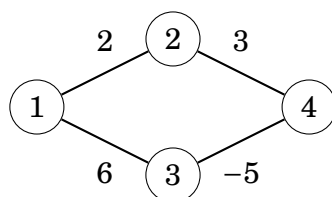
Una propiedad notable en el algoritmo de Dijkstra es que cada vez que se selecciona un nodo, su distancia es final. Por ejemplo, en este punto del algoritmo, las distancias 0, 1 y 3 son las distancias finales a los nodos 1, 5 y 4.

Después de esto, el algoritmo procesa los dos nodos restantes, y las distancias finales son las siguientes:



## Aristas negativas

La eficiencia del algoritmo de Dijkstra es basada en el hecho de que el gráfico no contiene aristas negativas. Si hay una arista negativa, el algoritmo puede dar resultados incorrectos. Como ejemplo, considere el siguiente gráfico:



El camino más corto desde el nodo 1 hasta el nodo 4 es  $1 \rightarrow 3 \rightarrow 4$  y su longitud es 1. Sin embargo, el algoritmo de Dijkstra encuentra el camino  $1 \rightarrow 2 \rightarrow 4$  siguiendo las aristas de peso mínimo. El algoritmo no tiene en cuenta que en el otro camino, el peso  $-5$  compensa el peso anterior grande 6.

## Implementación

La siguiente implementación del algoritmo de Dijkstra calcula las distancias mínimas desde un nodo  $x$  a otros nodos del gráfico. El gráfico se almacena como listas de adyacencia de modo que  $\text{adj}[a]$  contiene un par  $(b, w)$  siempre que haya una arista desde el nodo  $a$  hasta el nodo  $b$  con peso  $w$ .

Una implementación eficiente del algoritmo de Dijkstra requiere que sea posible encontrar eficientemente el nodo de distancia mínima que no se ha procesado. Una estructura de datos apropiada para esto es una cola de prioridad que contiene los nodos ordenados por sus distancias. Usando una cola de prioridad, el siguiente nodo a procesar se puede recuperar en tiempo logarítmico.

En el siguiente código, la cola de prioridad  $q$  contiene pares de la forma  $(-d, x)$ , lo que significa que la distancia actual al nodo  $x$  es  $d$ . El array `distance` contiene la distancia a cada nodo, y el array `processed` indica si un nodo se ha procesado. Inicialmente la distancia es 0 a  $x$  e  $\infty$  a todos los demás nodos.

```

for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
}

```

Tenga en cuenta que la cola de prioridad contiene *negativas* distancias a los nodos. La razón de esto es que la versión predeterminada de la cola de prioridad de C++ encuentra el máximo elementos, mientras que queremos encontrar elementos mínimos. Al usar distancias negativas, podemos usar directamente la cola de prioridad predeterminada<sup>3</sup>. También tenga en cuenta que puede haber varias instancias del mismo nodo en la cola de prioridad; sin embargo, solo la instancia con la distancia mínima será procesada.

La complejidad temporal de la implementación anterior es  $O(n + m \log m)$ , porque el algoritmo pasa por todos los nodos del gráfico y agrega para cada arista como máximo una distancia a la cola de prioridad.

### 13.3 Algoritmo de Floyd–Warshall

El **algoritmo de Floyd–Warshall**<sup>4</sup> proporciona una forma alternativa de abordar el problema de encontrar las rutas más cortas. A diferencia de los otros algoritmos de este capítulo, encuentra todas las rutas más cortas entre los nodos en una sola pasada.

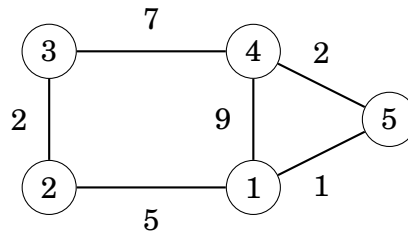
El algoritmo mantiene una matriz bidimensional que contiene las distancias entre los nodos. Primero, las distancias se calculan solo usando aristas directas entre los nodos, y después de esto, el algoritmo reduce las distancias usando nodos intermedios en las rutas.

#### Ejemplo

Consideremos cómo funciona el algoritmo de Floyd–Warshall en el siguiente gráfico:

<sup>3</sup>Por supuesto, también podríamos declarar la cola de prioridad como en el Capítulo 4.5 y usar distancias positivas, pero la implementación sería un poco más larga.

<sup>4</sup>El algoritmo lleva el nombre de R. W. Floyd y S. Warshall que lo publicaron de forma independiente en 1962 [23, 70].



Inicialmente, la distancia desde cada nodo a sí mismo es 0, y la distancia entre los nodos  $a$  y  $b$  es  $x$  si hay una arista entre los nodos  $a$  y  $b$  con peso  $x$ . Todas las demás distancias son infinitas. En este gráfico, la matriz inicial es la siguiente:

	1	2	3	4	5
1	0	5	$\infty$	9	1
2	5	0	2	$\infty$	$\infty$
3	$\infty$	2	0	7	$\infty$
4	9	$\infty$	7	0	2
5	1	$\infty$	$\infty$	2	0

El algoritmo consiste en rondas consecutivas. En cada ronda, el algoritmo selecciona un nuevo nodo que puede actuar como un nodo intermedio en las rutas a partir de ahora, y las distancias se reducen usando este nodo.

En la primera ronda, el nodo 1 es el nuevo nodo intermedio. Hay una nueva ruta entre los nodos 2 y 4 con longitud 14, porque el nodo 1 los conecta. También hay una nueva ruta entre los nodos 2 y 5 con longitud 6.

	1	2	3	4	5
1	0	5	$\infty$	9	1
2	5	0	2	<b>14</b>	<b>6</b>
3	$\infty$	2	0	7	$\infty$
4	9	<b>14</b>	7	0	2
5	1	<b>6</b>	$\infty$	2	0

En la segunda ronda, el nodo 2 es el nuevo nodo intermedio. Esto crea nuevas rutas entre los nodos 1 y 3 y entre los nodos 3 y 5:

	1	2	3	4	5
1	0	5	<b>7</b>	9	1
2	5	0	2	14	6
3	<b>7</b>	2	0	7	<b>8</b>
4	9	14	7	0	2
5	1	6	<b>8</b>	2	0

En la tercera ronda, el nodo 3 es el nuevo nodo intermedio. Hay una nueva ruta entre los nodos 2 y 4:

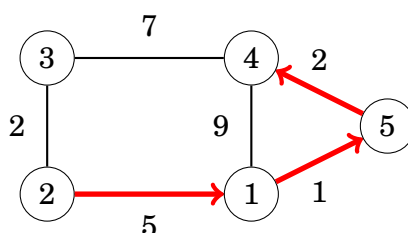


	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

El algoritmo continúa así, hasta que todos los nodos han sido nombrados nodos intermedios. Después de que el algoritmo ha terminado, la matriz contiene las distancias mínimas entre dos nodos cualesquiera:

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

Por ejemplo, la matriz nos dice que la distancia más corta entre los nodos 2 y 4 es 8. Esto corresponde a la siguiente ruta:



## Implementación

La ventaja del algoritmo de Floyd–Warshall es que es fácil de implementar. El siguiente código construye una matriz de distancia donde  $\text{distance}[a][b]$  es la distancia más corta entre los nodos  $a$  y  $b$ . Primero, el algoritmo inicializa  $\text{distance}$  usando la matriz de adyacencia  $\text{adj}$  del gráfico:

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}

```

Después de esto, las distancias más cortas se pueden encontrar de la siguiente manera:

```
for (int k = 1; k <= n; k++) {  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j <= n; j++) {  
            distance[i][j] = min(distance[i][j],  
                                distance[i][k]+distance[k][j]);  
        }  
    }  
}
```

La complejidad temporal del algoritmo es  $O(n^3)$ , porque contiene tres bucles anidados que recorren los nodos del gráfico.

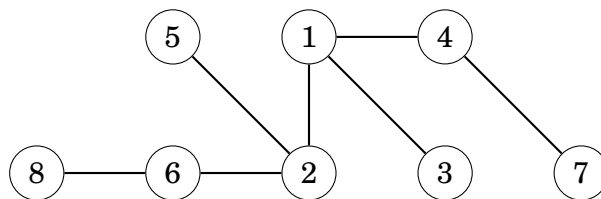
Dado que la implementación del algoritmo de Floyd–Warshall es simple, el algoritmo puede ser una buena opción incluso si solo se necesita para encontrar un solo camino más corto en el gráfico. Sin embargo, el algoritmo solo se puede usar cuando el gráfico es tan pequeño que una complejidad temporal cúbica es lo suficientemente rápida.

# Chapter 14

## Algoritmos de árbol

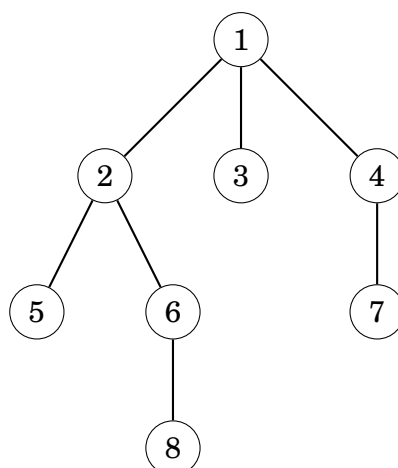
Un **árbol** es un grafo conectado y acíclico que consiste en  $n$  nodos y  $n - 1$  aristas. Eliminar cualquier arista de un árbol lo divide en dos componentes, y agregar cualquier arista a un árbol crea un ciclo. Además, siempre hay un camino único entre dos nodos de un árbol.

Por ejemplo, el siguiente árbol consta de 8 nodos y 7 aristas:



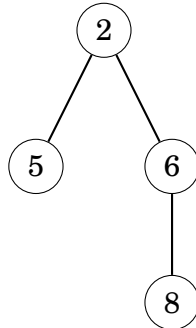
Las **hojas** de un árbol son los nodos con grado 1, es decir, con solo un vecino. Por ejemplo, las hojas del árbol anterior son los nodos 3, 5, 7 y 8.

En un árbol **enraizado**, uno de los nodos se designa como la **raíz** del árbol, y todos los demás nodos se colocan debajo de la raíz. Por ejemplo, en el siguiente árbol, el nodo 1 es el nodo raíz.



En un árbol enraizado, los **hijos** de un nodo son sus vecinos inferiores, y el **padre** de un nodo es su vecino superior. Cada nodo tiene exactamente un padre, excepto por la raíz que no tiene padre. Por ejemplo, en el árbol anterior, los hijos del nodo 2 son los nodos 5 y 6, y su padre es el nodo 1.

La estructura de un árbol enraizado es *recursiva*: cada nodo del árbol actúa como la raíz de un **subárbol** que contiene el nodo mismo y todos los nodos que están en los subárboles de sus hijos. Por ejemplo, en el árbol anterior, el subárbol del nodo 2 contiene los nodos 2, 5, 6 y 8:



## 14.1 Recorrido de árbol

Los algoritmos de recorrido de grafo general se pueden utilizar para recorrer los nodos de un árbol. Sin embargo, el recorrido de un árbol es más fácil de implementar que el de un grafo general, porque no hay ciclos en el árbol y no es posible alcanzar un nodo desde múltiples direcciones.

La forma típica de recorrer un árbol es comenzar una búsqueda en profundidad en un nodo arbitrario. La siguiente función recursiva se puede usar:

```
void dfs(int s, int e) {  
    // procesar nodo s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

La función recibe dos parámetros: el nodo actual  $s$  y el nodo anterior  $e$ . El propósito del parámetro  $e$  es asegurar que la búsqueda solo se mueva a nodos que no se han visitado todavía.

La siguiente llamada de función inicia la búsqueda en el nodo  $x$ :

```
dfs(x, 0);
```

En la primera llamada  $e = 0$ , porque no hay nodo anterior, y se permite proceder a cualquier dirección en el árbol.

## Programación dinámica

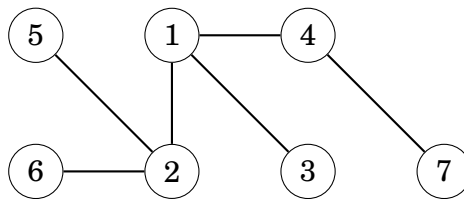
La programación dinámica se puede utilizar para calcular alguna información durante un recorrido de árbol. Usando programación dinámica, podemos, por ejemplo, calcular en tiempo  $O(n)$  para cada nodo de un árbol enraizado la cantidad de nodos en su subárbol o la longitud del camino más largo desde el nodo a una hoja.

Como ejemplo, calculemos para cada nodo  $s$  un valor  $\text{count}[s]$ : la cantidad de nodos en su subárbol. El subárbol contiene el nodo mismo y todos los nodos en los subárboles de sus hijos, por lo que podemos calcular la cantidad de nodos recursivamente usando el siguiente código:

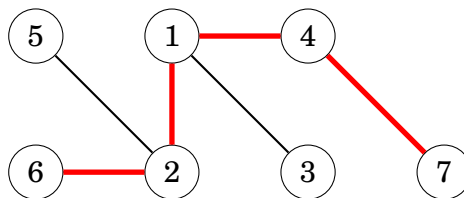
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

## 14.2 Diámetro

El **diámetro** de un árbol es la longitud máxima de un camino entre dos nodos. Por ejemplo, considera el siguiente árbol:



El diámetro de este árbol es 4, lo que corresponde al siguiente camino:



Tenga en cuenta que puede haber varios caminos de longitud máxima. En el camino anterior, podríamos reemplazar el nodo 6 con el nodo 5 para obtener otro camino con longitud 4.

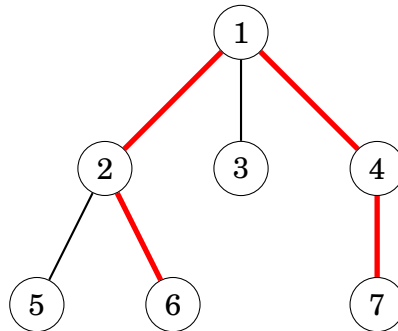
A continuación, discutiremos dos algoritmos de tiempo  $O(n)$  para calcular el diámetro de un árbol. El primer algoritmo se basa en la programación dinámica, y el segundo algoritmo utiliza dos búsquedas en profundidad.

### Algoritmo 1

Una forma general de abordar muchos problemas de árboles es primero enraizar el árbol arbitrariamente. Después de esto, podemos intentar resolver el problema por separado para cada subárbol. Nuestro primer algoritmo para calcular el diámetro se basa en esta idea.

Una observación importante es que cada camino en un árbol enraizado tiene un *punto más alto*: el nodo más alto que pertenece al camino. Por lo tanto, podemos calcular para cada nodo la longitud del camino más largo cuyo punto más alto es el nodo. Uno de esos caminos corresponde al diámetro del árbol.

Por ejemplo, en el siguiente árbol, el nodo 1 es el punto más alto en el camino que corresponde al diámetro:



Calculamos para cada nodo  $x$  dos valores:

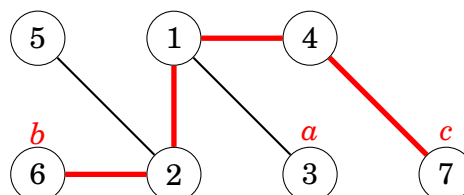
- $\text{toLeaf}(x)$ : la longitud máxima de un camino desde  $x$  hasta cualquier hoja
- $\text{maxLength}(x)$ : la longitud máxima de un camino cuyo punto más alto es  $x$

Por ejemplo, en el árbol anterior,  $\text{toLeaf}(1) = 2$ , porque hay un camino  $1 \rightarrow 2 \rightarrow 6$ , y  $\text{maxLength}(1) = 4$ , porque hay un camino  $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ . En este caso,  $\text{maxLength}(1)$  es igual al diámetro.

La programación dinámica se puede utilizar para calcular los anteriores valores para todos los nodos en  $O(n)$  tiempo. Primero, para calcular  $\text{toLeaf}(x)$ , recorremos los hijos de  $x$ , elegimos un hijo  $c$  con máximo  $\text{toLeaf}(c)$  y sumamos uno a este valor. Luego, para calcular  $\text{maxLength}(x)$ , elegimos dos hijos distintos  $a$  y  $b$  de modo que la suma  $\text{toLeaf}(a) + \text{toLeaf}(b)$  es máxima y sumamos dos a esta suma.

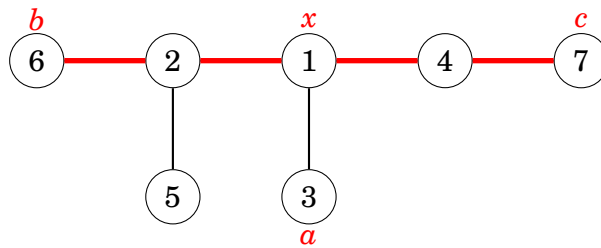
## Algoritmo 2

Otra forma eficiente de calcular el diámetro de un árbol se basa en dos búsquedas en profundidad. Primero, elegimos un nodo arbitrario  $a$  en el árbol y encontramos el nodo más lejano  $b$  de  $a$ . Luego, encontramos el nodo más lejano  $c$  de  $b$ . El diámetro del árbol es la distancia entre  $b$  y  $c$ . En la siguiente gráfica,  $a$ ,  $b$  y  $c$  podrían ser:



Este es un método elegante, pero ¿por qué funciona?

Es útil dibujar el árbol de manera diferente para que el camino que corresponde al diámetro sea horizontal, y todos los demás nodos cuelguen de él:

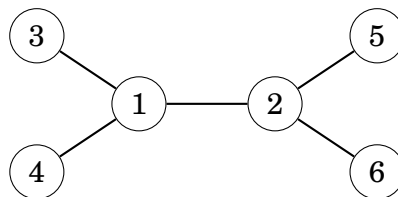


El nodo  $x$  indica el lugar donde el camino desde el nodo  $a$  se une al camino que corresponde al diámetro. El nodo más lejano de  $a$  es el nodo  $b$ , el nodo  $c$  o algún otro nodo que está al menos tan lejos del nodo  $x$ . Por lo tanto, este nodo siempre es una opción válida para un punto final de un camino que corresponde al diámetro.

### 14.3 Todos los caminos más largos

Nuestro próximo problema es calcular para cada nodo en el árbol la longitud máxima de un camino que comienza en el nodo. Esto se puede ver como una generalización de la problema de diámetro del árbol, porque el mayor de esos longitudes es igual al diámetro del árbol. También este problema se puede resolver en  $O(n)$  tiempo.

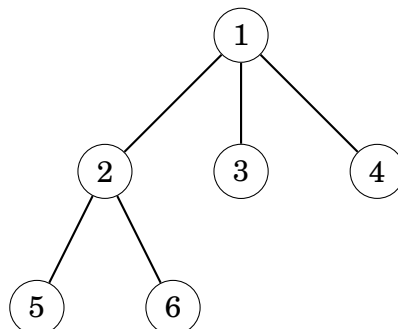
Como ejemplo, considere el siguiente árbol:



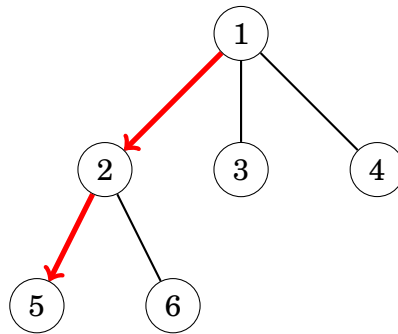
Sea  $\text{maxLength}(x)$  denotar la longitud máxima de un camino que comienza en el nodo  $x$ . Por ejemplo, en el árbol de arriba,  $\text{maxLength}(4) = 3$ , porque hay un camino  $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ . Aquí hay una tabla completa de los valores:

nodo $x$	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

También en este problema, un buen punto de partida para resolver el problema es enraizar el árbol arbitrariamente:

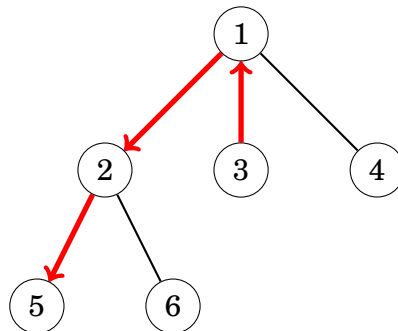


La primera parte del problema es calcular para cada nodo  $x$  la longitud máxima de un camino que pasa por un hijo de  $x$ . Por ejemplo, el camino más largo desde el nodo 1 pasa por su hijo 2:

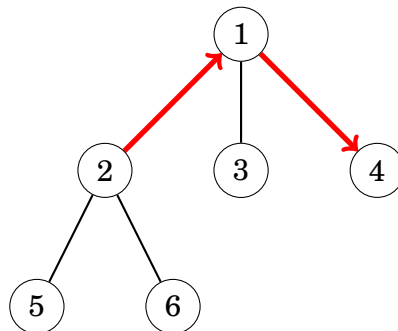


Esta parte es fácil de resolver en tiempo  $O(n)$ , porque podemos usar programación dinámica como lo hemos hecho anteriormente.

Luego, la segunda parte del problema es calcular para cada nodo  $x$  la longitud máxima de una ruta a través de su padre  $p$ . Por ejemplo, la ruta más larga desde el nodo 3 pasa por su padre 1:



A primera vista, parece que deberíamos elegir la ruta más larga desde  $p$ . Sin embargo, esto *no* siempre funciona, porque la ruta más larga desde  $p$  puede pasar por  $x$ . Aquí hay un ejemplo de esta situación:



Aún así, podemos resolver la segunda parte en tiempo  $O(n)$  almacenando *dos* longitudes máximas para cada nodo  $x$ :

- $\text{maxLength}_1(x)$ : la longitud máxima de una ruta desde  $x$
- $\text{maxLength}_2(x)$  la longitud máxima de una ruta desde  $x$  en otra dirección que la primera ruta



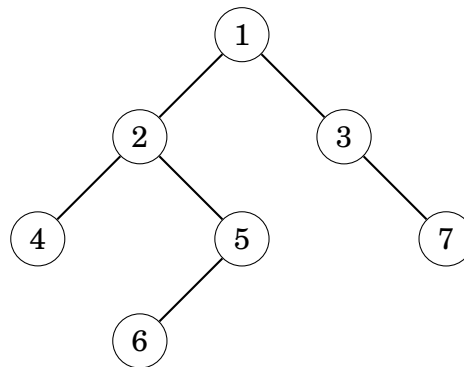
Por ejemplo, en el gráfico anterior,  $\text{maxLength}_1(1) = 2$  usando la ruta  $1 \rightarrow 2 \rightarrow 5$ , y  $\text{maxLength}_2(1) = 1$  usando la ruta  $1 \rightarrow 3$ .

Finalmente, si la ruta que corresponde a  $\text{maxLength}_1(p)$  pasa por  $x$ , concluimos que la longitud máxima es  $\text{maxLength}_2(p) + 1$ , y de lo contrario la longitud máxima es  $\text{maxLength}_1(p) + 1$ .

## 14.4 Árboles binarios

Un **árbol binario** es un árbol enraizado donde cada nodo tiene un subárbol izquierdo y derecho. Es posible que un subárbol de un nodo esté vacío. Por lo tanto, cada nodo en un árbol binario tiene cero, uno o dos hijos.

Por ejemplo, el siguiente árbol es un árbol binario:



Los nodos de un árbol binario tienen tres ordenamientos naturales que corresponden a diferentes formas de recorrer el árbol recursivamente:

- **preorden:** primero procesa la raíz, luego recorre el subárbol izquierdo, luego recorre el subárbol derecho
- **en orden:** primero recorre el subárbol izquierdo, luego procesa la raíz, luego recorre el subárbol derecho
- **posorden:** primero recorre el subárbol izquierdo, luego recorre el subárbol derecho, luego procesa la raíz

Para el árbol anterior, los nodos en preorden son  $[1, 2, 4, 5, 6, 3, 7]$ , en en orden  $[4, 2, 6, 5, 1, 3, 7]$  y en posorden  $[4, 6, 5, 2, 7, 3, 1]$ .

Si sabemos el preorden y el en orden de un árbol, podemos reconstruir la estructura exacta del árbol. Por ejemplo, el árbol anterior es el único árbol posible con preorden  $[1, 2, 4, 5, 6, 3, 7]$  y en orden  $[4, 2, 6, 5, 1, 3, 7]$ . De manera similar, el posorden y el en orden también determinan la estructura de un árbol.

Sin embargo, la situación es diferente si solo conocemos el preorden y el posorden de un árbol. En este caso, puede haber más de un árbol que coincida con los ordenamientos. Por ejemplo, en ambos árboles



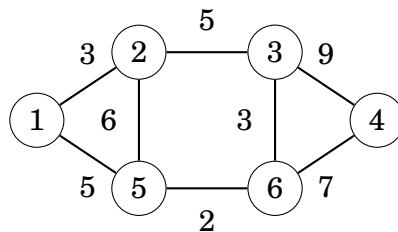
el preorden es  $[1, 2]$  y el postorden es  $[2, 1]$ , pero las estructuras de los árboles son diferentes.

# Chapter 15

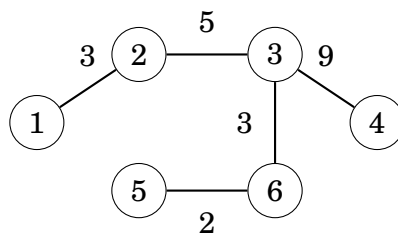
## Árboles de expansión

Un **árbol de expansión** de un grafo consiste en todos los nodos del grafo y algunas de las aristas del grafo de modo que exista un camino entre dos nodos cualesquiera. Al igual que los árboles en general, los árboles de expansión son conectados y acíclicos. Por lo general, hay varias formas de construir un árbol de expansión.

Por ejemplo, considere el siguiente grafo:

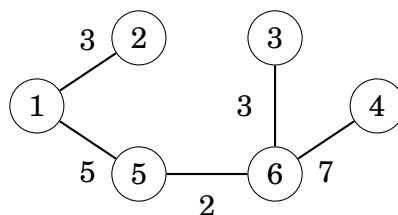


Un árbol de expansión para el grafo es el siguiente:

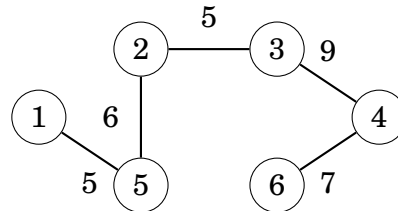


El peso de un árbol de expansión es la suma de los pesos de sus aristas. Por ejemplo, el peso del árbol de expansión anterior es  $3 + 5 + 9 + 3 + 2 = 22$ .

Un **árbol de expansión mínimo** es un árbol de expansión cuyo peso es lo más pequeño posible. El peso de un árbol de expansión mínimo para el grafo de ejemplo es 20, y dicho árbol puede construirse de la siguiente manera:



De manera similar, un **árbol de expansión máximo** es un árbol de expansión cuyo peso es lo más grande posible. El peso de un árbol de expansión máximo para el grafo de ejemplo es 32:



Tenga en cuenta que un grafo puede tener varios árboles de expansión mínimos y máximos, por lo que los árboles no son únicos.

Resulta que varios métodos voraces se pueden utilizar para construir árboles de expansión mínimos y máximos. En este capítulo, analizamos dos algoritmos que procesan las aristas del grafo ordenadas por sus pesos. Nos centramos en encontrar árboles de expansión mínimos, pero los mismos algoritmos pueden encontrar árboles de expansión máximos procesando las aristas en orden inverso.

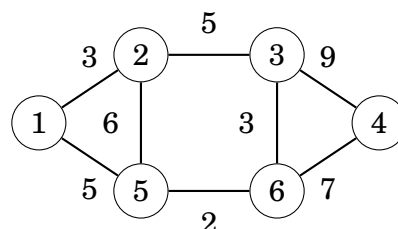
## 15.1 Algoritmo de Kruskal

En el **algoritmo de Kruskal**<sup>1</sup>, el árbol de expansión inicial solo contiene los nodos del grafo y no contiene ninguna arista. Luego, el algoritmo recorre las aristas ordenado por sus pesos, y siempre agrega una arista al árbol si no crea un ciclo.

El algoritmo mantiene los componentes del árbol. Inicialmente, cada nodo del grafo pertenece a un componente separado. Siempre que se agrega una arista al árbol, se unen dos componentes. Finalmente, todos los nodos pertenecen al mismo componente, y se ha encontrado un árbol de expansión mínimo.

### Ejemplo

Consideremos cómo el algoritmo de Kruskal procesa el siguiente gráfico:



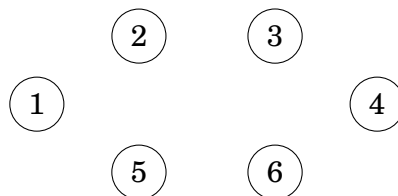
El primer paso del algoritmo es ordenar los bordes en orden creciente de sus pesos. El resultado es la siguiente lista:

<sup>1</sup>El algoritmo fue publicado en 1956 por J. B. Kruskal [48].

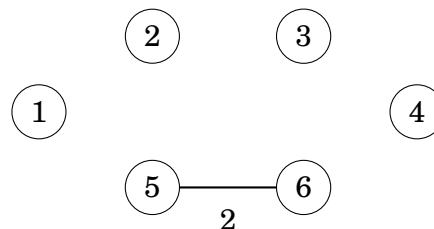
borde	peso
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Después de esto, el algoritmo recorre la lista y agrega cada borde al árbol si une dos componentes separados.

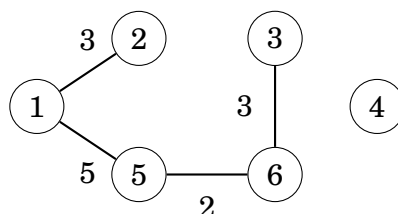
Inicialmente, cada nodo está en su propio componente:



El primer borde que se agregará al árbol es el borde 5-6 que crea un componente {5,6} al unir los componentes {5} y {6}:



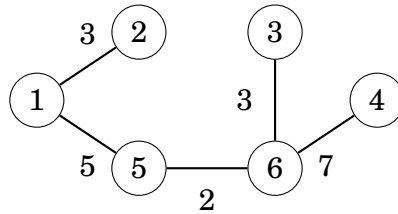
Después de esto, los bordes 1-2, 3-6 y 1-5 se agregan de manera similar:



Después de esos pasos, la mayoría de los componentes se han unido y hay dos componentes en el árbol: {1,2,3,5,6} y {4}.

El siguiente borde en la lista es el borde 2-3, pero no se incluirá en el árbol, porque los nodos 2 y 3 ya están en el mismo componente. Por la misma razón, el borde 2-5 no se incluirá en el árbol.

Finalmente, el borde 4–6 se incluirá en el árbol:

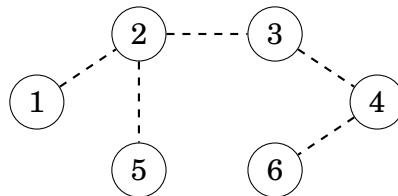


Después de esto, el algoritmo no agregará ningún borde nuevo, porque el gráfico está conectado y hay una ruta entre dos nodos. El gráfico resultante es un árbol de expansión mínimo con peso  $2 + 3 + 3 + 5 + 7 = 20$ .

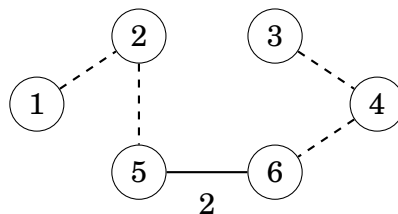
### ¿Por qué funciona esto?

Es una buena pregunta por qué funciona el algoritmo de Kruskal. ¿Por qué la estrategia codiciosa garantiza que nosotros encontremos un árbol de expansión mínimo?

Veamos qué pasa si el borde de peso mínimo de el gráfico *no* está incluido en el árbol de expansión. Por ejemplo, supongamos que un árbol de expansión para el gráfico anterior no contendría el borde de peso mínimo 5–6. No sabemos la estructura exacta de tal árbol de expansión, pero en cualquier caso tiene que contener algunos bordes. Asuma que el árbol sería como sigue:



Sin embargo, no es posible que el árbol anterior sería un árbol de expansión mínimo para el gráfico. La razón de esto es que podemos eliminar un borde del árbol y reemplazarlo con el borde de peso mínimo 5–6. Esto produce un árbol de expansión cuyo peso es *más pequeño*:



Por esta razón, siempre es óptimo incluir el borde de peso mínimo en el árbol para producir un árbol de expansión mínimo. Usando un argumento similar, podemos mostrar que es también es óptimo agregar el siguiente borde en orden de peso al árbol, y así sucesivamente. Por lo tanto, el algoritmo de Kruskal funciona correctamente y siempre produce un árbol de expansión mínimo.

## Implementación

Al implementar el algoritmo de Kruskal, es conveniente usar la representación de la lista de bordes del gráfico. La primera fase del algoritmo ordena la bordes en la lista en  $O(m \log m)$  tiempo. Después de esto, la segunda fase del algoritmo construye el árbol de expansión mínimo como sigue:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

El bucle recorre los bordes de la lista y siempre procesa un borde  $a-b$  donde  $a$  y  $b$  son dos nodos. Se necesitan dos funciones: la función `same` determina si  $a$  y  $b$  están en el mismo componente, y la función `unite` une los componentes que contienen  $a$  y  $b$ .

El problema es cómo implementar eficientemente las funciones `same` y `unite`. Una posibilidad es implementar la función `same` como un recorrido del gráfico y verificar si podemos obtener del nodo  $a$  al nodo  $b$ . Sin embargo, la complejidad temporal de tal función sería  $O(n + m)$  y el algoritmo resultante sería lento, porque la función `same` se llamará para cada borde del gráfico.

Resolveremos el problema utilizando una estructura de unión-búsqueda que implementa ambas funciones en  $O(\log n)$  tiempo. Por lo tanto, la complejidad temporal del algoritmo de Kruskal será  $O(m \log n)$  después de ordenar la lista de bordes.

## 15.2 Estructura de unión-búsqueda

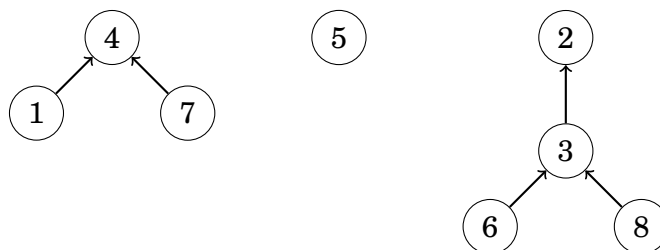
Una **estructura de unión-búsqueda** mantiene una colección de conjuntos. Los conjuntos son disjuntos, por lo que ningún elemento pertenece a más de un conjunto. Se admiten dos operaciones de tiempo  $O(\log n)$ : la operación `unite` une dos conjuntos, y la operación `find` encuentra el representante del conjunto que contiene un elemento dado<sup>2</sup>.

### Estructura

En una estructura de unión-búsqueda, un elemento en cada conjunto es el representante del conjunto, y hay una cadena desde cualquier otro elemento del conjunto hasta el representante. Por ejemplo, suponga que los conjuntos son  $\{1, 4, 7\}$ ,  $\{5\}$  y  $\{2, 3, 6, 8\}$ :

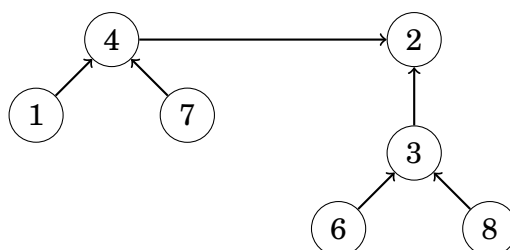
---

<sup>2</sup>La estructura presentada aquí fue introducida en 1971 por J. D. Hopcroft y J. D. Ullman [38]. Más tarde, en 1975, R. E. Tarjan estudió una variante más sofisticada de la estructura [64] que se discute en muchos algoritmos libros de texto hoy en día.



En este caso, los representantes de los conjuntos son 4, 5 y 2. Podemos encontrar el representante de cualquier elemento siguiendo la cadena que comienza en el elemento. Por ejemplo, el elemento 2 es el representante para el elemento 6, porque seguimos la cadena  $6 \rightarrow 3 \rightarrow 2$ . Dos elementos pertenecen al mismo conjunto exactamente cuando sus representantes son los mismos.

Dos conjuntos se pueden unir conectando el representante de un conjunto al representante del otro conjunto. Por ejemplo, los conjuntos  $\{1, 4, 7\}$  y  $\{2, 3, 6, 8\}$  se pueden unir de la siguiente manera:



El conjunto resultante contiene los elementos  $\{1, 2, 3, 4, 6, 7, 8\}$ . A partir de este momento, el elemento 2 es el representante de todo el conjunto y el antiguo representante 4 apunta al elemento 2.

La eficiencia de la estructura de unión-búsqueda depende de cómo se unen los conjuntos. Resulta que podemos seguir una estrategia simple: siempre conectar el representante del conjunto *más pequeño* al representante del conjunto *más grande* (o si los conjuntos son del mismo tamaño, podemos hacer una elección arbitraria). Usando esta estrategia, la longitud de cualquier cadena será  $O(\log n)$ , por lo que podemos encontrar el representante de cualquier elemento eficientemente siguiendo la cadena correspondiente.

## Implementación

La estructura de unión-búsqueda se puede implementar usando matrices. En la siguiente implementación, la matriz `link` contiene para cada elemento el siguiente elemento en la cadena o el elemento mismo si es un representante, y la matriz `size` indica para cada representante el tamaño del conjunto correspondiente.

Inicialmente, cada elemento pertenece a un conjunto separado:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

La función `find` devuelve el representante para un elemento  $x$ . El representante se puede encontrar siguiendo la cadena que comienza en  $x$ .



```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

La función `same` comprueba si los elementos  $a$  y  $b$  pertenecen al mismo conjunto. Esto se puede hacer fácilmente utilizando el función `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

La función `unite` une los conjuntos que contienen los elementos  $a$  y  $b$  (los elementos deben estar en conjuntos diferentes). La función primero encuentra los representantes de los conjuntos y luego conecta el más pequeño conjunto al conjunto más grande.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

La complejidad temporal de la función `find` es  $O(\log n)$  asumiendo que la longitud de cada cadena es  $O(\log n)$ . En este caso, las funciones `same` y `unite` también funcionan en tiempo  $O(\log n)$ . La función `unite` se asegura de que el longitud de cada cadena es  $O(\log n)$  conectando el conjunto más pequeño al conjunto más grande.

## 15.3 Algoritmo de Prim

**Algoritmo de Prim**<sup>3</sup> es un método alternativo para encontrar un árbol de expansión mínimo. El algoritmo primero agrega un nodo arbitrario al árbol. Después de esto, el algoritmo siempre elige un borde de peso mínimo que agrega un nuevo nodo al árbol. Finalmente, todos los nodos han sido agregados al árbol y se ha encontrado un árbol de expansión mínimo.

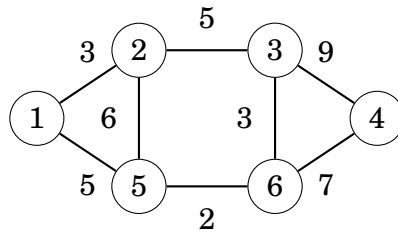
El algoritmo de Prim se parece al algoritmo de Dijkstra. La diferencia es que el algoritmo de Dijkstra siempre selecciona una arista cuya distancia desde el nodo de inicio es mínima, pero el algoritmo de Prim simplemente selecciona la arista de peso mínimo que agrega un nuevo nodo al árbol.

---

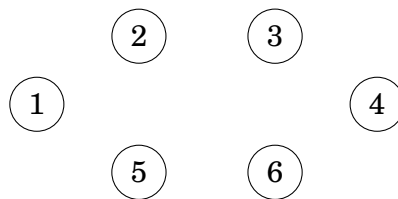
<sup>3</sup>El algoritmo es nombrado después de R. C. Prim quien lo publicó en 1957 [54]. Sin embargo, el mismo algoritmo ya fue descubierto en 1930 por V. Jarník.

## Ejemplo

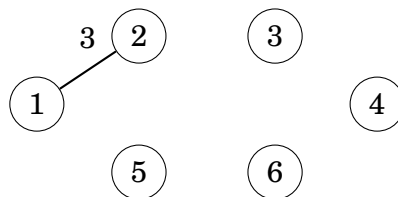
Consideremos cómo funciona el algoritmo de Prim en el siguiente gráfico:



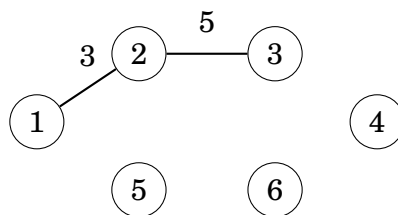
Inicialmente, no hay aristas entre los nodos:



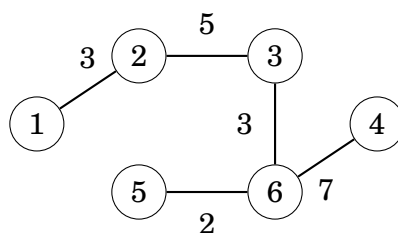
Un nodo arbitrario puede ser el nodo de inicio, así que elijamos el nodo 1. Primero, agregamos el nodo 2 que está conectado por una arista de peso 3:



Después de esto, hay dos aristas con peso 5, por lo que podemos agregar el nodo 3 o el nodo 5 al árbol. Agreguemos el nodo 3 primero:



El proceso continúa hasta que todos los nodos se han incluido en el árbol:



## Implementación

Al igual que el algoritmo de Dijkstra, el algoritmo de Prim se puede implementar de manera eficiente utilizando una cola de prioridad. La cola de prioridad debe contener todos los nodos que se pueden conectar al componente actual utilizando un solo borde, en orden creciente de los pesos de los bordes correspondientes.

La complejidad temporal del algoritmo de Prim es  $O(n + m \log m)$  que es igual a la complejidad temporal del algoritmo de Dijkstra. En la práctica, los algoritmos de Prim y Kruskal son ambos eficientes, y la elección del algoritmo es una cuestión de gusto. Aún así, la mayoría de los programadores competitivos utilizan el algoritmo de Kruskal.



# Chapter 16

## Grafos dirigidos

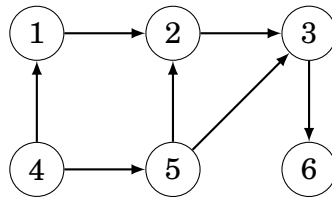
En este capítulo, nos centramos en dos clases de grafos dirigidos:

- **Grafos acíclicos:** No hay ciclos en el grafo, por lo que no hay camino desde ningún nodo hasta sí mismo<sup>1</sup>.
- **Grafos sucesores:** El grado de salida de cada nodo es 1, por lo que cada nodo tiene un sucesor único.

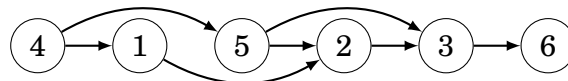
Resulta que en ambos casos, podemos diseñar algoritmos eficientes que se basan en las propiedades especiales de los grafos.

### 16.1 Ordenación topológica

Una **ordenación topológica** es una ordenación de los nodos de un grafo dirigido tal que si hay un camino desde el nodo  $a$  hasta el nodo  $b$ , entonces el nodo  $a$  aparece antes que el nodo  $b$  en la ordenación. Por ejemplo, para el grafo



una ordenación topológica es [4, 1, 5, 2, 3, 6]:



Un grafo acíclico siempre tiene una ordenación topológica. Sin embargo, si el grafo contiene un ciclo, no es posible formar una ordenación topológica, porque ningún nodo del ciclo puede aparecer antes que los demás nodos del ciclo en la ordenación. Resulta que la búsqueda en profundidad se puede utilizar para comprobar si un grafo dirigido contiene un ciclo y, si no contiene un ciclo, para construir una ordenación topológica.

<sup>1</sup>Los grafos acíclicos dirigidos a veces se denominan DAGs.

## Algoritmo

La idea es recorrer los nodos del grafo y siempre comenzar una búsqueda en profundidad en el nodo actual si aún no se ha procesado. Durante las búsquedas, los nodos tienen tres estados posibles:

- estado 0: el nodo no se ha procesado (blanco)
- estado 1: el nodo está en proceso (gris claro)
- estado 2: el nodo se ha procesado (gris oscuro)

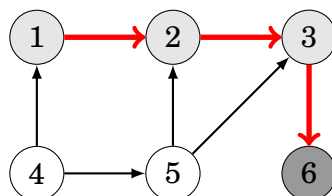
Inicialmente, el estado de cada nodo es 0. Cuando una búsqueda llega a un nodo por primera vez, su estado pasa a ser 1. Finalmente, después de que todos los sucesores del nodo hayan sido procesados, su estado pasa a ser 2.

Si el grafo contiene un ciclo, nos daremos cuenta de esto durante la búsqueda, porque tarde o temprano llegaremos a un nodo cuyo estado es 1. En este caso, no es posible construir una ordenación topológica.

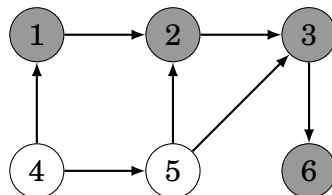
Si el grafo no contiene un ciclo, podemos construir una ordenación topológica agregando cada nodo a una lista cuando el estado del nodo pasa a ser 2. Esta lista en orden inverso es una ordenación topológica.

## Ejemplo 1

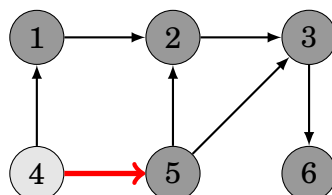
En el grafo de ejemplo, la búsqueda primero avanza desde el nodo 1 hasta el nodo 6:



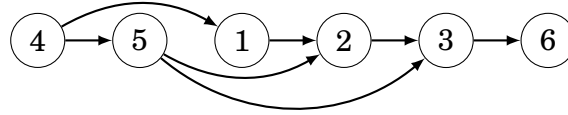
Ahora el nodo 6 se ha procesado, por lo que se agrega a la lista. Después de esto, también los nodos 3, 2 y 1 se agregan a la lista:



En este punto, la lista es [6, 3, 2, 1]. La próxima búsqueda comienza en el nodo 4:



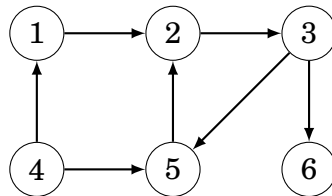
Por lo tanto, la lista final es [6,3,2,1,5,4]. Hemos procesado todos los nodos, por lo que se ha encontrado una ordenación topológica. La ordenación topológica es la lista inversa [4,5,1,2,3,6]:



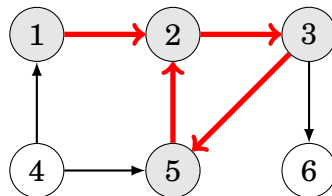
Tenga en cuenta que una ordenación topológica no es única, y puede haber varias ordenaciones topológicas para un gráfico.

## Ejemplo 2

Consideremos ahora un gráfico para el cual no podemos construir una ordenación topológica, porque el gráfico contiene un ciclo:



La búsqueda procede de la siguiente manera:



La búsqueda llega al nodo 2 cuyo estado es 1, lo que significa que el gráfico contiene un ciclo. En este ejemplo, hay un ciclo  $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ .

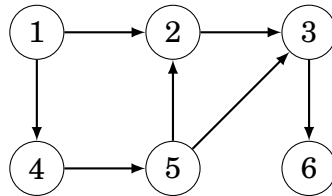
## 16.2 Programación dinámica

Si un gráfico dirigido es acíclico, la programación dinámica se puede aplicar a él. Por ejemplo, podemos resolver eficientemente los siguientes problemas relacionados con las rutas desde un nodo de inicio hasta un nodo final:

- ¿Cuántas rutas diferentes hay?
- ¿Cuál es la ruta más corta/más larga?
- ¿Cuál es el número mínimo/máximo de aristas en una ruta?
- ¿Qué nodos aparecen ciertamente en cualquier ruta?

## Contando el número de rutas

Como ejemplo, calculemos el número de rutas desde el nodo 1 al nodo 6 en el siguiente gráfico:



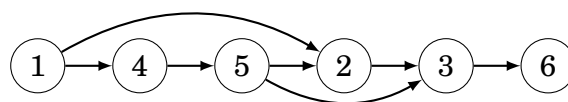
Hay un total de tres rutas de este tipo:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

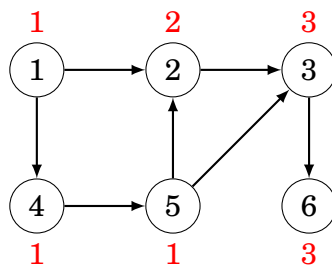
Sea  $\text{paths}(x)$  el número de rutas desde el nodo 1 al nodo  $x$ . Como caso base,  $\text{paths}(1) = 1$ . Entonces, para calcular otros valores de  $\text{paths}(x)$ , podemos usar la recursión

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \cdots + \text{paths}(a_k)$$

donde  $a_1, a_2, \dots, a_k$  son los nodos desde los que hay una arista a  $x$ . Dado que el gráfico es acíclico, los valores de  $\text{paths}(x)$  se pueden calcular en el orden de una clasificación topológica. Una clasificación topológica para el gráfico anterior es la siguiente:



Por lo tanto, los números de rutas son los siguientes:

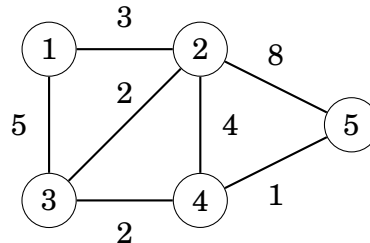


Por ejemplo, para calcular el valor de  $\text{paths}(3)$ , podemos usar la fórmula  $\text{paths}(2) + \text{paths}(5)$ , porque hay aristas desde los nodos 2 y 5 al nodo 3. Dado que  $\text{paths}(2) = 2$  y  $\text{paths}(5) = 1$ , concluimos que  $\text{paths}(3) = 3$ .

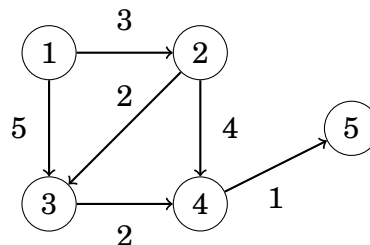


## Extensión del algoritmo de Dijkstra

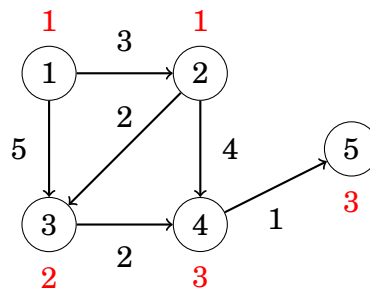
Un subproducto del algoritmo de Dijkstra es un gráfico dirigido y acíclico que indica para cada nodo del gráfico original las formas posibles de alcanzar el nodo utilizando una ruta más corta desde el nodo de inicio. La programación dinámica se puede aplicar a ese gráfico. Por ejemplo, en el gráfico



las rutas más cortas desde el nodo 1 pueden usar las siguientes aristas:



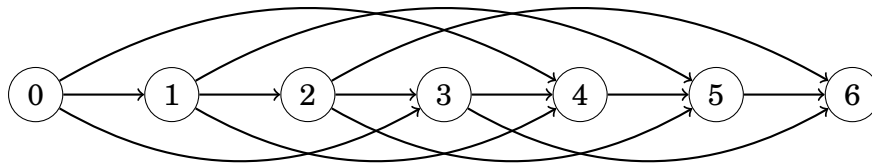
Ahora podemos, por ejemplo, calcular el número de caminos más cortos desde el nodo 1 hasta el nodo 5 utilizando programación dinámica:



## Representando problemas como grafos

En realidad, cualquier problema de programación dinámica puede representarse como un grafo dirigido y acíclico. En tal grafo, cada nodo corresponde a un estado de programación dinámica y las aristas indican cómo los estados dependen unos de otros.

Como ejemplo, considere el problema de formar una suma de dinero  $n$  utilizando monedas  $\{c_1, c_2, \dots, c_k\}$ . En este problema, podemos construir un grafo donde cada nodo corresponde a una suma de dinero, y las aristas muestran cómo se pueden elegir las monedas. Por ejemplo, para las monedas  $\{1, 3, 4\}$  y  $n = 6$ , el grafo es el siguiente:



Usando esta representación, el camino más corto desde el nodo 0 hasta el nodo  $n$  corresponde a una solución con el número mínimo de monedas, y el número total de caminos desde el nodo 0 hasta el nodo  $n$  es igual al número total de soluciones.

## 16.3 Caminos sucesores

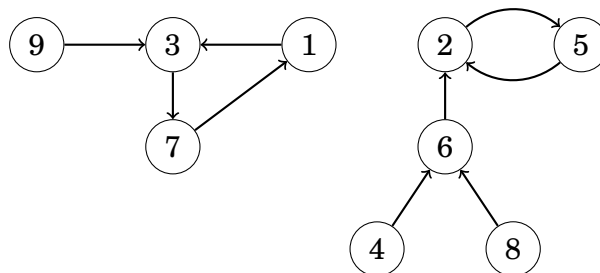
Para el resto del capítulo, nos centraremos en los **grafos sucesores**. En esos grafos, el grado de salida de cada nodo es 1, es decir, exactamente una arista comienza en cada nodo. Un grafo sucesor consiste en uno o más componentes, cada uno de los cuales contiene un ciclo y algunos caminos que conducen a él.

Los grafos sucesores a veces se llaman **grafos funcionales**. La razón de esto es que cualquier grafo sucesor corresponde a una función que define las aristas del grafo. El parámetro de la función es un nodo del grafo, y la función da el sucesor de ese nodo.

Por ejemplo, la función

$x$	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

define el siguiente grafo:



Dado que cada nodo de un grafo sucesor tiene un sucesor único, también podemos definir una función  $\text{succ}(x, k)$  que da el nodo al que llegaremos si comenzamos en el nodo  $x$  y caminamos  $k$  pasos hacia adelante. Por ejemplo, en el grafo anterior  $\text{succ}(4, 6) = 2$ , porque llegaremos al nodo 2 caminando 6 pasos desde el nodo 4:



Una forma sencilla de calcular un valor de  $\text{succ}(x, k)$  es comenzar en el nodo  $x$  y caminar  $k$  pasos hacia adelante, lo que lleva  $O(k)$  tiempo. Sin embargo,

utilizando el preprocesamiento, cualquier valor de  $\text{succ}(x, k)$  se puede calcular en solo  $O(\log k)$  tiempo.

La idea es precalcular todos los valores de  $\text{succ}(x, k)$  donde  $k$  es una potencia de dos y como máximo  $u$ , donde  $u$  es el número máximo de pasos que alguna vez caminaremos. Esto se puede hacer de manera eficiente, porque podemos usar la siguiente recursión:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Precalcular los valores lleva  $O(n \log u)$  tiempo, porque se calculan  $O(\log u)$  valores para cada nodo. En el gráfico anterior, los primeros valores son los siguientes:

$x$	1	2	3	4	5	6	7	8	9
$\text{succ}(x, 1)$	3	5	7	6	2	2	1	6	3
$\text{succ}(x, 2)$	7	2	1	2	5	5	3	2	7
$\text{succ}(x, 4)$	3	2	7	2	5	5	1	2	3
$\text{succ}(x, 8)$	7	2	1	2	5	5	3	2	7
...									

Después de esto, cualquier valor de  $\text{succ}(x, k)$  se puede calcular presentando el número de pasos  $k$  como una suma de potencias de dos. Por ejemplo, si queremos calcular el valor de  $\text{succ}(x, 11)$ , primero formamos la representación  $11 = 8 + 2 + 1$ . Usando eso,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Por ejemplo, en el gráfico anterior

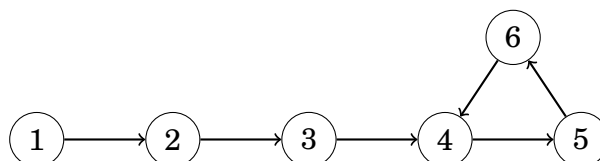
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Tal representación siempre consiste en  $O(\log k)$  partes, por lo que calcular un valor de  $\text{succ}(x, k)$  lleva  $O(\log k)$  tiempo.

## 16.4 Detección de ciclos

Considere un gráfico sucesor que solo contiene una ruta que termina en un ciclo. Podemos hacer las siguientes preguntas: si comenzamos nuestra caminata en el nodo de inicio, ¿cuál es el primer nodo en el ciclo? ¿y cuántos nodos contiene el ciclo?

Por ejemplo, en el gráfico



comenzamos nuestra caminata en el nodo 1, el primer nodo que pertenece al ciclo es el nodo 4, y el ciclo consiste en tres nodos (4, 5 y 6).

Una forma sencilla de detectar el ciclo es caminar en el gráfico y hacer un seguimiento de todos los nodos que se han visitado. Una vez que un nodo se visita por segunda vez, podemos concluir que el nodo es el primer nodo en el ciclo. Este método funciona en  $O(n)$  tiempo y también usa  $O(n)$  memoria.

Sin embargo, existen mejores algoritmos para la detección de ciclos. La complejidad temporal de tales algoritmos sigue siendo  $O(n)$ , pero solo usan  $O(1)$  memoria. Esta es una mejora importante si  $n$  es grande. A continuación, discutiremos el algoritmo de Floyd que alcanza estas propiedades.

## Algoritmo de Floyd

**Algoritmo de Floyd**<sup>2</sup> camina hacia adelante en el gráfico utilizando dos punteros  $a$  y  $b$ . Ambos punteros comienzan en un nodo  $x$  que es el nodo de inicio del gráfico. Luego, en cada turno, el puntero  $a$  camina un paso adelante y el puntero  $b$  camina dos pasos adelante. El proceso continúa hasta que los punteros se encuentran entre sí:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

En este punto, el puntero  $a$  ha caminado  $k$  pasos y el puntero  $b$  ha caminado  $2k$  pasos, por lo que la longitud del ciclo divide  $k$ . Por lo tanto, el primer nodo que pertenece al ciclo se puede encontrar moviendo el puntero  $a$  al nodo  $x$  y avanzando los punteros paso a paso hasta que se encuentren de nuevo.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

Después de esto, la longitud del ciclo se puede calcular de la siguiente manera:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

---

<sup>2</sup>La idea del algoritmo se menciona en [46] y se atribuye a R. W. Floyd; sin embargo, no se sabe si Floyd realmente descubrió el algoritmo.

---

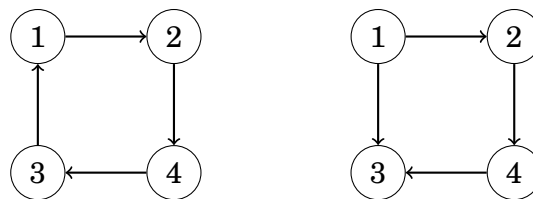


# Chapter 17

## Conectividad fuerte

En un grafo dirigido, los bordes se pueden recorrer en una sola dirección, por lo que incluso si el grafo está conectado, esto no garantiza que haya un camino de un nodo a otro nodo. Por esta razón, es significativo definir un nuevo concepto que requiere más que conectividad.

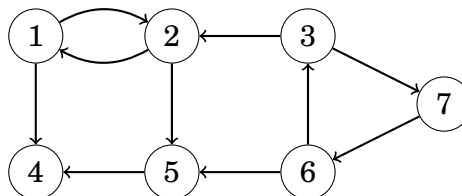
Un grafo es **fuertemente conexo** si hay un camino desde cualquier nodo a todos los demás nodos del grafo. Por ejemplo, en la siguiente imagen, el grafo izquierdo está fuertemente conectado mientras que el grafo derecho no.



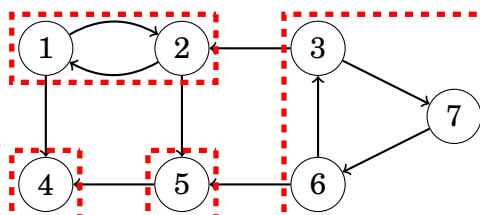
El grafo derecho no está fuertemente conectado porque, por ejemplo, no hay ningún camino desde el nodo 2 al nodo 1.

Los **componentes fuertemente conexos** de un grafo dividen el grafo en partes fuertemente conectadas que son lo más grandes posible. Los componentes fuertemente conexos forman un grafo **de componentes** acíclico que representa la estructura profunda del grafo original.

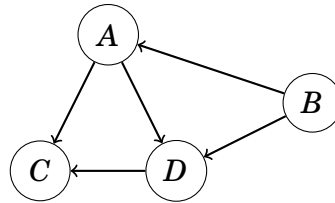
Por ejemplo, para el grafo



los componentes fuertemente conexos son los siguientes:



El grafo de componentes correspondiente es el siguiente:



Los componentes son  $A = \{1, 2\}$ ,  $B = \{3, 6, 7\}$ ,  $C = \{4\}$  y  $D = \{5\}$ .

Un grafo de componentes es un grafo dirigido acíclico, por lo que es más fácil de procesar que el grafo original. Dado que el grafo no contiene ciclos, siempre podemos construir una ordenación topológica y utilizar técnicas de programación dinámica como las que se presentan en el Capítulo 16.

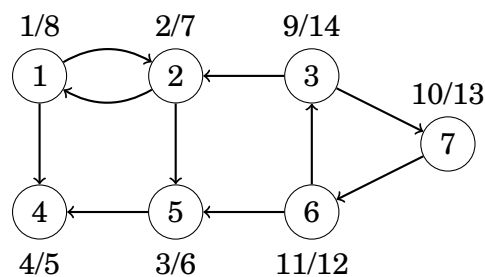
## 17.1 Algoritmo de Kosaraju

El **algoritmo de Kosaraju**<sup>1</sup> es un método eficiente para encontrar los componentes fuertemente conexos de un grafo dirigido. El algoritmo realiza dos búsquedas en profundidad: la primera búsqueda construye una lista de nodos de acuerdo con la estructura del grafo, y la segunda búsqueda forma los componentes fuertemente conexos.

### Búsqueda 1

La primera fase del algoritmo de Kosaraju construye una lista de nodos en el orden en que una búsqueda en profundidad los procesa. El algoritmo recorre los nodos, y comienza una búsqueda en profundidad en cada nodo sin procesar. Cada nodo se agregará a la lista después de haber sido procesado.

En el gráfico de ejemplo, los nodos se procesan en el siguiente orden:



La notación  $x/y$  significa que el procesamiento del nodo comenzó en el tiempo  $x$  y terminó en el tiempo  $y$ . Por lo tanto, la lista correspondiente es la siguiente:

<sup>1</sup>Según [1], S. R. Kosaraju inventó este algoritmo en 1978 pero no lo publicó. En 1981, el mismo algoritmo fue redescubierto y publicado por M. Sharir [57].

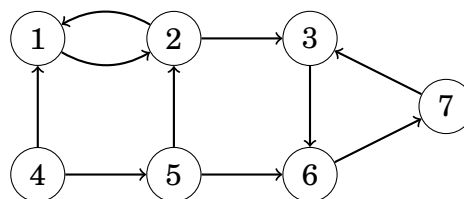


nodo	tiempo de procesamiento
4	5
5	6
2	7
1	8
6	12
7	13
3	14

## Búsqueda 2

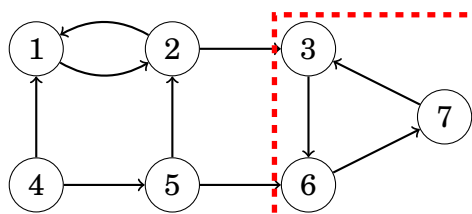
La segunda fase del algoritmo forma los componentes fuertemente conectados del gráfico. Primero, el algoritmo invierte cada borde en el gráfico. Esto garantiza que durante la segunda búsqueda, siempre encontraremos componentes fuertemente conectados que no tienen nodos adicionales.

Después de invertir los bordes, el gráfico de ejemplo es el siguiente:



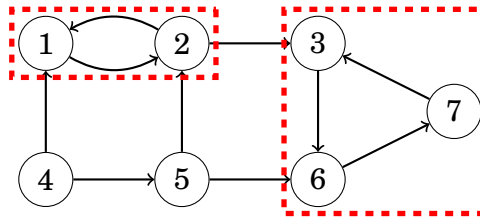
Después de esto, el algoritmo recorre la lista de nodos creada por la primera búsqueda, en orden *inverso*. Si un nodo no pertenece a un componente, el algoritmo crea un nuevo componente y comienza una búsqueda en profundidad que agrega todos los nodos nuevos encontrados durante la búsqueda al nuevo componente.

En el gráfico de ejemplo, el primer componente comienza en el nodo 3:

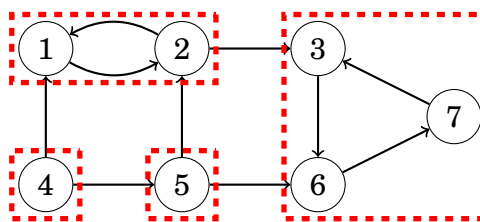


Tenga en cuenta que dado que todos los bordes están invertidos, el componente no "se filtra" a otras partes del gráfico.

Los siguientes nodos en la lista son los nodos 7 y 6, pero ya pertenecen a un componente, por lo que el siguiente nuevo componente comienza en el nodo 1:



Finalmente, el algoritmo procesa los nodos 5 y 4 que crean los componentes fuertemente conectados restantes:



La complejidad temporal del algoritmo es  $O(n+m)$ , porque el algoritmo realiza dos búsquedas en profundidad.

## 17.2 Problema 2SAT

La conectividad fuerte también está relacionada con el **problema 2SAT**<sup>2</sup>. En este problema, se nos da una fórmula lógica

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

donde cada  $a_i$  y  $b_i$  es una variable lógica ( $x_1, x_2, \dots, x_n$ ) o una negación de una variable lógica ( $\neg x_1, \neg x_2, \dots, \neg x_n$ ). Los símbolos " $\wedge$ " y " $\vee$ " denotan operadores lógicos "y" y "o". Nuestra tarea es asignar a cada variable un valor para que la fórmula sea verdadera, o indicar que esto no es posible.

Por ejemplo, la fórmula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

es verdadera cuando las variables se asignan de la siguiente manera:

$$\begin{cases} x_1 = \text{falso} \\ x_2 = \text{falso} \\ x_3 = \text{verdadero} \\ x_4 = \text{verdadero} \end{cases}$$

<sup>2</sup>El algoritmo presentado aquí fue introducido en [4]. También existe otro algoritmo conocido de tiempo lineal [19] que se basa en el retroceso.

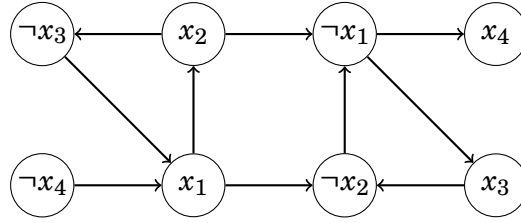
Sin embargo, la fórmula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

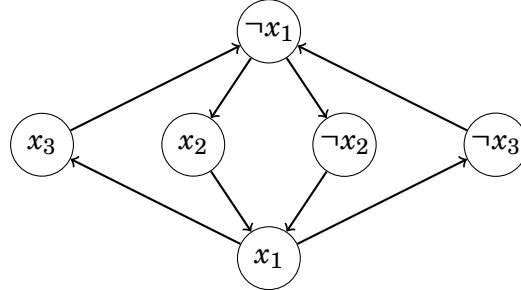
siempre es falsa, independientemente de cómo asignemos los valores. La razón de esto es que no podemos elegir un valor para  $x_1$  sin crear una contradicción. Si  $x_1$  es falso, tanto  $x_2$  como  $\neg x_2$  deberían ser verdaderos, lo cual es imposible, y si  $x_1$  es verdadero, tanto  $x_3$  como  $\neg x_3$  deberían ser verdaderos, lo cual también es imposible.

El problema 2SAT se puede representar como un gráfico cuyos nodos corresponden a variables  $x_i$  y negaciones  $\neg x_i$ , y los bordes determinan las conexiones entre las variables. Cada par  $(a_i \vee b_i)$  genera dos bordes:  $\neg a_i \rightarrow b_i$  y  $\neg b_i \rightarrow a_i$ . Esto significa que si  $a_i$  no se cumple,  $b_i$  debe cumplirse, y viceversa.

El gráfico para la fórmula  $L_1$  es:



Y el gráfico para la fórmula  $L_2$  es:



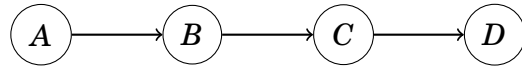
La estructura del grafo nos dice si es posible asignar los valores de las variables para que la fórmula sea verdadera. Resulta que esto se puede hacer exactamente cuando no hay nodos  $x_i$  y  $\neg x_i$  tales que ambos nodos pertenezcan al mismo componente fuertemente conectado. Si hay tales nodos, el grafo contiene un camino desde  $x_i$  a  $\neg x_i$  y también un camino desde  $\neg x_i$  a  $x_i$ , por lo que tanto  $x_i$  como  $\neg x_i$  deben ser verdaderos lo cual no es posible.

En el grafo de la fórmula  $L_1$  no hay nodos  $x_i$  y  $\neg x_i$  tales que ambos nodos pertenezcan al mismo componente fuertemente conectado, por lo que existe una solución. En el grafo de la fórmula  $L_2$  todos los nodos pertenecen al mismo componente fuertemente conectado, por lo que no existe una solución.

Si existe una solución, los valores para las variables se pueden encontrar recorriendo los nodos del grafo de componentes en un orden de clasificación topológica inverso. En cada paso, procesamos un componente que no contiene aristas que conduzcan a un componente no procesado. Si las variables en el

componente no han sido asignadas valores, sus valores serán determinados de acuerdo con los valores en el componente, y si ya tienen valores, permanecen sin cambios. El proceso continúa hasta que cada variable ha sido asignado un valor.

El grafo de componentes para la fórmula  $L_1$  es el siguiente:



Los componentes son  $A = \{\neg x_4\}$ ,  $B = \{x_1, x_2, \neg x_3\}$ ,  $C = \{\neg x_1, \neg x_2, x_3\}$  y  $D = \{x_4\}$ . Al construir la solución, primero procesamos el componente  $D$  donde  $x_4$  se vuelve verdadero. Después de esto, procesamos el componente  $C$  donde  $x_1$  y  $x_2$  se vuelven falsos y  $x_3$  se vuelve verdadero. Todas las variables han sido asignadas valores, por lo que los componentes restantes  $A$  y  $B$  no cambian las variables.

Tenga en cuenta que este método funciona, porque el grafo tiene una estructura especial: si hay caminos desde el nodo  $x_i$  al nodo  $x_j$  y desde el nodo  $x_j$  al nodo  $\neg x_j$ , entonces el nodo  $x_i$  nunca se vuelve verdadero. La razón de esto es que también hay un camino desde el nodo  $\neg x_j$  al nodo  $\neg x_i$ , y tanto  $x_i$  como  $x_j$  se vuelven falsos.

Un problema más difícil es el **3SAT problem**, donde cada parte de la fórmula es de la forma  $(a_i \vee b_i \vee c_i)$ . Este problema es NP-difícil, por lo que no se conoce ningún algoritmo eficiente para resolver el problema.

# Chapter 18

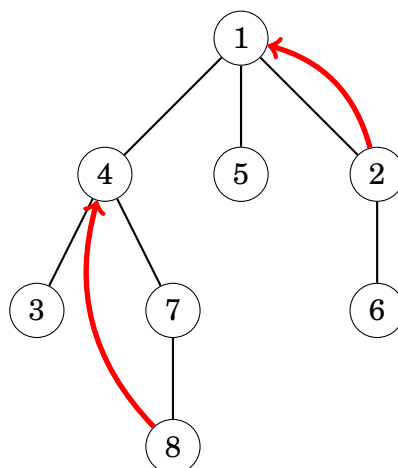
## Consultas de árboles

Este capítulo discute técnicas para procesar consultas en subárboles y caminos de un árbol enraizado. Por ejemplo, tales consultas son:

- ¿Cuál es el  $k$ -ésimo ancestro de un nodo?
- ¿Cuál es la suma de los valores en el subárbol de un nodo?
- ¿Cuál es la suma de los valores en un camino entre dos nodos?
- ¿Cuál es el ancestro común más bajo de dos nodos?

### 18.1 Encontrar ancestros

El  $k$ -ésimo **ancestro** de un nodo  $x$  en un árbol enraizado es el nodo que alcanzaremos si nos movemos  $k$  niveles arriba desde  $x$ . Sea  $\text{ancestro}(x, k)$  denotar el  $k$ -ésimo ancestro de un nodo  $x$  (o 0 si no hay tal ancestro). Por ejemplo, en el siguiente árbol,  $\text{ancestro}(2, 1) = 1$  y  $\text{ancestro}(8, 2) = 4$ .



Una manera fácil de calcular cualquier valor de  $\text{ancestro}(x, k)$  es realizar una secuencia de  $k$  movimientos en el árbol. Sin embargo, la complejidad temporal de este método es  $O(k)$ , que puede ser lenta, porque un árbol de  $n$  nodos puede tener una cadena de  $n$  nodos.

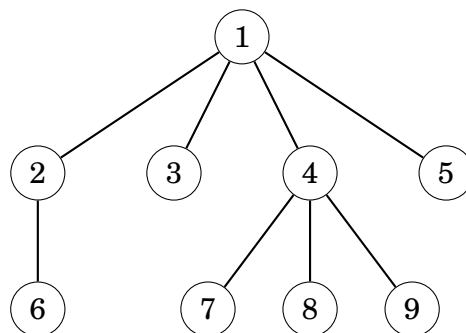
Afortunadamente, usando una técnica similar a la usada en el Capítulo 16.3, cualquier valor de  $\text{ancestro}(x, k)$  puede ser calculado eficientemente en tiempo  $O(\log k)$  después del preprocesamiento. La idea es precalcular todos los valores  $\text{ancestro}(x, k)$  donde  $k \leq n$  es una potencia de dos. Por ejemplo, los valores para el árbol anterior son los siguientes:

$x$	1	2	3	4	5	6	7	8
$\text{ancestro}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestro}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestro}(x, 4)$	0	0	0	0	0	0	0	0
...								

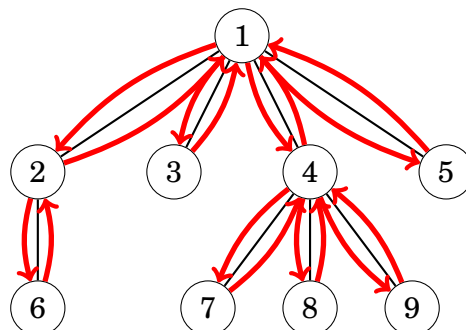
El preprocesamiento toma  $O(n \log n)$  tiempo, porque se calculan  $O(\log n)$  valores para cada nodo. Después de esto, cualquier valor de  $\text{ancestro}(x, k)$  puede ser calculado en tiempo  $O(\log k)$  representando  $k$  como una suma donde cada término es una potencia de dos.

## 18.2 Subárboles y caminos

Un **arreglo de recorrido de árbol** contiene los nodos de un árbol enraizado en el orden en que una búsqueda en profundidad desde el nodo raíz los visita. Por ejemplo, en el árbol



una búsqueda en profundidad procede de la siguiente manera:



Por lo tanto, la matriz de recorrido del árbol correspondiente es la siguiente:

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

## Consultas de subárbol

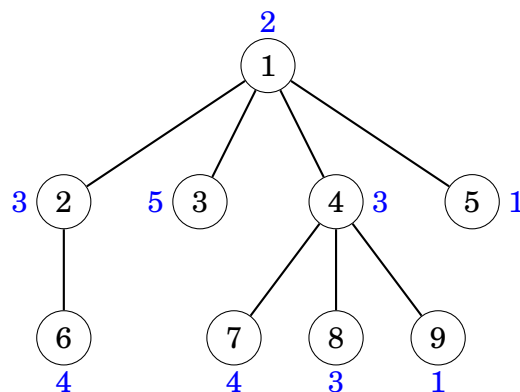
Cada subárbol de un árbol corresponde a una submatriz de la matriz de recorrido del árbol de modo que el primer elemento de la submatriz es el nodo raíz. Por ejemplo, la siguiente submatriz contiene los nodos del subárbol del nodo 4:

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

Usando este hecho, podemos procesar eficientemente consultas que están relacionadas con subárboles de un árbol. Como ejemplo, considere un problema donde cada nodo se le asigna un valor, y nuestra tarea es apoyar las siguientes consultas:

- actualizar el valor de un nodo
- calcular la suma de valores en el subárbol de un nodo

Considere el siguiente árbol donde los números azules son los valores de los nodos. Por ejemplo, la suma del subárbol del nodo 4 es  $3 + 4 + 3 + 1 = 11$ .



La idea es construir una matriz de recorrido del árbol que contiene tres valores para cada nodo: el identificador del nodo, el tamaño del subárbol, y el valor del nodo. Por ejemplo, la matriz para el árbol anterior es la siguiente:

node id	1	2	6	3	4	7	8	9	5
subtree size									
node value	2	3	4	5	3	4	3	1	1

Usando esta matriz, podemos calcular la suma de valores en cualquier subárbol encontrando primero el tamaño del subárbol y luego los valores de los nodos correspondientes. Por ejemplo, los valores en el subárbol del nodo 4 se pueden encontrar de la siguiente manera:

identificador del nodo	1	2	6	3	4	7	8	9	5
tamaño del subárbol	9	2	1	1	4	1	1	1	1
valor del nodo	2	3	4	5	3	4	3	1	1

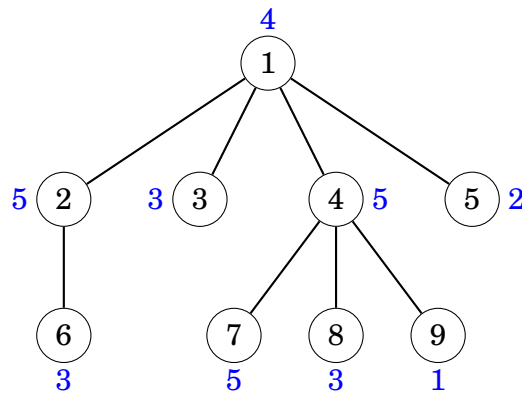
Para responder a las consultas de forma eficiente, basta con almacenar los valores de los nodos en un árbol binario indexado o un árbol de segmentos. Después de esto, podemos actualizar un valor y calcular la suma de los valores en tiempo  $O(\log n)$ .

## Consultas de ruta

Usando una matriz de recorrido de árbol, también podemos calcular de manera eficiente las sumas de valores en rutas desde el nodo raíz hasta cualquier nodo del árbol. Considere un problema donde nuestra tarea es admitir las siguientes consultas:

- cambiar el valor de un nodo
- calcular la suma de valores en una ruta desde la raíz hasta un nodo

Por ejemplo, en el siguiente árbol, la suma de valores desde el nodo raíz hasta el nodo 7 es  $4 + 5 + 5 = 14$ :



Podemos resolver este problema como antes, pero ahora cada valor en la última fila de la matriz es la suma de valores en una ruta desde la raíz hasta el nodo. Por ejemplo, la siguiente matriz corresponde al árbol anterior:

identificador del nodo	1	2	6	3	4	7	8	9	5
tamaño del subárbol	9	2	1	1	4	1	1	1	1
suma de la ruta	4	9	12	7	9	14	12	10	6

Cuando el valor de un nodo aumenta en  $x$ , las sumas de todos los nodos en su subárbol aumentan en  $x$ . Por ejemplo, si el valor del nodo 4 aumenta en 1, la matriz cambia de la siguiente manera:

identificador de nodo	1	2	6	3	4	7	8	9	5
tamaño del subárbol	9	2	1	1	4	1	1	1	1
suma de la ruta	4	9	12	7	10	15	13	11	6

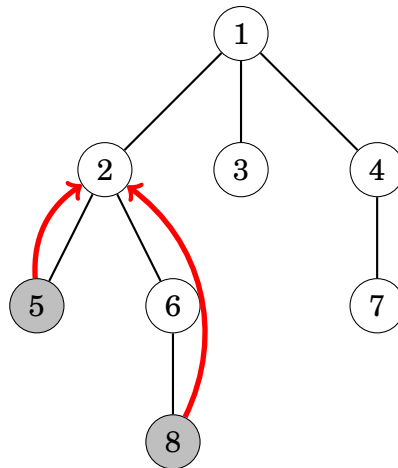


Por lo tanto, para soportar ambas operaciones, deberíamos poder aumentar todos los valores en un rango y recuperar un solo valor. Esto se puede hacer en tiempo  $O(\log n)$  usando un índice binario o un árbol de segmentos (ver Capítulo 9.4).

## 18.3 Antepasado común más bajo

El **antepasado común más bajo** de dos nodos de un árbol enraizado es el nodo más bajo cuyo subárbol contiene ambos nodos. Un problema típico es procesar de manera eficiente las consultas que piden encontrar el antepasado común más bajo de dos nodos.

Por ejemplo, en el siguiente árbol, el antepasado común más bajo de los nodos 5 y 8 es el nodo 2:



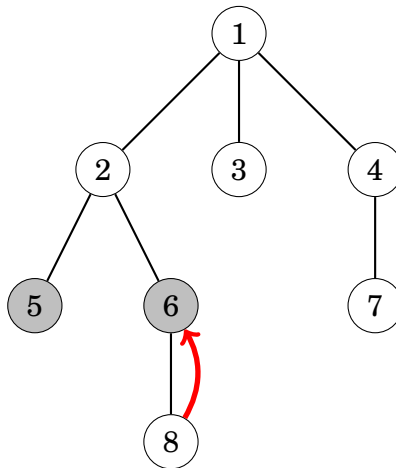
A continuación, discutiremos dos técnicas eficientes para encontrar el antepasado común más bajo de dos nodos.

### Método 1

Una forma de resolver el problema es usar el hecho de que podemos encontrar eficientemente el  $k$ -ésimo antepasado de cualquier nodo en el árbol. Usando esto, podemos dividir el problema de encontrar el antepasado común más bajo en dos partes.

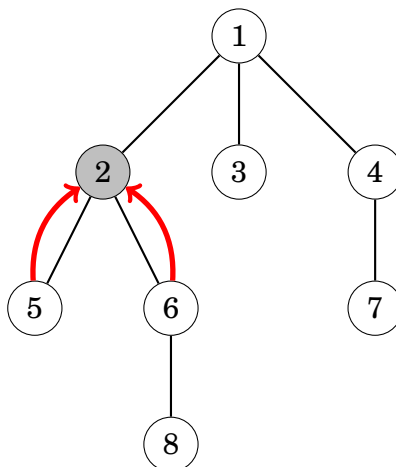
Usamos dos punteros que inicialmente apuntan a los dos nodos cuyo antepasado común más bajo debemos encontrar. Primero, movemos uno de los punteros hacia arriba para que ambos punteros apunten a nodos en el mismo nivel.

En el escenario de ejemplo, movemos el segundo puntero un nivel hacia arriba para que apunte al nodo 6 que está en el mismo nivel que el nodo 5:



Después de esto, determinamos el número mínimo de pasos necesarios para mover ambos punteros hacia arriba para que apunten al mismo nodo. El nodo al que apuntan los punteros después de esto es el antepasado común más bajo.

En el escenario de ejemplo, basta con mover ambos punteros un paso hacia arriba al nodo 2, que es el antepasado común más bajo:

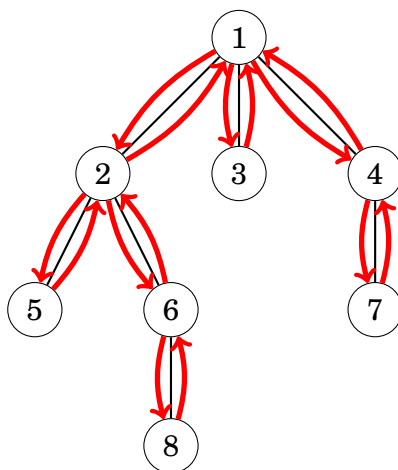


Dado que ambas partes del algoritmo se pueden realizar en tiempo  $O(\log n)$  utilizando información precalculada, podemos encontrar el ancestro común más bajo de dos nodos en tiempo  $O(\log n)$ .

## Método 2

Otra forma de resolver el problema se basa en una matriz de recorrido de árbol<sup>1</sup>. Una vez más, la idea es recorrer los nodos utilizando una búsqueda en profundidad:

<sup>1</sup>Este algoritmo de ancestro común más bajo se presentó en [7]. Esta técnica a veces se llama la técnica de recorrido de Euler **técnica de recorrido de Euler** [66].



Sin embargo, usamos una matriz de recorrido de árbol diferente a la anterior: agregamos cada nodo a la matriz *siempre* cuando la búsqueda en profundidad recorre el nodo, y no solo en la primera visita. Por lo tanto, un nodo que tiene  $k$  hijos aparece  $k + 1$  veces en la matriz y hay un total de  $2n - 1$  nodos en la matriz.

Almacenamos dos valores en la matriz: el identificador del nodo y la profundidad del nodo en el árbol. La siguiente matriz corresponde al árbol anterior:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
identificador del nodo	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
profundidad	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Ahora podemos encontrar el antepasado común más bajo de los nodos  $a$  y  $b$  encontrando el nodo con la profundidad *mínima* entre los nodos  $a$  y  $b$  en el array. Por ejemplo, el antepasado común más bajo de los nodos 5 y 8 se puede encontrar de la siguiente manera:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
identificador del nodo	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
profundidad	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

El nodo 5 está en la posición 2, el nodo 8 está en la posición 5, y el nodo con la profundidad mínima entre las posiciones  $2 \dots 5$  es el nodo 2 en la posición 3 cuya profundidad es 2. Por lo tanto, el antepasado común más bajo de los nodos 5 y 8 es el nodo 2.

Por lo tanto, para encontrar el antepasado común más bajo de dos nodos, basta con procesar una consulta de rango mínimo. Como el array es estático, podemos procesar tales consultas en tiempo  $O(1)$  después de un preprocesamiento de tiempo  $O(n \log n)$ .

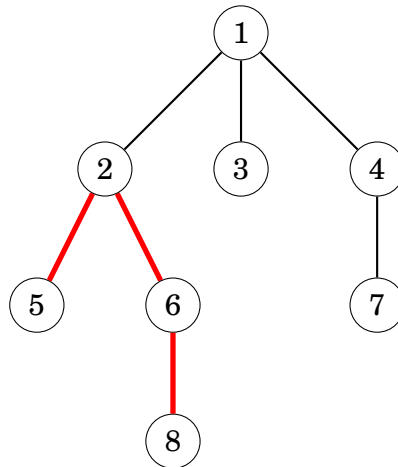
## Distancias de los nodos

La distancia entre los nodos  $a$  y  $b$  es igual a la longitud del camino de  $a$  a  $b$ . Resulta que el problema de calcular la distancia entre los nodos se reduce a encontrar su antepasado común más bajo.

Primero, enraízamos el árbol arbitrariamente. Después de esto, la distancia de los nodos  $a$  y  $b$  se puede calcular usando la fórmula

$$\text{profundidad}(a) + \text{profundidad}(b) - 2 \cdot \text{profundidad}(c),$$

donde  $c$  es el antepasado común más bajo de  $a$  y  $b$  y  $\text{profundidad}(s)$  denota la profundidad del nodo  $s$ . Por ejemplo, considere la distancia de los nodos 5 y 8:



El antepasado común más bajo de los nodos 5 y 8 es el nodo 2. Las profundidades de los nodos son  $\text{profundidad}(5) = 3$ ,  $\text{profundidad}(8) = 4$  y  $\text{profundidad}(2) = 2$ , por lo que la distancia entre los nodos 5 y 8 es  $3 + 4 - 2 \cdot 2 = 3$ .

## 18.4 Algoritmos fuera de línea

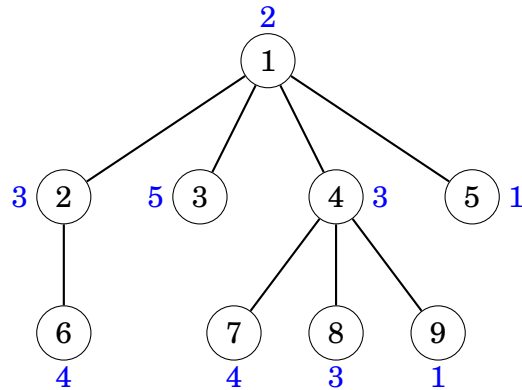
Hasta ahora, hemos discutido algoritmos *en línea* para consultas de árboles. Esos algoritmos son capaces de procesar consultas una tras otra de modo que cada consulta se responda antes de recibir la siguiente consulta.

Sin embargo, en muchos problemas, la propiedad en línea no es necesaria. En esta sección, nos centramos en algoritmos *fuera de línea*. Esos algoritmos reciben un conjunto de consultas que pueden ser respondidas en cualquier orden. A menudo es más fácil diseñar un algoritmo fuera de línea en comparación con un algoritmo en línea.

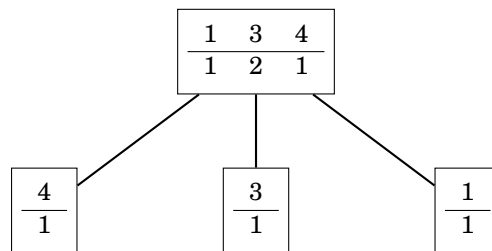
### Fusión de estructuras de datos

Un método para construir un algoritmo fuera de línea es realizar un recorrido en profundidad del árbol y mantener estructuras de datos en los nodos. En cada nodo  $s$ , creamos una estructura de datos  $d[s]$  que se basa en las estructuras de datos de los hijos de  $s$ . Entonces, usando esta estructura de datos, se procesan todas las consultas relacionadas con  $s$ .

Como ejemplo, considere el siguiente problema: Se nos da un árbol donde cada nodo tiene algún valor. Nuestra tarea es procesar consultas de la forma "calcular el número de nodos con valor  $x$  en el subárbol del nodo  $s$ ". Por ejemplo, en el siguiente árbol, el subárbol del nodo 4 contiene dos nodos cuyo valor es 3.



En este problema, podemos usar estructuras de mapa para responder a las consultas. Por ejemplo, los mapas para el nodo 4 y sus hijos son los siguientes:



Si creamos tal estructura de datos para cada nodo, podemos procesar fácilmente todas las consultas dadas, porque podemos manejar todas las consultas relacionadas con un nodo inmediatamente después de crear su estructura de datos. Por ejemplo, lo anterior estructura de mapa para el nodo 4 nos dice que su subárbol contiene dos nodos cuyo valor es 3.

Sin embargo, sería demasiado lento crear todas las estructuras de datos desde cero. En cambio, en cada nodo  $s$ , creamos una estructura de datos inicial  $d[s]$  que solo contiene el valor de  $s$ . Después de esto, revisamos los hijos de  $s$  y *fusionamos*  $d[s]$  y todas las estructuras de datos  $d[u]$  donde  $u$  es un hijo de  $s$ .

Por ejemplo, en el árbol anterior, el mapa para el nodo 4 se crea fusionando los siguientes mapas:



Aquí, el primer mapa es la estructura de datos inicial para el nodo 4, y los otros tres mapas corresponden a los nodos 7, 8 y 9.

La fusión en el nodo  $s$  se puede hacer de la siguiente manera: Recorremos los hijos de  $s$  y en cada hijo  $u$  fusionamos  $d[s]$  y  $d[u]$ . Siempre copiamos el contenido de  $d[u]$  a  $d[s]$ . Sin embargo, antes de esto, *intercambiamos* el contenido de  $d[s]$  y

$d[u]$  si  $d[s]$  es menor que  $d[u]$ . Al hacer esto, cada valor se copia solo  $O(\log n)$  veces durante el recorrido del árbol, lo que garantiza que el algoritmo sea eficiente.

Para intercambiar el contenido de dos estructuras de datos  $a$  y  $b$  de manera eficiente, solo podemos usar el siguiente código:

```
swap(a, b);
```

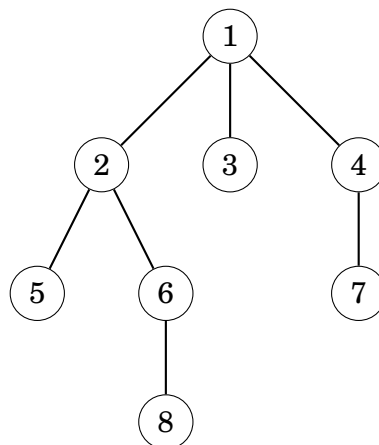
Está garantizado que el código anterior funciona en tiempo constante cuando  $a$  y  $b$  son estructuras de datos de la biblioteca estándar de C++.

## Antepasados comunes más bajos

También existe un algoritmo fuera de línea para procesar un conjunto de consultas de antepasados comunes más bajos<sup>2</sup>. El algoritmo se basa en la estructura de datos union-find (ver Capítulo 15.2), y el beneficio del algoritmo es que es más fácil de implementar que los algoritmos discutidos anteriormente en este capítulo. El algoritmo se le da como entrada un conjunto de pares de nodos, y determina para cada uno de estos pares el antepasado común más bajo de los nodos. El algoritmo realiza un recorrido en profundidad del árbol y mantiene conjuntos disjuntos de nodos. Inicialmente, cada nodo pertenece a un conjunto separado. Para cada conjunto, también almacenamos el nodo más alto en el árbol que pertenece al conjunto.

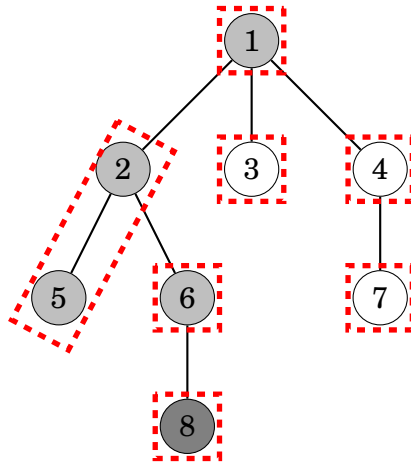
Cuando el algoritmo visita un nodo  $x$ , recorre todos los nodos  $y$  tales que el antepasado común más bajo de  $x$  e  $y$  tiene que ser encontrado. Si  $y$  ya ha sido visitado, el algoritmo informa que el antepasado común más bajo de  $x$  e  $y$  es el nodo más alto en el conjunto de  $y$ . Luego, después de procesar el nodo  $x$ , el algoritmo une los conjuntos de  $x$  y su padre.

Por ejemplo, suponga que queremos encontrar el más bajo antepasados comunes de pares de nodos (5, 8) y (2, 7) en el siguiente árbol:

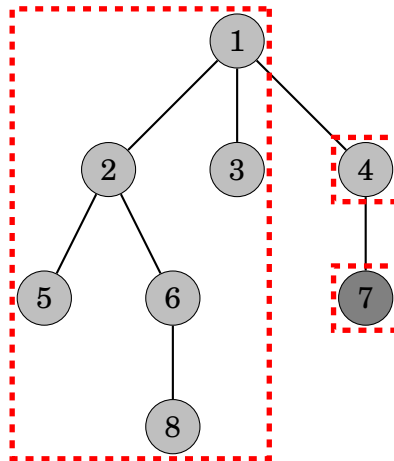


En los siguientes árboles, los nodos grises denotan nodos visitados y los grupos de nodos discontinuos pertenecen al mismo conjunto. Cuando el algoritmo visita el nodo 8, observa que el nodo 5 ha sido visitado y el nodo más alto en su conjunto es 2. Por lo tanto, el antepasado común más bajo de los nodos 5 y 8 es 2:

<sup>2</sup>Este algoritmo fue publicado por R. E. Tarjan en 1979 [65].



Más tarde, al visitar el nodo 7, el algoritmo determina que el antepasado común más bajo de los nodos 2 y 7 es 1:







# Chapter 19

## Camino y circuitos

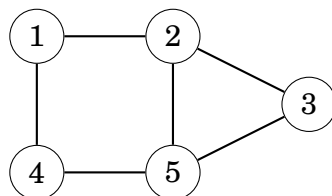
Este capítulo se centra en dos tipos de caminos en grafos:

- Un camino **Euleriano** es un camino que atraviesa cada arista exactamente una vez.
- Un camino **Hamiltoniano** es un camino que visita cada nodo exactamente una vez.

Si bien los caminos Eulerianos y Hamiltonianos parecen conceptos similares a primera vista, los problemas computacionales relacionados con ellos son muy diferentes. Resulta que hay una regla simple que determina si un grafo contiene un camino Euleriano, y también existe un algoritmo eficiente para encontrar dicho camino si existe. Por el contrario, verificar la existencia de un camino Hamiltoniano es un problema NP-difícil, y no se conoce ningún algoritmo eficiente para resolver el problema.

### 19.1 Caminos Eulerianos

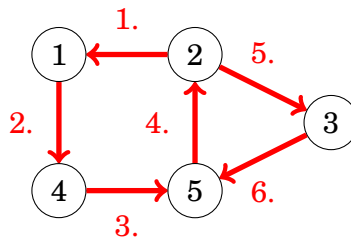
Un camino **Euleriano**<sup>1</sup> es un camino que atraviesa exactamente una vez cada arista del grafo. Por ejemplo, el grafo



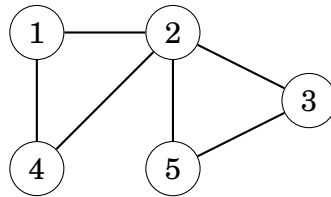
tiene un camino Euleriano desde el nodo 2 hasta el nodo 5:

---

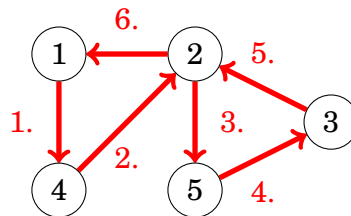
<sup>1</sup>L. Euler estudió este tipo de caminos en 1736 cuando resolvió el famoso problema de los puentes de Königsberg. Este fue el nacimiento de la teoría de grafos.



Un **circuito Euleriano** es un camino Euleriano que comienza y termina en el mismo nodo. Por ejemplo, el grafo



tiene un circuito Euleriano que comienza y termina en el nodo 1:

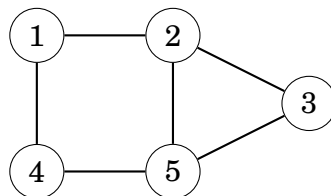


## Existencia

La existencia de caminos y circuitos Eulerianos depende de los grados de los nodos. Primero, un grafo no dirigido tiene un camino Euleriano exactamente cuando todas las aristas pertenecen al mismo componente conectado y

- el grado de cada nodo es par o
- el grado de exactamente dos nodos es impar, y el grado de todos los demás nodos es par.

En el primer caso, cada camino Euleriano es también un circuito Euleriano. En el segundo caso, los nodos de grado impar son los nodos de inicio y finalización de un camino Euleriano que no es un circuito Euleriano. Por ejemplo, en el gráfico



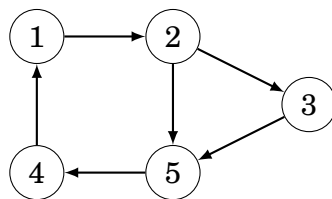
los nodos 1, 3 y 4 tienen un grado de 2, y los nodos 2 y 5 tienen un grado de 3. Exactamente dos nodos tienen un grado impar, por lo que hay un camino euleriano entre los nodos 2 y 5, pero el gráfico no contiene un circuito euleriano.

En un gráfico dirigido, nos centramos en los grados de entrada y salida de los nodos. Un gráfico dirigido contiene un camino euleriano exactamente cuando todos los bordes pertenecen al mismo componente conectado y

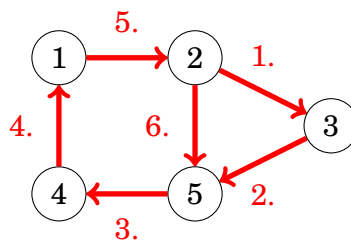
- en cada nodo, el grado de entrada es igual al grado de salida, o
- en un nodo, el grado de entrada es uno mayor que el grado de salida, en otro nodo, el grado de salida es uno mayor que el grado de entrada, y en todos los demás nodos, el grado de entrada es igual al grado de salida.

En el primer caso, cada camino euleriano también es un circuito euleriano, y en el segundo caso, el gráfico contiene un camino euleriano que comienza en el nodo cuyo grado de salida es mayor y termina en el nodo cuyo grado de entrada es mayor.

Por ejemplo, en el gráfico



los nodos 1, 3 y 4 tienen tanto grado de entrada 1 como grado de salida 1, el nodo 2 tiene grado de entrada 1 y grado de salida 2, y el nodo 5 tiene grado de entrada 2 y grado de salida 1. Por lo tanto, el gráfico contiene un camino euleriano desde el nodo 2 hasta el nodo 5:



## Algoritmo de Hierholzer

**Algoritmo de Hierholzer**<sup>2</sup> es un método eficiente para construir un circuito euleriano. El algoritmo consiste en varias rondas, cada una de las cuales agrega nuevos bordes al circuito. Por supuesto, asumimos que el gráfico contiene un circuito euleriano; de lo contrario, el algoritmo de Hierholzer no puede encontrarlo.

Primero, el algoritmo construye un circuito que contiene algunos (no necesariamente todos) de los bordes del gráfico. Después de esto, el algoritmo extiende el circuito paso a paso agregando subcircuitos a él. El proceso continúa hasta que se han agregado todos los bordes al circuito.

El algoritmo extiende el circuito siempre buscando un nodo  $x$  que pertenece al circuito pero tiene un borde saliente que no está incluido en el circuito. El

<sup>2</sup>El algoritmo fue publicado en 1873 después de la muerte de Hierholzer [35].

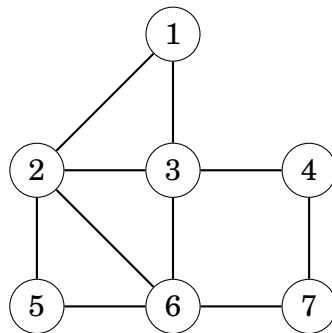
algoritmo construye un nuevo camino desde el nodo  $x$  que solo contiene bordes que aún no están en el circuito. Tarde o temprano, el camino volverá al nodo  $x$ , lo que crea un subcircuito.

Si el gráfico solo contiene un camino euleriano, aún podemos usar el algoritmo de Hierholzer para encontrarlo agregando un borde extra al gráfico y eliminando el borde después de que el circuito se haya construido. Por ejemplo, en un gráfico no dirigido, agregamos el borde extra entre los dos nodos de grado impar.

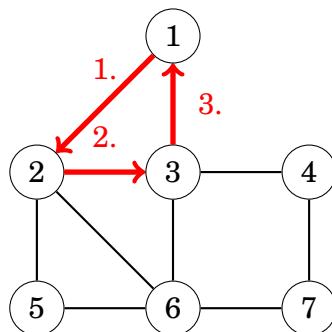
A continuación, veremos cómo el algoritmo de Hierholzer construye un circuito euleriano para un gráfico no dirigido.

## Ejemplo

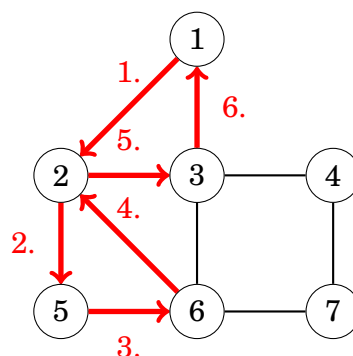
Consideremos el siguiente gráfico:



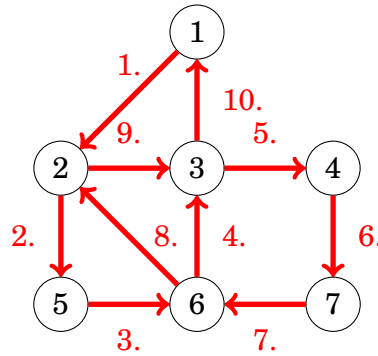
Supongamos que el algoritmo primero crea un circuito que comienza en el nodo 1. Un posible circuito es  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ :



Después de esto, el algoritmo agrega el subcircuito  $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$  al circuito:



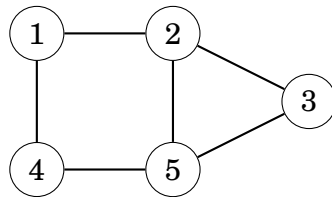
Finalmente, el algoritmo agrega el subcircuito  $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$  al circuito:



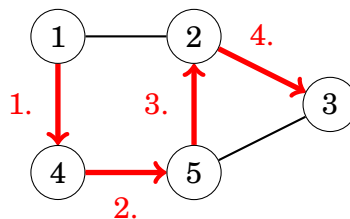
Ahora todos los bordes están incluidos en el circuito, por lo que hemos construido con éxito un circuito euleriano.

## 19.2 Caminos hamiltonianos

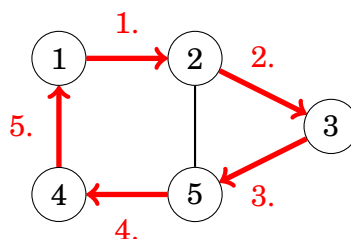
Un **camino hamiltoniano** es un camino que visita cada nodo del grafo exactamente una vez. Por ejemplo, el grafo



contiene un camino hamiltoniano desde el nodo 1 hasta el nodo 3:



Si un camino hamiltoniano comienza y termina en el mismo nodo, se llama **circuito hamiltoniano**. El gráfico anterior también tiene un circuito hamiltoniano que comienza y termina en el nodo 1:



## Existencia

No se conoce ningún método eficiente para probar si un gráfico contiene un camino hamiltoniano, y el problema es NP-difícil. Aún así, en algunos casos especiales, podemos estar seguros de que un gráfico contiene un camino hamiltoniano.

Una simple observación es que si el gráfico es completo, es decir, existe una arista entre todos los pares de nodos, también contiene un camino hamiltoniano. También se han logrado resultados más fuertes:

- **Teorema de Dirac:** Si el grado de cada nodo es al menos  $n/2$ , el gráfico contiene un camino hamiltoniano.
- **Teorema de Ore:** Si la suma de grados de cada par de nodos no adyacentes es al menos  $n$ , el gráfico contiene un camino hamiltoniano.

Una propiedad común en estos teoremas y otros resultados es que garantizan la existencia de un camino hamiltoniano si el gráfico tiene *un gran número* de aristas. Esto tiene sentido, porque cuantas más aristas contiene el gráfico, más posibilidades hay de construir un camino hamiltoniano.

## Construcción

Dado que no existe una forma eficiente de comprobar si existe un camino hamiltoniano, es claro que tampoco existe un método para construir el camino de forma eficiente, porque de lo contrario, podríamos simplemente intentar construir el camino y ver si existe.

Una forma simple de buscar un camino hamiltoniano es utilizar un algoritmo de retroceso que recorre todas las formas posibles de construir el camino. La complejidad temporal de dicho algoritmo es al menos  $O(n!)$ , porque hay  $n!$  formas diferentes de elegir el orden de  $n$  nodos.

Una solución más eficiente se basa en la programación dinámica (ver Capítulo 10.5). La idea es calcular los valores de una función  $\text{posible}(S, x)$ , donde  $S$  es un subconjunto de nodos y  $x$  es uno de los nodos. La función indica si existe un camino hamiltoniano que visita los nodos de  $S$  y termina en el nodo  $x$ . Es posible implementar esta solución en  $O(2^n n^2)$  tiempo.

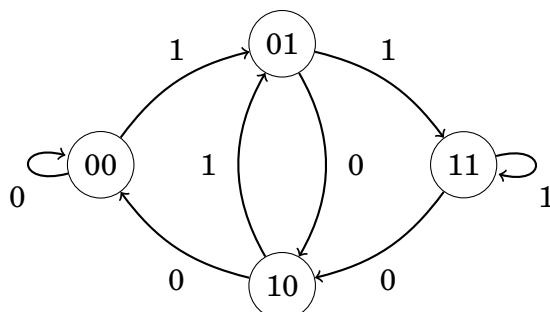
## 19.3 Secuencias de De Bruijn

Una **Secuencia de De Bruijn** es una cadena que contiene cada cadena de longitud  $n$  exactamente una vez como subcadena, para un alfabeto fijo de  $k$  caracteres. La longitud de dicha cadena es  $k^n + n - 1$  caracteres. Por ejemplo, cuando  $n = 3$  y  $k = 2$ , un ejemplo de una secuencia de De Bruijn es

0001011100.

Las subcadenas de esta cadena son todas combinaciones de tres bits: 000, 001, 010, 011, 100, 101, 110 y 111.

Resulta que cada secuencia de De Bruijn corresponde a un camino euleriano en un grafo. La idea es construir un grafo donde cada nodo contiene una cadena de  $n - 1$  caracteres y cada arista agrega un carácter a la cadena. El siguiente grafo corresponde al escenario anterior:



Un camino euleriano en este grafo corresponde a una cadena que contiene todas las cadenas de longitud  $n$ . La cadena contiene los caracteres del nodo inicial y todos los caracteres de las aristas. El nodo inicial tiene  $n - 1$  caracteres y hay  $k^n$  caracteres en las aristas, por lo que la longitud de la cadena es  $k^n + n - 1$ .

## 19.4 Recorridos del caballo

Un **recorrido del caballo** es una secuencia de movimientos de un caballo en un tablero de ajedrez  $n \times n$  siguiendo las reglas del ajedrez de modo que el caballo visite cada casilla exactamente una vez. Un recorrido del caballo se llama recorrido *cerrado* si el caballo finalmente regresa a la casilla inicial y de lo contrario se llama recorrido *abierto*.

Por ejemplo, aquí hay un recorrido abierto del caballo en un tablero de  $5 \times 5$ :

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Un recorrido del caballo corresponde a un camino hamiltoniano en un grafo cuyos nodos representan las casillas del tablero, y dos nodos están conectados con una arista si un caballo puede moverse entre las casillas de acuerdo con las reglas del ajedrez.

Una forma natural de construir un recorrido del caballo es usar el retroceso. La búsqueda se puede hacer más eficiente utilizando *heurísticas* que intentan guiar al caballo para que se encuentre rápidamente un recorrido completo.

## Regla de Warnsdorf

**Regla de Warnsdorf** es una heurística simple y efectiva para encontrar un recorrido del caballo<sup>3</sup>. Usando la regla, es posible construir eficientemente un recorrido incluso en un tablero grande. La idea es siempre mover el caballo de modo que termine en un cuadrado donde el número de movimientos posibles es *lo más pequeño posible*.

Por ejemplo, en la siguiente situación, hay cinco cuadrados posibles a los que el caballo puede moverse (cuadrados  $a \dots e$ ):

1				$a$
		2		
$b$				$e$
	$c$		$d$	

En esta situación, la regla de Warnsdorf mueve el caballo al cuadrado  $a$ , porque después de esta elección, solo hay un movimiento posible. Las otras opciones moverían el caballo a cuadrados donde habría tres movimientos disponibles.

---

<sup>3</sup>Esta heurística fue propuesta en el libro de Warnsdorf [69] en 1823. También hay algoritmos polinomiales para encontrar recorridos del caballo [52], pero son más complicados.



# Chapter 20

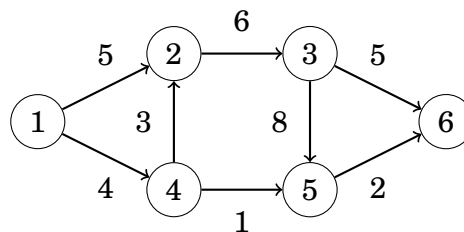
## Flujos y cortes

En este capítulo, nos centraremos en los siguientes dos problemas:

- **Encontrar un flujo máximo:** ¿Cuál es la cantidad máxima de flujo que podemos enviar de un nodo a otro nodo?
- **Encontrar un corte mínimo:** ¿Cuál es un conjunto de aristas de peso mínimo que separa dos nodos del grafo?

La entrada para ambos problemas es un grafo dirigido, con pesos que contiene dos nodos especiales: la *fente* es un nodo sin aristas entrantes, y el *sumidero* es un nodo sin aristas salientes.

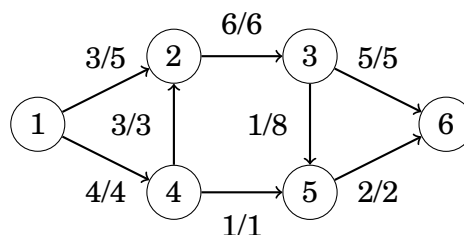
Como ejemplo, usaremos el siguiente grafo donde el nodo 1 es la fuente y el nodo 6 es el sumidero:



### Flujo máximo

En el problema del **flujo máximo**, nuestra tarea es enviar la mayor cantidad de flujo posible de la fuente al sumidero. El peso de cada arista es una capacidad que restringe el flujo que puede pasar por la arista. En cada nodo intermedio, el flujo entrante y saliente tiene que ser igual.

Por ejemplo, el tamaño máximo de un flujo en el grafo de ejemplo es 7. La siguiente imagen muestra cómo podemos enrutar el flujo:

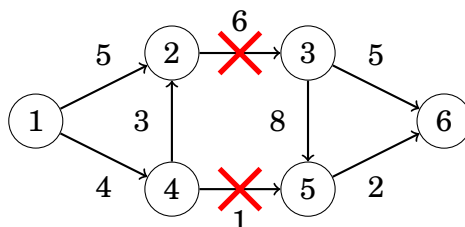


La notación  $v/k$  significa que se enruta un flujo de  $v$  unidades a través de una arista cuya capacidad es de  $k$  unidades. El tamaño del flujo es 7, porque la fuente envía  $3 + 4$  unidades de flujo y el sumidero recibe  $5 + 2$  unidades de flujo. Es fácil ver que este flujo es máximo, porque la capacidad total de las aristas que conducen al sumidero es 7.

## Corte mínimo

En el problema del **corte mínimo**, nuestra tarea es eliminar un conjunto de aristas del grafo de modo que no haya camino desde la fuente al sumidero después de la eliminación y el peso total de las aristas eliminadas sea mínimo.

El tamaño mínimo de un corte en el grafo de ejemplo es 7. Basta con eliminar las aristas  $2 \rightarrow 3$  y  $4 \rightarrow 5$ :



Después de eliminar los bordes, no habrá ningún camino desde la fuente hasta el sumidero. El tamaño del corte es 7, porque los pesos de los bordes eliminados son 6 y 1. El corte es mínimo, porque no hay una forma válida de eliminar bordes del gráfico de modo que su peso total sea menor que 7.

No es una coincidencia que el tamaño máximo de un flujo y el tamaño mínimo de un corte sean los mismos en el ejemplo anterior. Resulta que un flujo máximo y un corte mínimo son *siempre* igualmente grandes, por lo que los conceptos son dos caras de la misma moneda.

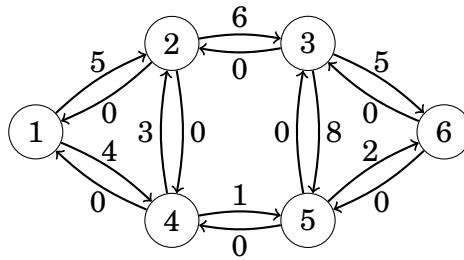
A continuación, discutiremos el algoritmo de Ford–Fulkerson que se puede utilizar para encontrar el flujo máximo y el corte mínimo de un gráfico. El algoritmo también nos ayuda a entender *por qué* son igualmente grandes.

## 20.1 Algoritmo de Ford–Fulkerson

El **algoritmo de Ford–Fulkerson** [25] encuentra el flujo máximo en un gráfico. El algoritmo comienza con un flujo vacío, y en cada paso encuentra un camino desde la fuente hasta el sumidero que genera más flujo. Finalmente, cuando el algoritmo no puede aumentar el flujo más, se ha encontrado el flujo máximo.

El algoritmo utiliza una representación especial del gráfico donde cada borde original tiene un inverso borde en otra dirección. El peso de cada borde indica cuánta más flujo podríamos enrutar a través de él. Al comienzo del algoritmo, el peso de cada borde original es igual a la capacidad del borde y el peso de cada borde inverso es cero.

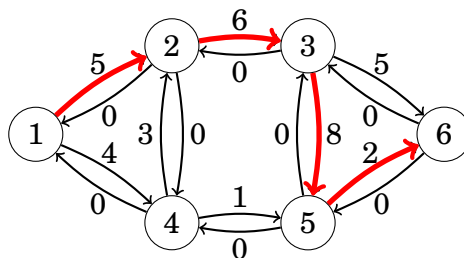
La nueva representación para el gráfico de ejemplo es la siguiente:



## Descripción del algoritmo

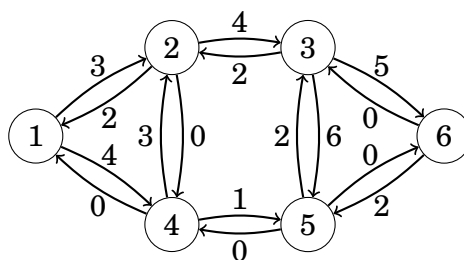
El algoritmo de Ford–Fulkerson consta de varias rondas. En cada ronda, el algoritmo encuentra un camino desde la fuente hasta el sumidero tal que cada borde en el camino tiene un peso positivo. Si hay más de un camino posible disponible, podemos elegir cualquiera de ellos.

Por ejemplo, supongamos que elegimos el siguiente camino:



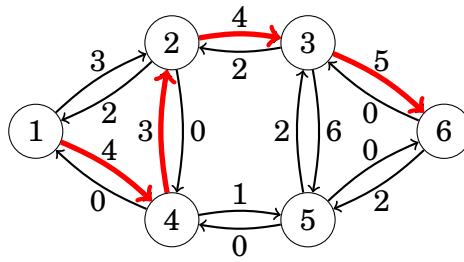
Después de elegir la ruta, el flujo aumenta en  $x$  unidades, donde  $x$  es el peso de borde más pequeño en la ruta. Además, el peso de cada borde en la ruta disminuye en  $x$  y el peso de cada borde inverso aumenta en  $x$ .

En la ruta anterior, los pesos de los bordes son 5, 6, 8 y 2. El peso más pequeño es 2, por lo que el flujo aumenta en 2 y el nuevo gráfico es el siguiente:



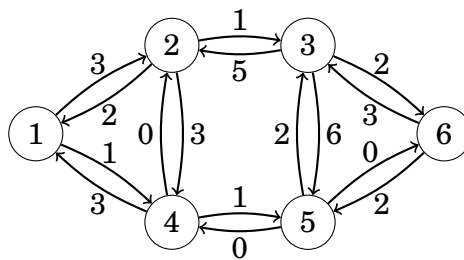
La idea es que aumentar el flujo disminuye la cantidad de flujo que puede pasar a través de los bordes en el futuro. Por otro lado, es posible cancelar el flujo más tarde usando los bordes inversos del gráfico si resulta que sería beneficioso enrutar el flujo de otra manera.

El algoritmo aumenta el flujo mientras exista una ruta desde la fuente hasta el sumidero a través de bordes de peso positivo. En el ejemplo actual, nuestra siguiente ruta puede ser la siguiente:

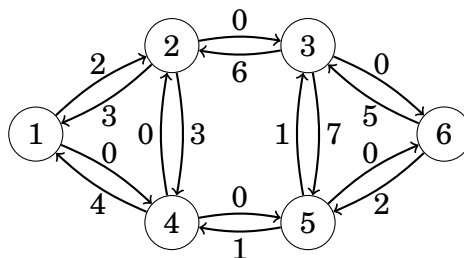


El peso mínimo de la arista en este camino es 3, por lo que el camino aumenta el flujo en 3, y el flujo total después de procesar el camino es 5.

El nuevo gráfico será como sigue:



Todavía necesitamos dos rondas más antes de alcanzar el flujo máximo. Por ejemplo, podemos elegir los caminos  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$  y  $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ . Ambos caminos aumentan el flujo en 1, y el gráfico final es como sigue:



No es posible aumentar el flujo más, porque no hay camino desde la fuente al sumidero con pesos de borde positivos. Por lo tanto, el algoritmo termina y el flujo máximo es 7.

## Encontrar caminos

El algoritmo de Ford–Fulkerson no especifica cómo debemos elegir los caminos que aumentan el flujo. En cualquier caso, el algoritmo terminará tarde o temprano y encontrará correctamente el flujo máximo. Sin embargo, la eficiencia del algoritmo depende de la forma en que se eligen los caminos.

Una forma sencilla de encontrar caminos es usar la búsqueda en profundidad. Por lo general, esto funciona bien, pero en el peor de los casos, cada camino solo aumenta el flujo en 1 y el algoritmo es lento. Afortunadamente, podemos evitar esta situación utilizando una de las siguientes técnicas:

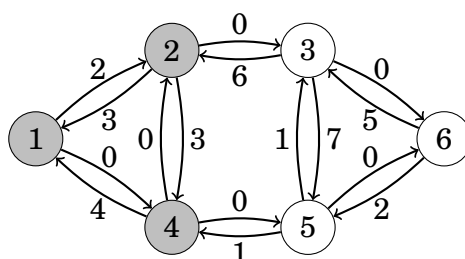
El **algoritmo de Edmonds–Karp** [18] elige cada camino de modo que el número de aristas en el camino sea lo más pequeño posible. Esto se puede hacer utilizando la búsqueda en amplitud en lugar de la búsqueda en profundidad para encontrar caminos. Se puede probar que esto garantiza que el flujo aumente rápidamente, y la complejidad temporal del algoritmo es  $O(m^2n)$ .

El **algoritmo de escalado** [2] utiliza la búsqueda en profundidad para encontrar caminos donde cada peso de arista es al menos un valor umbral. Inicialmente, el valor umbral es algún número grande, por ejemplo, la suma de todos los pesos de arista del gráfico. Siempre que no se pueda encontrar un camino, el valor umbral se divide por 2. La complejidad temporal del algoritmo es  $O(m^2 \log c)$ , donde  $c$  es el valor umbral inicial.

En la práctica, el algoritmo de escalado es más fácil de implementar, porque la búsqueda en profundidad se puede utilizar para encontrar caminos. Ambos algoritmos son lo suficientemente eficientes para problemas que normalmente aparecen en las competiciones de programación.

## Cortes mínimos

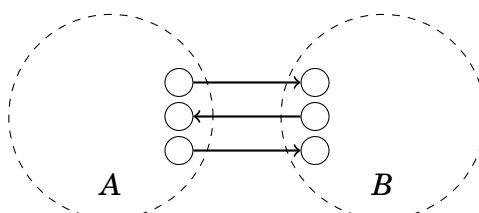
Resulta que una vez que el algoritmo de Ford–Fulkerson ha encontrado un flujo máximo, también ha determinado un corte mínimo. Sea  $A$  el conjunto de nodos que se pueden alcanzar desde la fuente usando aristas de peso positivo. En el gráfico de ejemplo,  $A$  contiene los nodos 1, 2 y 4:



Ahora, el corte mínimo consta de los bordes del gráfico original que comienzan en algún nodo en  $A$ , terminan en algún nodo fuera de  $A$ , y cuya capacidad se utiliza completamente en el flujo máximo. En el gráfico anterior, estos bordes son  $2 \rightarrow 3$  y  $4 \rightarrow 5$ , que corresponden al corte mínimo  $6 + 1 = 7$ .

¿Por qué el flujo producido por el algoritmo es máximo y por qué es el corte mínimo? La razón es que un gráfico no puede contener un flujo cuyo tamaño sea mayor que el peso de cualquier corte del gráfico. Por lo tanto, siempre que un flujo y un corte son igualmente grandes, son un flujo máximo y un corte mínimo.

Consideremos cualquier corte del gráfico tal que la fuente pertenezca a  $A$ , el sumidero pertenezca a  $B$  y haya algunos bordes entre los conjuntos:



El tamaño del corte es la suma de los bordes que van de  $A$  a  $B$ . Este es un límite superior para el flujo en el gráfico, porque el flujo tiene que proceder de  $A$  a  $B$ . Por lo tanto, el tamaño de un flujo máximo es menor o igual que el tamaño de cualquier corte en el gráfico.

Por otro lado, el algoritmo de Ford–Fulkerson produce un flujo cuyo tamaño es *exactamente* tan grande como el tamaño de un corte en el gráfico. Por lo tanto, el flujo tiene que ser un flujo máximo y el corte tiene que ser un corte mínimo.

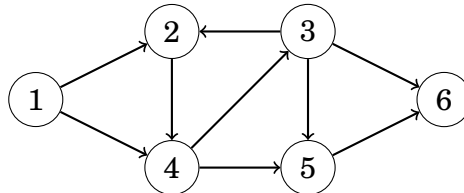
## 20.2 Caminos disjuntos

Muchos problemas de gráficos se pueden resolver reduciéndolos al problema de flujo máximo. Nuestro primer ejemplo de tal problema es el siguiente: se nos da un gráfico dirigido con una fuente y un sumidero, y nuestra tarea es encontrar el número máximo de caminos disjuntos desde la fuente hasta el sumidero.

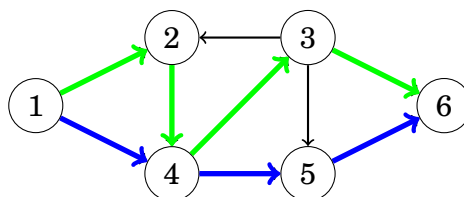
### Caminos disjuntos de borde

Primero nos centraremos en el problema de encontrar el número máximo de **caminos disjuntos de borde** desde la fuente hasta el sumidero. Esto significa que debemos construir un conjunto de caminos de modo que cada borde aparezca en como máximo un camino.

Por ejemplo, considere el siguiente gráfico:



En este gráfico, el número máximo de caminos disjuntos de borde es 2. Podemos elegir los caminos  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$  y  $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$  de la siguiente manera:



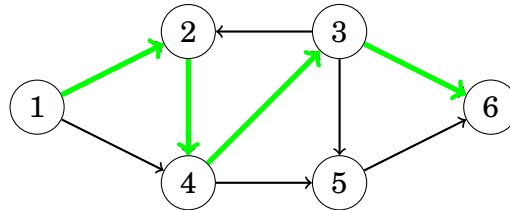
Resulta que el número máximo de caminos disjuntos de aristas es igual al flujo máximo del grafo, asumiendo que la capacidad de cada arista es uno. Después de que se ha construido el flujo máximo, los caminos disjuntos de aristas se pueden encontrar codiciosamente siguiendo caminos desde la fuente hasta el sumidero.

### Caminos disjuntos de nodos

Consideremos ahora otro problema: encontrar el número máximo de **caminos disjuntos de nodos** desde la fuente hasta el sumidero. En este problema, cada

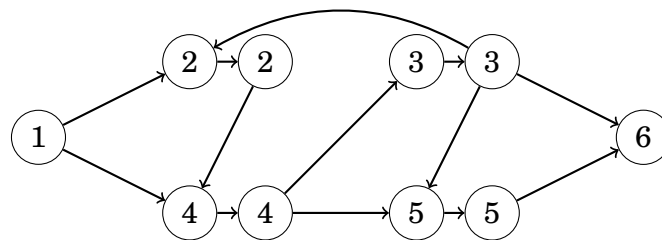
nodo, excepto la fuente y el sumidero, puede aparecer en como máximo un camino. El número de caminos disjuntos de nodos puede ser menor que el número de caminos disjuntos de aristas.

Por ejemplo, en el gráfico anterior, el número máximo de caminos disjuntos de nodos es 1:

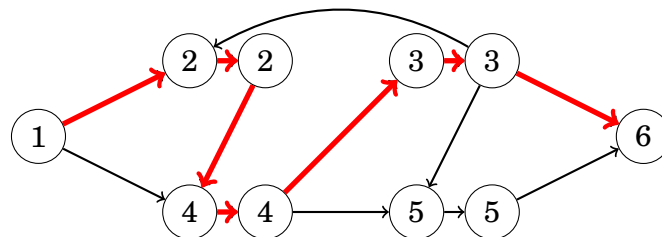


También podemos reducir este problema al problema de flujo máximo. Dado que cada nodo puede aparecer en como máximo un camino, tenemos que limitar el flujo que pasa por los nodos. Un método estándar para esto es dividir cada nodo en dos nodos de modo que el primer nodo tenga las aristas entrantes del nodo original, el segundo nodo tenga las aristas salientes del nodo original, y haya una nueva arista desde el primer nodo hasta el segundo nodo.

En nuestro ejemplo, el gráfico se convierte en el siguiente:



El flujo máximo para el gráfico es el siguiente:



Por lo tanto, el número máximo de caminos nodo-disjuntos desde la fuente hasta el sumidero es 1.

## 20.3 Emparejamientos máximos

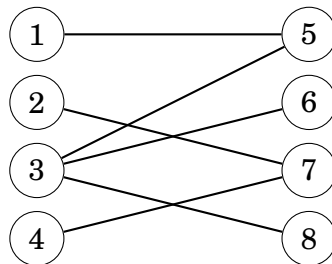
El problema del **emparejamiento máximo** pide encontrar un conjunto de tamaño máximo de pares de nodos en un grafo no dirigido tal que cada par esté conectado con una arista y cada nodo pertenezca a como máximo un par.

Existen algoritmos polinomiales para encontrar emparejamientos máximos en grafos generales [17], pero tales algoritmos son complejos y rara vez se ven

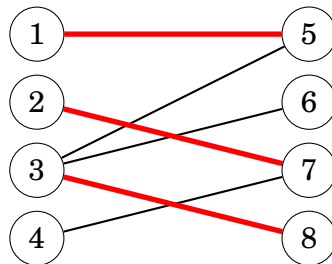
en concursos de programación. Sin embargo, en grafos bipartitos, el problema del emparejamiento máximo es mucho más fácil de resolver, porque podemos reducirlo al problema de flujo máximo.

## Encontrar emparejamientos máximos

Los nodos de un grafo bipartito siempre se pueden dividir en dos grupos de modo que todas las aristas del grafo vayan del grupo izquierdo al grupo derecho. Por ejemplo, en el siguiente grafo bipartito, los grupos son  $\{1, 2, 3, 4\}$  y  $\{5, 6, 7, 8\}$ .

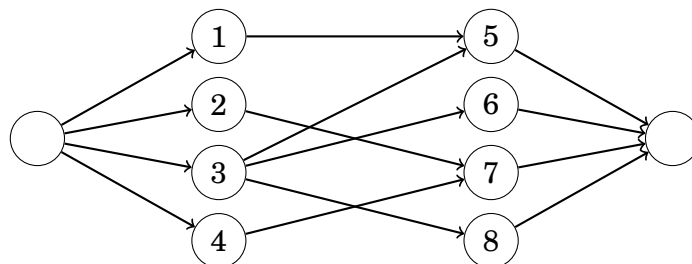


El tamaño de un emparejamiento máximo de este grafo es 3:



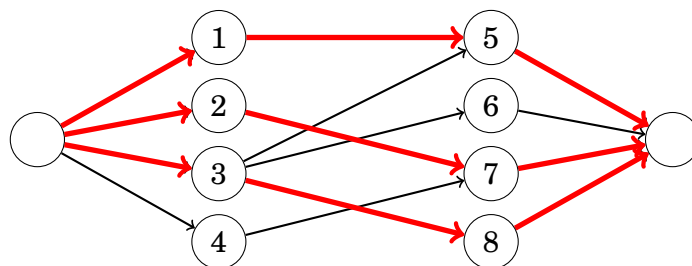
Podemos reducir el problema del emparejamiento máximo bipartito al problema del flujo máximo agregando dos nuevos nodos al grafo: una fuente y un sumidero. También agregamos aristas desde la fuente a cada nodo izquierdo y desde cada nodo derecho hasta el sumidero. Después de esto, el tamaño de un flujo máximo en el grafo es igual al tamaño de un emparejamiento máximo en el grafo original.

Por ejemplo, la reducción para el grafo anterior es como sigue:



El flujo máximo de este grafo es como sigue:



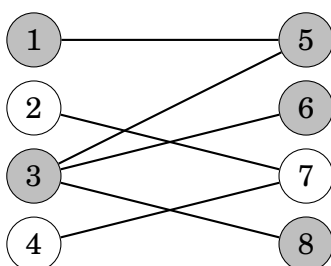


## Teorema de Hall

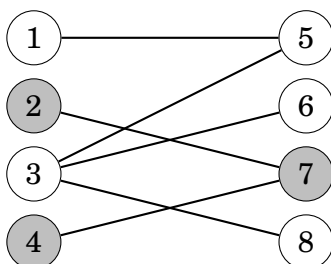
La **Teorema de Hall** se puede utilizar para averiguar si un grafo bipartito tiene un emparejamiento que contenga todos los nodos izquierdos o derechos. Si el número de nodos izquierdos y derechos es el mismo, el teorema de Hall nos dice si es posible construir un **emparejamiento perfecto** que contenga todos los nodos del grafo.

Supongamos que queremos encontrar un emparejamiento que contenga todos los nodos izquierdos. Sea  $X$  cualquier conjunto de nodos izquierdos y sea  $f(X)$  el conjunto de sus vecinos. De acuerdo con el teorema de Hall, un emparejamiento que contenga todos los nodos izquierdos existe exactamente cuando para cada  $X$ , la condición  $|X| \leq |f(X)|$  se cumple.

Estudiemos el teorema de Hall en el grafo de ejemplo. Primero, sea  $X = \{1, 3\}$  lo que produce  $f(X) = \{5, 6, 8\}$ :



La condición del teorema de Hall se cumple, porque  $|X| = 2$  y  $|f(X)| = 3$ . Luego, sea  $X = \{2, 4\}$  lo que produce  $f(X) = \{7\}$ :



En este caso,  $|X| = 2$  y  $|f(X)| = 1$ , por lo que la condición del teorema de Hall no se cumple. Esto significa que no es posible formar un emparejamiento perfecto para el grafo. Este resultado no es sorprendente, porque ya sabemos que el emparejamiento máximo del grafo es 3 y no 4.

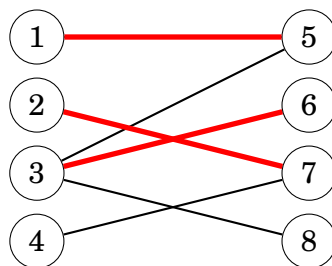
Si la condición del teorema de Hall no se cumple, el conjunto  $X$  proporciona una explicación *por qué* no podemos formar tal emparejamiento. Como  $X$  contiene

más nodos que  $f(X)$ , no hay pares para todos los nodos en  $X$ . Por ejemplo, en el grafo anterior, ambos nodos 2 y 4 deberían estar conectados con el nodo 7 lo cual no es posible.

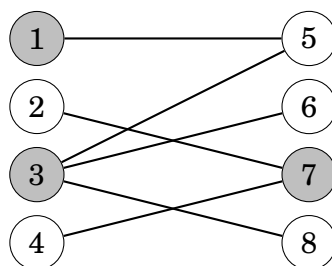
## Teorema de König

Una **cubierta de nodos mínima** de un grafo es un conjunto mínimo de nodos tal que cada arista del grafo tiene al menos un extremo en el conjunto. En un grafo general, encontrar una cubierta de nodos mínima es un problema NP-duro. Sin embargo, si el grafo es bipartito, el **Teorema de König** nos dice que el tamaño de una cubierta de nodos mínima y el tamaño de un emparejamiento máximo son siempre iguales. Por lo tanto, podemos calcular el tamaño de una cubierta de nodos mínima usando un algoritmo de flujo máximo.

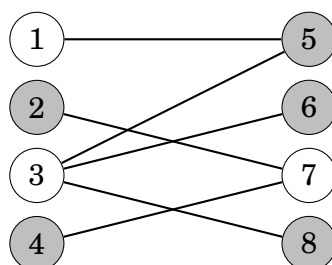
Consideremos el siguiente grafo con un emparejamiento máximo de tamaño 3:



Ahora el teorema de König nos dice que el tamaño de una cubierta de nodos mínima también es 3. Tal cubierta se puede construir de la siguiente manera:



Los nodos que *no* pertenecen a una cobertura de nodos mínima forman un conjunto **independiente máximo**. Este es el conjunto más grande posible de nodos tal que no hay dos nodos en el conjunto conectados con una arista. Una vez más, encontrar un conjunto independiente máximo en un gráfico general es un problema NP-difícil, pero en un gráfico bipartito podemos usar el teorema de König para resolver el problema de manera eficiente. En el gráfico de ejemplo, el conjunto independiente máximo es el siguiente:

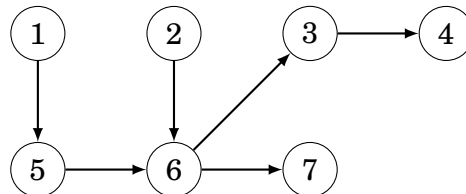


## 20.4 Coberturas de caminos

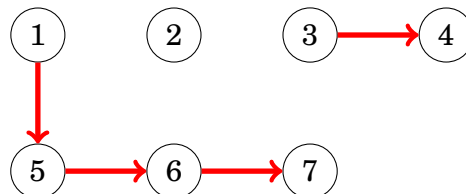
Una **cobertura de caminos** es un conjunto de caminos en un gráfico tal que cada nodo del gráfico pertenece a al menos un camino. Resulta que en los gráficos dirigidos y acíclicos, podemos reducir el problema de encontrar una cobertura de caminos mínima al problema de encontrar un flujo máximo en otro gráfico.

### Cobertura de caminos nodos-disjuntos

En una **cobertura de caminos nodos-disjuntos**, cada nodo pertenece a exactamente un camino. Como ejemplo, considera el siguiente gráfico:



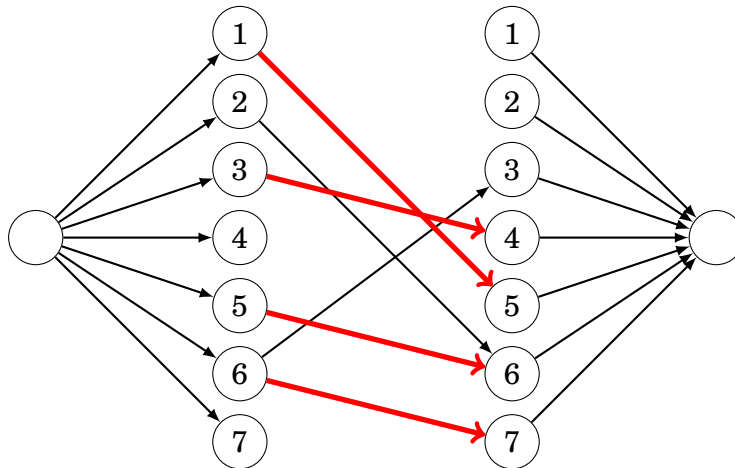
Una cobertura de caminos nodos-disjuntos mínima de este gráfico consta de tres caminos. Por ejemplo, podemos elegir los siguientes caminos:



Tenga en cuenta que uno de los caminos solo contiene el nodo 2, por lo que es posible que un camino no contenga ninguna arista.

Podemos encontrar una cobertura de caminos nodos-disjuntos mínima construyendo un *gráfico de coincidencia* donde cada nodo del gráfico original está representado por dos nodos: un nodo izquierdo y un nodo derecho. Hay una arista de un nodo izquierdo a un nodo derecho si hay una arista en el gráfico original. Además, el gráfico de coincidencia contiene una fuente y un sumidero, y hay aristas de la fuente a todos los nodos izquierdos y de todos los nodos derechos al sumidero.

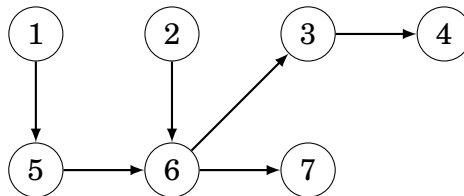
Una coincidencia máxima en el gráfico resultante corresponde a una cobertura de caminos nodos-disjuntos mínima en el gráfico original. Por ejemplo, el siguiente gráfico de coincidencia para el gráfico anterior contiene una coincidencia máxima de tamaño 4:



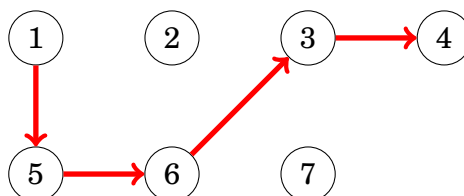
Cada arista en la coincidencia máxima del gráfico de coincidencia corresponde a una arista en la cubierta mínima de camino sin nodos coincidentes del gráfico original. Por lo tanto, el tamaño de la cubierta mínima de camino sin nodos coincidentes es  $n - c$ , donde  $n$  es el número de nodos en el gráfico original y  $c$  es el tamaño de la coincidencia máxima.

## Cubierta general de caminos

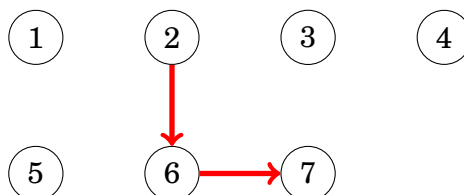
Una **cubierta general de caminos** es una cubierta de caminos donde un nodo puede pertenecer a más de un camino. Una cubierta general de caminos mínima puede ser más pequeña que una cubierta mínima de camino sin nodos coincidentes, porque un nodo puede utilizarse varias veces en caminos. Considere nuevamente el siguiente gráfico:



La cubierta general de caminos mínima de este gráfico consiste en dos caminos. Por ejemplo, el primer camino puede ser el siguiente:

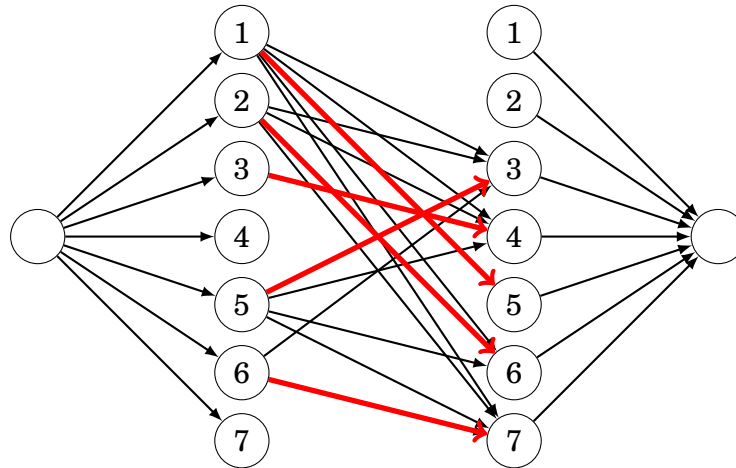


Y el segundo camino puede ser el siguiente:



Una cubierta general de caminos mínima puede encontrarse casi como una cubierta mínima de camino sin nodos coincidentes. Basta con agregar algunas aristas nuevas al gráfico de coincidencia para que haya una arista  $a \rightarrow b$  siempre cuando haya un camino de  $a$  a  $b$  en el gráfico original (posiblemente a través de varias aristas).

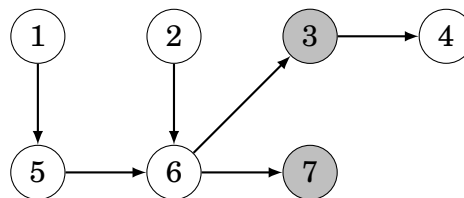
El gráfico de coincidencia para el gráfico anterior es el siguiente:



## Teorema de Dilworth

Una **cadena** es un conjunto de nodos de un grafo tal que no hay camino desde ningún nodo a otro nodo usando las aristas del grafo. El **Teorema de Dilworth** establece que en un grafo acíclico dirigido, el tamaño de una cubierta de camino general mínima es igual al tamaño de una cadena máxima.

Por ejemplo, los nodos 3 y 7 forman una cadena en el siguiente grafo:



Esta es una cadena máxima, porque no es posible construir ninguna cadena que contenga tres nodos. Hemos visto antes que el tamaño de una cubierta de camino general mínima de este grafo consiste en dos caminos.



# **Part III**

## **Temas avanzados**





# Chapter 21

## Teoría de números

**Teoría de números** es una rama de las matemáticas que estudia los enteros. La teoría de números es un campo fascinante, porque muchas preguntas que involucran enteros son muy difíciles de resolver incluso si parecen simples a primera vista.

Como ejemplo, considere la siguiente ecuación:

$$x^3 + y^3 + z^3 = 33$$

Es fácil encontrar tres números reales  $x$ ,  $y$  y  $z$  que satisfagan la ecuación. Por ejemplo, podemos elegir

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

Sin embargo, es un problema abierto en la teoría de números si hay tres *enteros*  $x$ ,  $y$  y  $z$  que satisfagan la ecuación [6].

En este capítulo, nos centraremos en conceptos básicos y algoritmos en teoría de números. A lo largo del capítulo, asumiremos que todos los números son enteros, a menos que se indique lo contrario.

### 21.1 Primos y factores

Un número  $a$  se llama **factor** o **divisor** de un número  $b$  si  $a$  divide  $b$ . Si  $a$  es un factor de  $b$ , escribimos  $a \mid b$ , y de lo contrario escribimos  $a \nmid b$ . Por ejemplo, los factores de 24 son 1, 2, 3, 4, 6, 8, 12 y 24.

Un número  $n > 1$  es un **primo** si sus únicos factores positivos son 1 y  $n$ . Por ejemplo, 7, 19 y 41 son primos, pero 35 no es un primo, porque  $5 \cdot 7 = 35$ . Para cada número  $n > 1$ , existe una única **factorización prima**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

donde  $p_1, p_2, \dots, p_k$  son primos distintos y  $\alpha_1, \alpha_2, \dots, \alpha_k$  son números positivos. Por ejemplo, la factorización prima de 84 es

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

El **número de factores** de un número  $n$  es

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

porque para cada primo  $p_i$ , hay  $\alpha_i + 1$  formas de elegir cuántas veces aparece en el factor. Por ejemplo, el número de factores de 84 es  $\tau(84) = 3 \cdot 2 \cdot 2 = 12$ . Los factores son 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 y 84.

El **suma de factores** de  $n$  es

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

donde la última fórmula se basa en la fórmula de progresión geométrica. Por ejemplo, la suma de factores de 84 es

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

El **producto de factores** de  $n$  es

$$\mu(n) = n^{\tau(n)/2},$$

porque podemos formar  $\tau(n)/2$  pares de los factores, cada uno con producto  $n$ . Por ejemplo, los factores de 84 producen los pares  $1 \cdot 84$ ,  $2 \cdot 42$ ,  $3 \cdot 28$ , etc., y el producto de los factores es  $\mu(84) = 84^6 = 351298031616$ .

Un número  $n$  se llama **número perfecto** si  $n = \sigma(n) - n$ , es decir,  $n$  es igual a la suma de sus factores entre 1 y  $n - 1$ . Por ejemplo, 28 es un número perfecto, porque  $28 = 1 + 2 + 4 + 7 + 14$ .

## Número de primos

Es fácil demostrar que hay un número infinito de primos. Si el número de primos fuera finito, podríamos construir un conjunto  $P = \{p_1, p_2, \dots, p_n\}$  que contendría todos los primos. Por ejemplo,  $p_1 = 2$ ,  $p_2 = 3$ ,  $p_3 = 5$ , y así sucesivamente. Sin embargo, usando  $P$ , podríamos formar un nuevo primo

$$p_1 p_2 \cdots p_n + 1$$

que es más grande que todos los elementos de  $P$ . Esta es una contradicción, y el número de primos tiene que ser infinito.

## Densidad de primos

La densidad de los primos significa qué tan a menudo hay primos entre los números. Sea  $\pi(n)$  el número de primos entre 1 y  $n$ . Por ejemplo,  $\pi(10) = 4$ , porque hay 4 primos entre 1 y 10: 2, 3, 5 y 7.

Es posible demostrar que

$$\pi(n) \approx \frac{n}{\ln n},$$

lo que significa que los primos son bastante frecuentes. Por ejemplo, el número de primos entre 1 y  $10^6$  es  $\pi(10^6) = 78498$ , y  $10^6 / \ln 10^6 \approx 72382$ .

## Conjeturas

Hay muchas *conjeturas* que involucran primos. La mayoría de la gente piensa que las conjeturas son ciertas, pero nadie ha podido demostrarlas. Por ejemplo, las siguientes conjeturas son famosas:

- **Conjetura de Goldbach:** Cada entero par  $n > 2$  se puede representar como una suma  $n = a + b$  de modo que tanto  $a$  como  $b$  sean primos.
- **Conjetura de los primos gemelos:** Existe un número infinito de pares de la forma  $\{p, p + 2\}$ , donde tanto  $p$  como  $p + 2$  son primos.
- **Conjetura de Legendre:** Siempre hay un primo entre los números  $n^2$  y  $(n + 1)^2$ , donde  $n$  es cualquier entero positivo.

## Algoritmos básicos

Si un número  $n$  no es primo, se puede representar como un producto  $a \cdot b$ , donde  $a \leq \sqrt{n}$  o  $b \leq \sqrt{n}$ , por lo que ciertamente tiene un factor entre 2 y  $\lfloor \sqrt{n} \rfloor$ . Usando esta observación, podemos tanto probar si un número es primo como encontrar la factorización prima de un número en  $O(\sqrt{n})$  tiempo. La siguiente función `prime` comprueba si el número dado  $n$  es primo. La función intenta dividir  $n$  por todos los números entre 2 y  $\lfloor \sqrt{n} \rfloor$ , y si ninguno de ellos divide  $n$ , entonces  $n$  es primo.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

La siguiente función `factors` construye un vector que contiene la factorización prima de  $n$ . La función divide  $n$  por sus factores primos, y los agrega al vector. El proceso termina cuando el número restante  $n$  no tiene factores entre 2 y  $\lfloor \sqrt{n} \rfloor$ . Si  $n > 1$ , es primo y el último factor.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Tenga en cuenta que cada factor primo aparece en el vector tantas veces como divide al número. Por ejemplo,  $24 = 2^3 \cdot 3$ , por lo que el resultado de la función es  $[2, 2, 2, 3]$ .

## Criba de Eratóstenes

La **criba de Eratóstenes** es un algoritmo de preprocesamiento que construye una matriz con la que podemos comprobar de forma eficiente si un número dado entre  $2 \dots n$  es primo y, si no lo es, encontrar un factor primo del número.

El algoritmo construye una matriz sieve cuyas posiciones  $2, 3, \dots, n$  se utilizan. El valor  $\text{sieve}[k] = 0$  significa que  $k$  es primo, y el valor  $\text{sieve}[k] \neq 0$  significa que  $k$  no es primo y uno de sus factores primos es  $\text{sieve}[k]$ .

El algoritmo itera a través de los números  $2 \dots n$  uno por uno. Siempre que se encuentra un nuevo primo  $x$ , el algoritmo registra que los múltiplos de  $x$  ( $2x, 3x, 4x, \dots$ ) no son primos, porque el número  $x$  los divide.

Por ejemplo, si  $n = 20$ , la matriz es la siguiente:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

El siguiente código implementa la criba de Eratóstenes. El código asume que cada elemento de sieve es inicialmente cero.

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}
```

El bucle interno del algoritmo se ejecuta  $n/x$  veces para cada valor de  $x$ . Por lo tanto, un límite superior para el tiempo de ejecución del algoritmo es la suma armónica

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

De hecho, el algoritmo es más eficiente, porque el bucle interno solo se ejecutará si el número  $x$  es primo. Se puede demostrar que el tiempo de ejecución del algoritmo es solo  $O(n \log \log n)$ , una complejidad muy cercana a  $O(n)$ .

## Algoritmo de Euclides

El **máximo común divisor** de los números  $a$  y  $b$ ,  $\text{gcd}(a, b)$ , es el número más grande que divide tanto a  $a$  como a  $b$ , y el **mínimo común múltiplo** de  $a$  y  $b$ ,  $\text{lcm}(a, b)$ , es el número más pequeño que es divisible por tanto  $a$  como  $b$ . Por ejemplo,  $\text{gcd}(24, 36) = 12$  y  $\text{lcm}(24, 36) = 72$ .

El máximo común divisor y el mínimo común múltiplo están conectados de la siguiente manera:

$$\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

**Algoritmo de Euclides**<sup>1</sup> proporciona una forma eficiente de encontrar el máximo común divisor de dos números. El algoritmo se basa en la siguiente fórmula:

$$\text{mcd}(a, b) = \begin{cases} a & b = 0 \\ \text{mcd}(b, a \bmod b) & b \neq 0 \end{cases}$$

Por ejemplo,

$$\text{mcd}(24, 36) = \text{mcd}(36, 24) = \text{mcd}(24, 12) = \text{mcd}(12, 0) = 12.$$

El algoritmo se puede implementar de la siguiente manera:

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

Se puede demostrar que el algoritmo de Euclides funciona en  $O(\log n)$  tiempo, donde  $n = \min(a, b)$ . El peor caso para el algoritmo es el caso en que  $a$  y  $b$  son números de Fibonacci consecutivos. Por ejemplo,

$$\text{mcd}(13, 8) = \text{mcd}(8, 5) = \text{mcd}(5, 3) = \text{mcd}(3, 2) = \text{mcd}(2, 1) = \text{mcd}(1, 0) = 1.$$

## Función totiente de Euler

Los números  $a$  y  $b$  son **coprimsos** si  $\text{mcd}(a, b) = 1$ . **Función totiente de Euler**  $\varphi(n)$  da el número de números coprimos a  $n$  entre 1 y  $n$ . Por ejemplo,  $\varphi(12) = 4$ , porque 1, 5, 7 y 11 son coprimos con 12.

El valor de  $\varphi(n)$  se puede calcular a partir de la factorización prima de  $n$  usando la fórmula

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1).$$

Por ejemplo,  $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$ . Tenga en cuenta que  $\varphi(n) = n - 1$  si  $n$  es primo.

## 21.2 Aritmética modular

En **aritmética modular**, el conjunto de números está limitado para que solo se utilicen los números  $0, 1, 2, \dots, m - 1$ , donde  $m$  es una constante. Cada número  $x$

<sup>1</sup>Euclides fue un matemático griego que vivió alrededor del año 300 a. C. Este es quizás el primer algoritmo conocido en la historia.

es representado por el número  $x \bmod m$ : el resto después de dividir  $x$  entre  $m$ . Por ejemplo, si  $m = 17$ , entonces  $75$  está representado por  $75 \bmod 17 = 7$ .

A menudo podemos tomar restos antes de hacer cálculos. En particular, las siguientes fórmulas se cumplen:

$$\begin{aligned}(x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\(x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\(x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\x^n \bmod m &= (x \bmod m)^n \bmod m\end{aligned}$$

## Exponenciación modular

A menudo es necesario calcular de manera eficiente el valor de  $x^n \bmod m$ . Esto se puede hacer en  $O(\log n)$  tiempo usando la siguiente recursión:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ es par} \\ x^{n-1} \cdot x & n \text{ es impar} \end{cases}$$

Es importante que en el caso de un  $n$  par, el valor de  $x^{n/2}$  se calcula solo una vez. Esto garantiza que la complejidad temporal del algoritmo es  $O(\log n)$ , porque  $n$  siempre se reduce a la mitad cuando es par.

La siguiente función calcula el valor de  $x^n \bmod m$ :

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x, n/2, m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

## Teorema de Fermat y teorema de Euler

**Teorema de Fermat** establece que

$$x^{m-1} \bmod m = 1$$

cuando  $m$  es primo y  $x$  y  $m$  son coprimos. Esto también produce

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

De manera más general, **teorema de Euler** establece que

$$x^{\varphi(m)} \bmod m = 1$$

cuando  $x$  y  $m$  son coprimos. El teorema de Fermat se deriva del teorema de Euler, porque si  $m$  es un primo, entonces  $\varphi(m) = m - 1$ .

## Inverso modular

El inverso de  $x$  módulo  $m$  es un número  $x^{-1}$  tal que

$$xx^{-1} \bmod m = 1.$$

Por ejemplo, si  $x = 6$  y  $m = 17$ , entonces  $x^{-1} = 3$ , porque  $6 \cdot 3 \bmod 17 = 1$ .

Usando inversos modulares, podemos dividir números módulo  $m$ , porque la división por  $x$  corresponde a la multiplicación por  $x^{-1}$ . Por ejemplo, para evaluar el valor de  $36/6 \bmod 17$ , podemos usar la fórmula  $2 \cdot 3 \bmod 17$ , porque  $36 \bmod 17 = 2$  y  $6^{-1} \bmod 17 = 3$ .

Sin embargo, un inverso modular no siempre existe. Por ejemplo, si  $x = 2$  y  $m = 4$ , la ecuación

$$xx^{-1} \bmod m = 1$$

no se puede resolver, porque todos los múltiplos de 2 son pares y el resto nunca puede ser 1 cuando  $m = 4$ . Resulta que el valor de  $x^{-1} \bmod m$  se puede calcular exactamente cuando  $x$  y  $m$  son coprimos.

Si existe un inverso modular, se puede calcular usando la fórmula

$$x^{-1} = x^{\varphi(m)-1}.$$

Si  $m$  es primo, la fórmula se convierte en

$$x^{-1} = x^{m-2}.$$

Por ejemplo,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Esta fórmula nos permite calcular eficientemente inversos modulares utilizando el algoritmo de exponenciación modular. La fórmula se puede derivar utilizando el teorema de Euler. Primero, el inverso modular debe satisfacer la siguiente ecuación:

$$xx^{-1} \bmod m = 1.$$

Por otro lado, según el teorema de Euler,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

por lo que los números  $x^{-1}$  y  $x^{\varphi(m)-1}$  son iguales.

## Aritmética informática

En la programación, los enteros sin signo se representan módulo  $2^k$ , donde  $k$  es el número de bits del tipo de datos. Una consecuencia habitual de esto es que un número se envuelve si se vuelve demasiado grande.

Por ejemplo, en C++, los números de tipo `unsigned int` se representan módulo  $2^{32}$ . El siguiente código declara una variable `unsigned int` cuyo valor es 123456789. Después de esto, el valor se multiplicará por sí mismo, y el resultado es  $123456789^2 \bmod 2^{32} = 2537071545$ .

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

## 21.3 Resolver ecuaciones

### Ecuaciones diofánticas

Una **ecuación diofántica** es una ecuación de la forma

$$ax + by = c,$$

donde  $a$ ,  $b$  y  $c$  son constantes y los valores de  $x$  e  $y$  deben encontrarse. Cada número en la ecuación debe ser un entero. Por ejemplo, una solución para la ecuación  $5x + 2y = 11$  es  $x = 3$  e  $y = -2$ .

Podemos resolver eficientemente una ecuación diofántica usando el algoritmo de Euclides. Resulta que podemos extender el algoritmo de Euclides para que encuentre números  $x$  e  $y$  que satisfagan la siguiente ecuación:

$$ax + by = \gcd(a, b)$$

Se puede resolver una ecuación diofántica si  $c$  es divisible por  $\gcd(a, b)$ , y de lo contrario no se puede resolver.

Como ejemplo, encontremos números  $x$  e  $y$  que satisfagan la siguiente ecuación:

$$39x + 15y = 12$$

La ecuación se puede resolver porque  $\gcd(39, 15) = 3$  y  $3 \mid 12$ . Cuando el algoritmo de Euclides calcula el máximo común divisor de 39 y 15, produce la siguiente secuencia de llamadas a funciones:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

Esto corresponde a las siguientes ecuaciones:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Usando estas ecuaciones, podemos derivar

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

y multiplicando esto por 4, el resultado es

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

por lo que una solución a la ecuación es  $x = 8$  e  $y = -20$ .

La solución a una ecuación diofántica no es única, porque podemos formar un número infinito de soluciones si conocemos una solución. Si un par  $(x, y)$  es una solución, entonces también todos los pares

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

son soluciones, donde  $k$  es cualquier entero.



## Teorema chino del resto

El **teorema chino del resto** resuelve un grupo de ecuaciones de la forma

$$\begin{aligned}x &= a_1 \bmod m_1 \\x &= a_2 \bmod m_2 \\&\dots \\x &= a_n \bmod m_n\end{aligned}$$

donde todos los pares de  $m_1, m_2, \dots, m_n$  son primos entre sí.

Sea  $x_m^{-1}$  el inverso de  $x$  módulo  $m$ , y

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Usando esta notación, una solución a las ecuaciones es

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

En esta solución, para cada  $k = 1, 2, \dots, n$ ,

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

porque

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Dado que todos los demás términos en la suma son divisibles por  $m_k$ , no tienen ningún efecto en el resto, y  $x \bmod m_k = a_k$ .

Por ejemplo, una solución para

$$\begin{aligned}x &= 3 \bmod 5 \\x &= 4 \bmod 7 \\x &= 2 \bmod 3\end{aligned}$$

es

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Una vez que hayamos encontrado una solución  $x$ , podemos crear un número infinito de otras soluciones, porque todos los números de la forma

$$x + m_1 m_2 \cdots m_n$$

son soluciones.

## 21.4 Otros resultados

### Teorema de Lagrange

El **teorema de Lagrange** establece que cada entero positivo se puede representar como una suma de cuatro cuadrados, es decir,  $a^2 + b^2 + c^2 + d^2$ . Por ejemplo, el número 123 se puede representar como la suma  $8^2 + 5^2 + 5^2 + 3^2$ .

## Teorema de Zeckendorf

**Teorema de Zeckendorf** establece que todo entero positivo tiene una representación única como suma de números de Fibonacci tal que ningún par de números son iguales o consecutivos números de Fibonacci. Por ejemplo, el número 74 se puede representar como la suma  $55 + 13 + 5 + 1$ .

## Triplas pitagóricas

Un **triple pitagórico** es una terna  $(a, b, c)$  que satisface el teorema de Pitágoras  $a^2 + b^2 = c^2$ , lo que significa que hay un triángulo rectángulo con longitudes de lado  $a$ ,  $b$  y  $c$ . Por ejemplo,  $(3, 4, 5)$  es un triple pitagórico.

Si  $(a, b, c)$  es un triple pitagórico, todos los triples de la forma  $(ka, kb, kc)$  también son triples pitagóricos donde  $k > 1$ . Un triple pitagórico es *primitivo* si  $a$ ,  $b$  y  $c$  son coprimos, y todos los triples pitagóricos se pueden construir a partir de triples primitivos utilizando un multiplicador  $k$ .

**La fórmula de Euclides** se puede utilizar para producir todos los triples pitagóricos primitivos. Cada uno de estos triples es de la forma

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

donde  $0 < m < n$ ,  $n$  y  $m$  son coprimos y al menos uno de  $n$  y  $m$  es par. Por ejemplo, cuando  $m = 1$  y  $n = 2$ , la fórmula produce el triple pitagórico más pequeño

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

## Teorema de Wilson

**El teorema de Wilson** establece que un número  $n$  es primo exactamente cuando

$$(n - 1)! \bmod n = n - 1.$$

Por ejemplo, el número 11 es primo, porque

$$10! \bmod 11 = 10,$$

y el número 12 no es primo, porque

$$11! \bmod 12 = 0 \neq 11.$$

Por lo tanto, el teorema de Wilson se puede utilizar para averiguar si un número es primo. Sin embargo, en la práctica, el teorema no se puede aplicar a valores grandes de  $n$ , porque es difícil calcular valores de  $(n - 1)!$  cuando  $n$  es grande.

# Chapter 22

## Combinatoria

**Combinatoria** estudia métodos para contar combinaciones de objetos. Usualmente, el objetivo es encontrar una manera de contar las combinaciones de manera eficiente sin generar cada combinación por separado.

Como ejemplo, considere el problema de contar la cantidad de formas de representar un entero  $n$  como una suma de enteros positivos. Por ejemplo, hay 8 representaciones para 4:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- $4$

Un problema combinatorio a menudo se puede resolver usando una función recursiva. En este problema, podemos definir una función  $f(n)$  que da la cantidad de representaciones para  $n$ . Por ejemplo,  $f(4) = 8$  de acuerdo con el ejemplo anterior. Los valores de la función se pueden calcular recursivamente de la siguiente manera:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

El caso base es  $f(0) = 1$ , porque la suma vacía representa el número 0. Entonces, si  $n > 0$ , consideramos todas las formas de elegir el primer número de la suma. Si el primer número es  $k$ , hay  $f(n-k)$  representaciones para la parte restante de la suma. Por lo tanto, calculamos la suma de todos los valores de la forma  $f(n-k)$  donde  $k < n$ .

Los primeros valores para la función son:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

A veces, una fórmula recursiva se puede reemplazar con una fórmula de forma cerrada. En este problema,

$$f(n) = 2^{n-1},$$

que se basa en el hecho de que hay  $n - 1$  posiciones posibles para los signos +- en la suma y podemos elegir cualquier subconjunto de ellos.

## 22.1 Coeficientes binomiales

El **coeficiente binomial**  $\binom{n}{k}$  es igual al número de formas en que podemos elegir un subconjunto de  $k$  elementos de un conjunto de  $n$  elementos. Por ejemplo,  $\binom{5}{3} = 10$ , porque el conjunto  $\{1, 2, 3, 4, 5\}$  tiene 10 subconjuntos de 3 elementos:

$$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$$

### Fórmula 1

Los coeficientes binomiales se pueden calcular recursivamente de la siguiente manera:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

La idea es fijar un elemento  $x$  en el conjunto. Si  $x$  está incluido en el subconjunto, tenemos que elegir  $k - 1$  elementos de  $n - 1$  elementos, y si  $x$  no está incluido en el subconjunto, tenemos que elegir  $k$  elementos de  $n - 1$  elementos.

Los casos base para la recursión son

$$\binom{n}{0} = \binom{n}{n} = 1,$$

porque siempre hay exactamente una forma de construir un subconjunto vacío y un subconjunto que contiene todos los elementos.

### Fórmula 2

Otra forma de calcular los coeficientes binomiales es la siguiente:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Hay  $n!$  permutaciones de  $n$  elementos. Recorremos todas las permutaciones y siempre incluimos los primeros  $k$  elementos de la permutación en el subconjunto. Dado que el orden de los elementos en el subconjunto y fuera del subconjunto no importa, el resultado se divide por  $k!$  y  $(n - k)!$

## Propiedades

Para los coeficientes binomiales,

$$\binom{n}{k} = \binom{n}{n-k},$$

porque en realidad dividimos un conjunto de  $n$  elementos en dos subconjuntos: el primero contiene  $k$  elementos y el segundo contiene  $n - k$  elementos.

La suma de los coeficientes binomiales es

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

La razón del nombre "coeficiente binomial" se puede ver cuando el binomio  $(a + b)$  se eleva a la  $n$ ésima potencia:

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

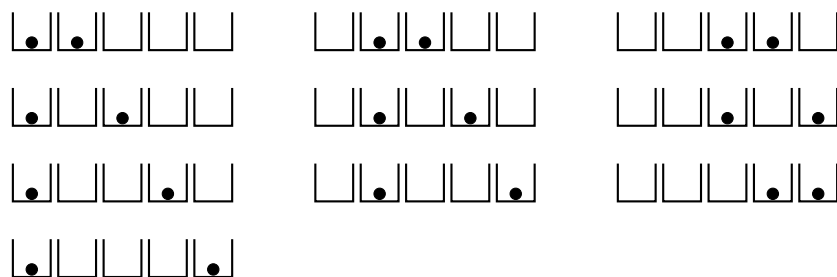
Los coeficientes binomiales también aparecen en el **Triángulo de Pascal** donde cada valor es igual a la suma de dos valores arriba:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ 1 & 4 & 6 & 4 & 1 & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \end{array}$$

## Cajas y bolas

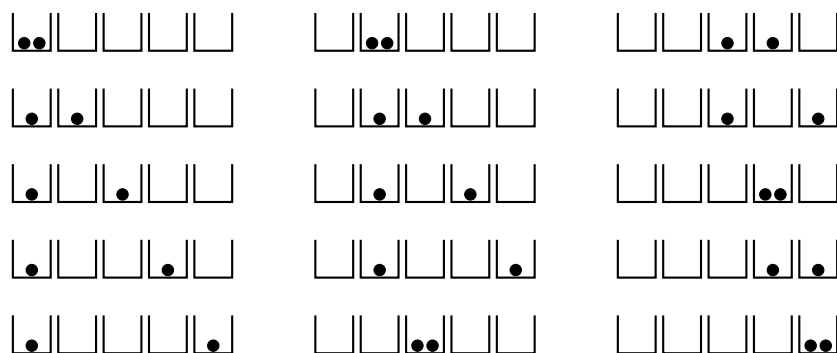
"Cajas y bolas" es un modelo útil, donde contamos las formas de colocar  $k$  bolas en  $n$  cajas. Consideremos tres escenarios:

*Escenario 1:* Cada caja puede contener como máximo una bola. Por ejemplo, cuando  $n = 5$  y  $k = 2$ , hay 10 soluciones:



En este escenario, la respuesta es directamente el coeficiente binomial  $\binom{n}{k}$ .

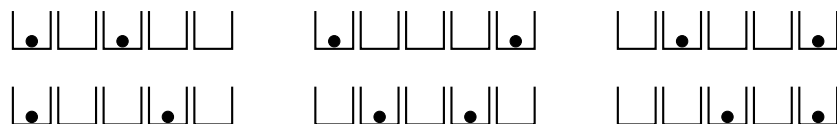
*Escenario 2:* Una caja puede contener múltiples bolas. Por ejemplo, cuando  $n = 5$  y  $k = 2$ , hay 15 soluciones:



El proceso de colocar las bolas en las cajas puede ser representado como una cadena que consiste en símbolos "o" y "→". Inicialmente, asumamos que estamos parados en la caja más a la izquierda. El símbolo "o" significa que colocamos una bola en la caja actual, y el símbolo "→" significa que nos movemos a la siguiente caja a la derecha.

Usando esta notación, cada solución es una cadena que contiene  $k$  veces el símbolo "o" y  $n - 1$  veces el símbolo "→". Por ejemplo, la solución de arriba a la derecha en la imagen anterior corresponde a la cadena "→ → o → o →". Así, el número de soluciones es  $\binom{k+n-1}{k}$ .

*Escenario 3:* Cada caja puede contener como máximo una bola, y además, ninguna de las dos cajas adyacentes puede contener una bola. Por ejemplo, cuando  $n = 5$  y  $k = 2$ , hay 6 soluciones:



En este escenario, podemos asumir que  $k$  bolas se colocan inicialmente en las cajas y hay una caja vacía entre cada dos cajas adyacentes. La tarea restante es elegir las posiciones para las cajas vacías restantes. Hay  $n - 2k + 1$  cajas de este tipo y  $k + 1$  posiciones para ellas. Así, usando la fórmula del escenario 2, el número de soluciones es  $\binom{n-k+1}{n-2k+1}$ .

## Coeficientes multinomiales

### El coeficiente multinomial

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

es igual al número de maneras en que podemos dividir  $n$  elementos en subconjuntos de tamaños  $k_1, k_2, \dots, k_m$ , donde  $k_1 + k_2 + \dots + k_m = n$ . Los coeficientes multinomiales pueden verse como una generalización de los coeficientes binomiales; si  $m = 2$ , la fórmula anterior corresponde a la fórmula del coeficiente binomial.

## 22.2 Números de Catalan

El **número de Catalan**  $C_n$  es igual al número de expresiones de paréntesis válidas que consisten en  $n$  paréntesis izquierdos y  $n$  paréntesis derechos.

Por ejemplo,  $C_3 = 5$ , porque podemos construir las siguientes expresiones de paréntesis usando tres paréntesis izquierdos y derechos:

- $()()()$
- $((()))$
- $()(())$
- $((()))$
- $((()()))$

### Expresiones de paréntesis

¿Qué es exactamente una *expresión de paréntesis válida*? Las siguientes reglas definen precisamente todas las expresiones de paréntesis válidas:

- Una expresión de paréntesis vacía es válida.
- Si una expresión  $A$  es válida, entonces también la expresión  $(A)$  es válida.
- Si las expresiones  $A$  y  $B$  son válidas, entonces también la expresión  $AB$  es válida.

Otra forma de caracterizar expresiones de paréntesis válidas es que si elegimos cualquier prefijo de dicha expresión, tiene que contener al menos tantos paréntesis izquierdos como paréntesis derechos. Además, la expresión completa tiene que contener un número igual de paréntesis izquierdos y derechos.

### Fórmula 1

Los números de Catalan se pueden calcular usando la fórmula

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

La suma recorre las formas de dividir la expresión en dos partes de modo que ambas partes sean válidas expresiones y la primera parte sea lo más corta posible pero no vacía. Para cualquier  $i$ , la primera parte contiene  $i + 1$  pares de paréntesis y el número de expresiones es el producto de los siguientes valores:

- $C_i$ : el número de formas de construir una expresión usando los paréntesis de la primera parte, sin contar los paréntesis externos
- $C_{n-i-1}$ : el número de formas de construir una expresión usando los paréntesis de la segunda parte

El caso base es  $C_0 = 1$ , porque podemos construir una expresión de paréntesis vacía usando cero pares de paréntesis.

## Fórmula 2

Los números de Catalan también se pueden calcular usando coeficientes binomiales:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

La fórmula se puede explicar de la siguiente manera:

Hay un total de  $\binom{2n}{n}$  formas de construir una expresión de paréntesis (no necesariamente válida) que contiene  $n$  paréntesis izquierdos y  $n$  paréntesis derechos. Calculemos el número de tales expresiones que *no* son válidas.

Si una expresión de paréntesis no es válida, tiene que contener un prefijo donde el número de paréntesis derechos excede el número de paréntesis izquierdos. La idea es invertir cada paréntesis que pertenece a tal prefijo. Por ejemplo, la expresión  $()()()$  contiene un prefijo  $()$ , y después de invertir el prefijo, la expresión se convierte en  $)((()()$ .

La expresión resultante consta de  $n+1$  paréntesis izquierdos y  $n-1$  paréntesis derechos. El número de tales expresiones es  $\binom{2n}{n+1}$ , que es igual al número de no válidas expresiones de paréntesis. Por lo tanto, el número de expresiones de paréntesis válidas se puede calcular usando la fórmula

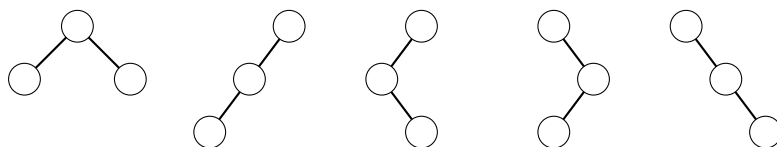
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

## Contar árboles

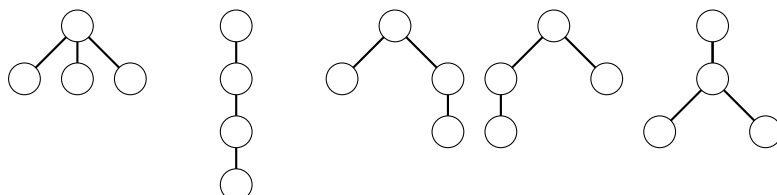
Los números de Catalan también están relacionados con los árboles:

- hay  $C_n$  árboles binarios de  $n$  nodos
- hay  $C_{n-1}$  árboles enraizados de  $n$  nodos

Por ejemplo, para  $C_3 = 5$ , los árboles binarios son



y los árboles enraizados son



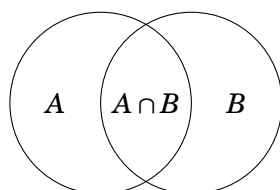


## 22.3 Inclusión-exclusión

**Inclusión-exclusión** es una técnica que se puede usar para contar el tamaño de una unión de conjuntos cuando se conocen los tamaños de las intersecciones, y viceversa. Un ejemplo simple de la técnica es la fórmula

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

donde  $A$  y  $B$  son conjuntos y  $|X|$  denota el tamaño de  $X$ . La fórmula se puede ilustrar como sigue:

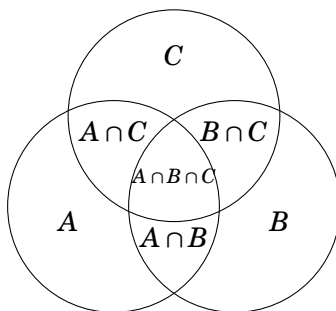


Nuestro objetivo es calcular el tamaño de la unión  $A \cup B$  que corresponde al área de la región que pertenece a al menos un círculo. La imagen muestra que podemos calcular el área de  $A \cup B$  sumando primero las áreas de  $A$  y  $B$  y luego restando el área de  $A \cap B$ .

La misma idea se puede aplicar cuando el número de conjuntos es mayor. Cuando hay tres conjuntos, la fórmula de inclusión-exclusión es

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

y la imagen correspondiente es



En el caso general, el tamaño de la unión  $X_1 \cup X_2 \cup \dots \cup X_n$  se puede calcular recorriendo todas las posibles intersecciones que contienen algunos de los conjuntos  $X_1, X_2, \dots, X_n$ . Si la intersección contiene un número impar de conjuntos, su tamaño se agrega a la respuesta, y de lo contrario su tamaño se resta de la respuesta.

Tenga en cuenta que hay fórmulas similares para calcular el tamaño de una intersección a partir de los tamaños de uniones. Por ejemplo,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

y

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

## Desarreglos

Como ejemplo, contemos el número de **desarreglos** de elementos  $\{1, 2, \dots, n\}$ , es decir, permutaciones donde ningún elemento permanece en su posición original. Por ejemplo, cuando  $n = 3$ , hay dos desarreglos:  $(2, 3, 1)$  y  $(3, 1, 2)$ .

Un enfoque para resolver el problema es usar inclusión-exclusión. Sea  $X_k$  el conjunto de permutaciones que contienen el elemento  $k$  en la posición  $k$ . Por ejemplo, cuando  $n = 3$ , los conjuntos son los siguientes:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Usando estos conjuntos, el número de desarreglos es igual a

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

por lo que basta con calcular el tamaño de la unión. Usando inclusión-exclusión, esto se reduce a calcular tamaños de intersecciones que se pueden hacer de manera eficiente. Por ejemplo, cuando  $n = 3$ , el tamaño de  $|X_1 \cup X_2 \cup X_3|$  es

$$\begin{aligned} &|X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

por lo que el número de soluciones es  $3! - 4 = 2$ .

Resulta que el problema también se puede resolver sin usar inclusión-exclusión. Sea  $f(n)$  el número de desarreglos para  $\{1, 2, \dots, n\}$ . Podemos usar la siguiente fórmula recursiva:

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

La fórmula se puede derivar considerando las posibilidades de cómo cambia el elemento 1 en el desarreglo. Hay  $n - 1$  formas de elegir un elemento  $x$  que reemplaza al elemento 1. En cada una de estas elecciones, hay dos opciones:

*Opción 1:* También reemplazamos el elemento  $x$  con el elemento 1. Después de esto, la tarea restante es construir un desordenamiento de  $n - 2$  elementos.

*Opción 2:* Reemplazamos el elemento  $x$  con algún otro elemento que no sea 1. Ahora tenemos que construir un desordenamiento de  $n - 1$  elementos, porque no podemos reemplazar el elemento  $x$  con el elemento 1, y todos los demás elementos deben ser cambiados.

## 22.4 Lema de Burnside

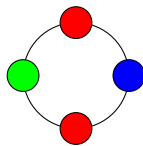
**Lema de Burnside** se puede usar para contar el número de combinaciones para que solo se cuente un representante para cada grupo de combinaciones simétricas.

El lema de Burnside establece que el número de combinaciones es

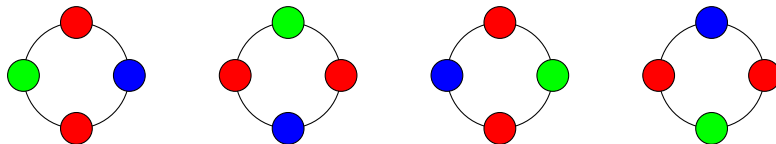
$$\sum_{k=1}^n \frac{c(k)}{n},$$

donde hay  $n$  maneras de cambiar la posición de una combinación, y hay  $c(k)$  combinaciones que permanecen sin cambios cuando se aplica la  $k$ -ésima forma.

Como ejemplo, calculemos el número de collares de  $n$  perlas, donde cada perla tiene  $m$  colores posibles. Dos collares son simétricos si son similares después de rotarlos. Por ejemplo, el collar



tiene los siguientes collares simétricos:



Hay  $n$  maneras de cambiar la posición de un collar, porque podemos rotarlo  $0, 1, \dots, n-1$  pasos en el sentido de las agujas del reloj. Si el número de pasos es 0, todos los  $m^n$  collares permanecen iguales, y si el número de pasos es 1, solo los  $m$  collares donde cada perla tiene el mismo color permanecen iguales.

Más generalmente, cuando el número de pasos es  $k$ , un total de

$$m^{\text{mcd}(k,n)}$$

collares permanecen iguales, donde  $\text{mcd}(k,n)$  es el máximo común divisor de  $k$  y  $n$ . La razón de esto es que los bloques de perlas de tamaño  $\text{mcd}(k,n)$  se reemplazarán entre sí. Por lo tanto, según el lema de Burnside, el número de collares es

$$\sum_{i=0}^{n-1} \frac{m^{\text{mcd}(i,n)}}{n}.$$

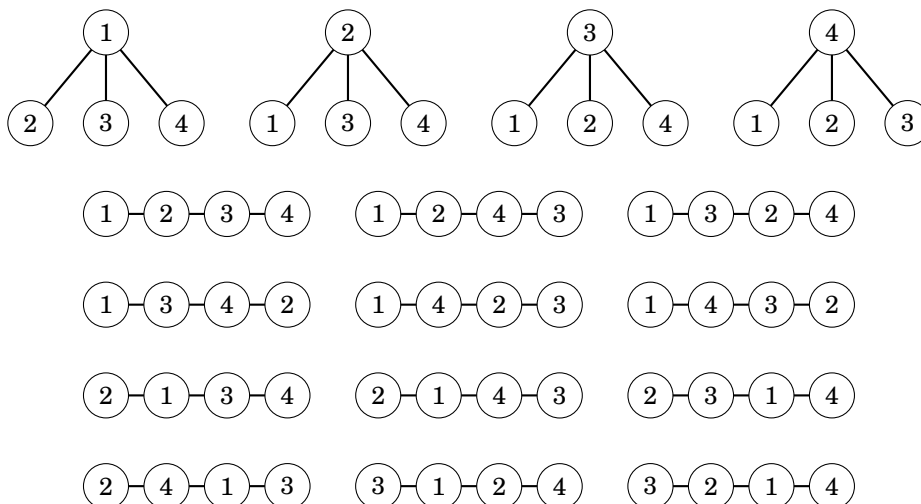
Por ejemplo, el número de collares de longitud 4 con 3 colores es

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

## 22.5 Fórmula de Cayley

**Fórmula de Cayley** establece que hay  $n^{n-2}$  árboles etiquetados que contienen  $n$  nodos. Los nodos están etiquetados  $1, 2, \dots, n$ , y dos árboles son diferentes si su estructura o etiquetado es diferente.

Por ejemplo, cuando  $n = 4$ , el número de árboles etiquetados es  $4^{4-2} = 16$ :

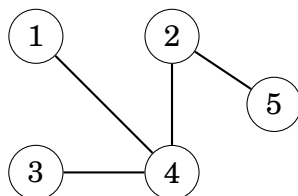


A continuación veremos cómo se puede derivar la fórmula de Cayley utilizando códigos de Prüfer.

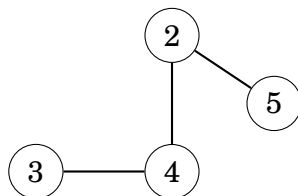
## Código de Prüfer

Un **código de Prüfer** es una secuencia de  $n - 2$  números que describe un árbol etiquetado. El código se construye siguiendo un proceso que elimina  $n - 2$  hojas del árbol. En cada paso, se elimina la hoja con la etiqueta más pequeña, y la etiqueta de su único vecino se agrega al código.

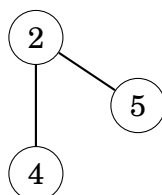
Por ejemplo, calculemos el código de Prüfer del siguiente gráfico:



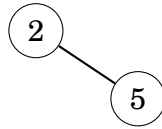
Primero eliminamos el nodo 1 y agregamos el nodo 4 al código:



Luego eliminamos el nodo 3 y agregamos el nodo 4 al código:



Finalmente eliminamos el nodo 4 y agregamos el nodo 2 al código:



Por lo tanto, el código de Prüfer del gráfico es  $[4, 4, 2]$ .

Podemos construir un código de Prüfer para cualquier árbol, y lo que es más importante, el árbol original se puede reconstruir a partir de un código de Prüfer. Por lo tanto, el número de árboles etiquetados de  $n$  nodos es igual a  $n^{n-2}$ , el número de códigos de Prüfer de tamaño  $n$ .



# Chapter 23

## Matrices

Una **matriz** es un concepto matemático que corresponde a una matriz bidimensional en programación. Por ejemplo,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

es una matriz de tamaño  $3 \times 4$ , es decir, tiene 3 filas y 4 columnas. La notación  $[i, j]$  se refiere a el elemento en la fila  $i$  y la columna  $j$  en una matriz. Por ejemplo, en la matriz anterior,  $A[2, 3] = 8$  y  $A[3, 1] = 9$ .

Un caso especial de una matriz es un **vector** que es una matriz unidimensional de tamaño  $n \times 1$ . Por ejemplo,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

es un vector que contiene tres elementos.

La **transpuesta**  $A^T$  de una matriz  $A$  se obtiene cuando las filas y columnas de  $A$  se intercambian, es decir,  $A^T[i, j] = A[j, i]$ :

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Una matriz es una **matriz cuadrada** si tiene el mismo número de filas y columnas. Por ejemplo, la siguiente matriz es una matriz cuadrada:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

### 23.1 Operaciones

La suma  $A + B$  de las matrices  $A$  y  $B$  se define si las matrices son del mismo tamaño. El resultado es una matriz donde cada elemento es la suma de los

elementos correspondientes en  $A$  y  $B$ .

Por ejemplo,

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Multiplicar una matriz  $A$  por un valor  $x$  significa que cada elemento de  $A$  se multiplica por  $x$ . Por ejemplo,

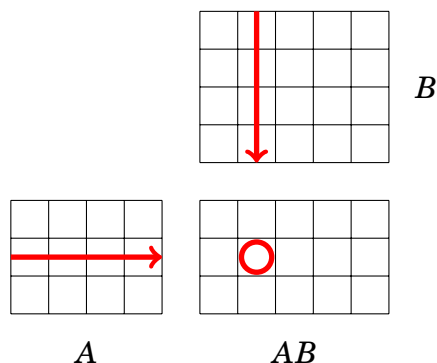
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

## Multiplicación de matrices

El producto  $AB$  de las matrices  $A$  y  $B$  se define si  $A$  es de tamaño  $a \times n$  y  $B$  es de tamaño  $n \times b$ , es decir, el ancho de  $A$  es igual a la altura de  $B$ . El resultado es una matriz de tamaño  $a \times b$  cuyos elementos se calculan utilizando la fórmula

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j].$$

La idea es que cada elemento de  $AB$  es una suma de productos de elementos de  $A$  y  $B$  de acuerdo con la siguiente imagen:



Por ejemplo,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

La multiplicación de matrices es asociativa, por lo que  $A(BC) = (AB)C$  se cumple, pero no es conmutativa, por lo que  $AB = BA$  no suele cumplirse.

Una **matriz identidad** es una matriz cuadrada donde cada elemento en la diagonal es 1 y todos los demás elementos son 0. Por ejemplo, la siguiente matriz es la matriz identidad  $3 \times 3$ :

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Multiplicar una matriz por una matriz identidad no la cambia. Por ejemplo,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{y} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Usando un algoritmo sencillo, podemos calcular el producto de dos matrices  $n \times n$  en tiempo  $O(n^3)$ . También existen algoritmos más eficientes para la multiplicación de matrices<sup>1</sup>, pero son principalmente de interés teórico y tales algoritmos no son necesarios en programación competitiva.

## Potencia de matriz

La potencia  $A^k$  de una matriz  $A$  se define si  $A$  es una matriz cuadrada. La definición se basa en la multiplicación de matrices:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ veces}}$$

Por ejemplo,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Además,  $A^0$  es una matriz identidad. Por ejemplo,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

La matriz  $A^k$  puede calcularse eficientemente en tiempo  $O(n^3 \log k)$  usando el algoritmo en el Capítulo 21.2. Por ejemplo,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

## Determinante

El **determinante**  $\det(A)$  de una matriz  $A$  se define si  $A$  es una matriz cuadrada. Si  $A$  es de tamaño  $1 \times 1$ , entonces  $\det(A) = A[1, 1]$ . El determinante de una matriz más grande es calculado recursivamente usando la fórmula

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

donde  $C[i, j]$  es el **cofactor** de  $A$  en  $[i, j]$ . El cofactor se calcula usando la fórmula

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

---

<sup>1</sup>El primer algoritmo de este tipo fue el algoritmo de Strassen, publicado en 1969 [63], cuya complejidad temporal es  $O(n^{2.80735})$ ; el mejor algoritmo actual [27] funciona en tiempo  $O(n^{2.37286})$

donde  $M[i, j]$  se obtiene eliminando la fila  $i$  y la columna  $j$  de  $A$ . Debido al coeficiente  $(-1)^{i+j}$  en el cofactor, cada determinante alternativo es positivo y negativo. Por ejemplo,

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

y

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

El determinante de  $A$  nos dice si existe una **matriz inversa**  $A^{-1}$  tal que  $A \cdot A^{-1} = I$ , donde  $I$  es una matriz identidad. Resulta que  $A^{-1}$  existe exactamente cuando  $\det(A) \neq 0$ , y se puede calcular usando la fórmula

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Por ejemplo,

$$\underbrace{\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{bmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_I.$$

## 23.2 Recurrencias lineales

Una **recurrencia lineal** es una función  $f(n)$  cuyos valores iniciales son  $f(0), f(1), \dots, f(k-1)$  y los valores más grandes se calculan recursivamente usando la fórmula

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

donde  $c_1, c_2, \dots, c_k$  son coeficientes constantes.

La programación dinámica se puede utilizar para calcular cualquier valor de  $f(n)$  en tiempo  $O(kn)$  calculando todos los valores de  $f(0), f(1), \dots, f(n)$  uno tras otro. Sin embargo, si  $k$  es pequeño, es posible calcular  $f(n)$  mucho más eficientemente en  $O(k^3 \log n)$  tiempo usando operaciones matriciales.

### Números de Fibonacci

Un ejemplo simple de una recurrencia lineal es la siguiente función que define los números de Fibonacci:

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

En este caso,  $k = 2$  y  $c_1 = c_2 = 1$ .

Para calcular eficientemente los números de Fibonacci, representamos la fórmula de Fibonacci como una matriz cuadrada  $X$  de tamaño  $2 \times 2$ , para la cual se cumple lo siguiente:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Así, los valores  $f(i)$  y  $f(i+1)$  se dan como "entrada" para  $X$ , y  $X$  calcula los valores  $f(i+1)$  y  $f(i+2)$  a partir de ellos. Resulta que dicha matriz es

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Por ejemplo,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Así, podemos calcular  $f(n)$  usando la fórmula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

El valor de  $X^n$  se puede calcular en tiempo  $O(\log n)$ , por lo que el valor de  $f(n)$  también se puede calcular en tiempo  $O(\log n)$ .

## Caso general

Consideremos ahora el caso general donde  $f(n)$  es cualquier recurrencia lineal. De nuevo, nuestro objetivo es construir una matriz  $X$  para la cual

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Tal matriz es

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

En las primeras  $k-1$  filas, cada elemento es 0 excepto que un elemento es 1. Estas filas reemplazan  $f(i)$  con  $f(i+1)$ ,  $f(i+1)$  con  $f(i+2)$ , y así sucesivamente. La última fila contiene los coeficientes de la recurrencia para calcular el nuevo valor  $f(i+k)$ .

Ahora,  $f(n)$  se puede calcular en  $O(k^3 \log n)$  tiempo usando la fórmula

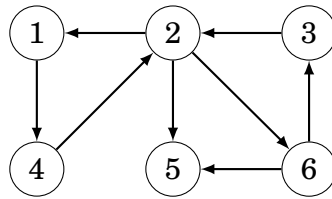
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

## 23.3 Gráficos y matrices

### Contando caminos

Las potencias de una matriz de adyacencia de un gráfico tienen una propiedad interesante. Cuando  $V$  es una matriz de adyacencia de un gráfico no ponderado, la matriz  $V^n$  contiene el número de caminos de  $n$  aristas entre los nodos en el gráfico.

Por ejemplo, para el gráfico



la matriz de adyacencia es

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Ahora, por ejemplo, la matriz

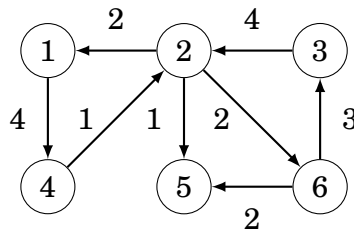
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

contiene el número de caminos de 4 aristas entre los nodos. Por ejemplo,  $V^4[2,5] = 2$ , porque hay dos caminos de 4 aristas desde el nodo 2 hasta el nodo 5:  $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$  y  $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$ .

### Caminos más cortos

Usando una idea similar en un gráfico ponderado, podemos calcular para cada par de nodos el mínimo longitud de un camino entre ellos que contiene exactamente  $n$  aristas. Para calcular esto, tenemos que definir la multiplicación de matrices de una nueva manera, para que no calculemos los números de caminos, sino que minimicemos las longitudes de los caminos.

Como ejemplo, considera el siguiente gráfico:



Construyamos una matriz de adyacencia donde  $\infty$  significa que una arista no existe, y otros valores corresponden a pesos de aristas. La matriz es

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

En lugar de la fórmula

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j]$$

ahora usamos la fórmula

$$AB[i,j] = \min_{k=1}^n A[i,k] + B[k,j]$$

para la multiplicación de matrices, por lo que calculamos un mínimo en lugar de una suma, y una suma de elementos en lugar de un producto. Después de esta modificación, las potencias de la matriz corresponden a las rutas más cortas en el gráfico.

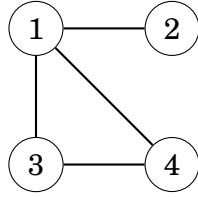
Por ejemplo, como

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

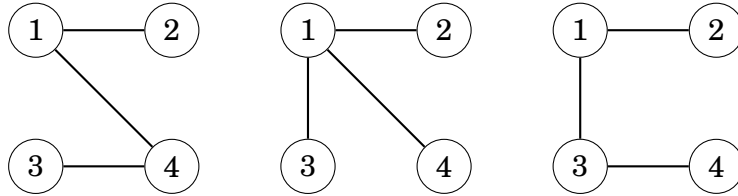
podemos concluir que la longitud mínima de una ruta de 4 aristas desde el nodo 2 hasta el nodo 5 es 8. Tal ruta es  $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ .

## Teorema de Kirchhoff

**Teorema de Kirchhoff** proporciona una forma de calcular el número de árboles de expansión de un gráfico como un determinante de una matriz especial. Por ejemplo, el gráfico



tiene tres árboles de expansión:



Para calcular el número de árboles de expansión, construimos una **matriz laplaciana**  $L$ , donde  $L[i, i]$  es el grado del nodo  $i$  y  $L[i, j] = -1$  si hay una arista entre los nodos  $i$  y  $j$ , y de lo contrario  $L[i, j] = 0$ . La matriz laplaciana para el gráfico anterior es la siguiente:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Se puede demostrar que el número de árboles de expansión es igual al determinante de una matriz que se obtiene cuando eliminamos cualquier fila y cualquier columna de  $L$ . Por ejemplo, si eliminamos la primera fila y columna, el resultado es

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

El determinante siempre es el mismo, independientemente de qué fila y columna eliminemos de  $L$ .

Tenga en cuenta que la fórmula de Cayley en el Capítulo 22.5 es un caso especial del teorema de Kirchhoff, porque en un gráfico completo de  $n$  nodos

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

# Chapter 24

## Probabilidad

Una **probabilidad** es un número real entre 0 y 1 que indica qué tan probable es un evento. Si un evento está seguro de ocurrir, su probabilidad es 1, y si un evento es imposible, su probabilidad es 0. La probabilidad de un evento se denota  $P(\dots)$  donde los tres puntos describen el evento.

Por ejemplo, al tirar un dado, el resultado es un entero entre 1 y 6, y la probabilidad de cada resultado es  $1/6$ . Por ejemplo, podemos calcular las siguientes probabilidades:

- $P(\text{"el resultado es 4"}) = 1/6$
- $P(\text{"el resultado no es 6"}) = 5/6$
- $P(\text{"el resultado es par"}) = 1/2$

### 24.1 Cálculo

Para calcular la probabilidad de un evento, podemos usar combinatoria o simular el proceso que genera el evento. Como ejemplo, calculemos la probabilidad de sacar tres cartas con el mismo valor de una baraja de cartas barajada (por ejemplo,  $\spadesuit 8$ ,  $\clubsuit 8$  y  $\diamondsuit 8$ ).

#### Método 1

Podemos calcular la probabilidad usando la fórmula

$$\frac{\text{número de resultados deseados}}{\text{número total de resultados}}.$$

En este problema, los resultados deseados son aquellos en los que el valor de cada carta es el mismo. Hay  $13\binom{4}{3}$  resultados de este tipo, porque hay 13 posibilidades para el valor de las cartas y  $\binom{4}{3}$  formas de elegir 3 palos de 4 palos posibles.

Hay un total de  $\binom{52}{3}$  resultados, porque elegimos 3 cartas de 52 cartas. Por lo tanto, la probabilidad del evento es

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

## Método 2

Otra forma de calcular la probabilidad es simular el proceso que genera el evento. En este ejemplo, sacamos tres cartas, por lo que el proceso consiste en tres pasos. Requerimos que cada paso del proceso sea exitoso.

Sacar la primera carta ciertamente tiene éxito, porque no hay restricciones. El segundo paso tiene éxito con una probabilidad de  $3/51$ , porque quedan 51 cartas y 3 de ellas tienen el mismo valor que la primera carta. De manera similar, el tercer paso tiene éxito con una probabilidad de  $2/50$ .

La probabilidad de que todo el proceso tenga éxito es

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

## 24.2 Eventos

Un evento en teoría de la probabilidad se puede representar como un conjunto

$$A \subset X,$$

donde  $X$  contiene todos los resultados posibles y  $A$  es un subconjunto de resultados. Por ejemplo, al sacar un dado, los resultados son

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Ahora, por ejemplo, el evento "el resultado es par" corresponde al conjunto

$$A = \{2, 4, 6\}.$$

A cada resultado  $x$  se le asigna una probabilidad  $p(x)$ . Luego, la probabilidad  $P(A)$  de un evento  $A$  se puede calcular como una suma de probabilidades de resultados usando la fórmula

$$P(A) = \sum_{x \in A} p(x).$$

Por ejemplo, al tirar un dado,  $p(x) = 1/6$  para cada resultado  $x$ , por lo que la probabilidad del evento "el resultado es par" es

$$p(2) + p(4) + p(6) = 1/2.$$

La probabilidad total de los resultados en  $X$  debe ser 1, es decir,  $P(X) = 1$ .

Dado que los eventos en teoría de la probabilidad son conjuntos, podemos manipularlos usando operaciones estándar de conjuntos:

- El **complemento**  $\bar{A}$  significa "A no ocurre". Por ejemplo, al tirar un dado, el complemento de  $A = \{2, 4, 6\}$  es  $\bar{A} = \{1, 3, 5\}$ .
- La **unión**  $A \cup B$  significa "A o B ocurren". Por ejemplo, la unión de  $A = \{2, 5\}$  y  $B = \{4, 5, 6\}$  es  $A \cup B = \{2, 4, 5, 6\}$ .
- La **intersección**  $A \cap B$  significa "A y B ocurren". Por ejemplo, la intersección de  $A = \{2, 5\}$  y  $B = \{4, 5, 6\}$  es  $A \cap B = \{5\}$ .



## Complemento

La probabilidad del complemento  $\bar{A}$  se calcula usando la fórmula

$$P(\bar{A}) = 1 - P(A).$$

A veces, podemos resolver un problema fácilmente usando complementos resolviendo el problema opuesto. Por ejemplo, la probabilidad de obtener al menos un seis al tirar un dado diez veces es

$$1 - (5/6)^{10}.$$

Aquí  $5/6$  es la probabilidad de que el resultado de un solo tiro no sea seis, y  $(5/6)^{10}$  es la probabilidad de que ninguno de los diez tiros sea un seis. El complemento de esto es la respuesta al problema.

## Unión

La probabilidad de la unión  $A \cup B$  se calcula usando la fórmula

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Por ejemplo, al tirar un dado, la unión de los eventos

$$A = \text{"el resultado es par"}$$

y

$$B = \text{"el resultado es menor que 4"}$$

es

$$A \cup B = \text{"el resultado es par o menor que 4",}$$

y su probabilidad es

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Si los eventos  $A$  y  $B$  son **disjuntos**, es decir,  $A \cap B$  está vacío, la probabilidad del evento  $A \cup B$  es simplemente

$$P(A \cup B) = P(A) + P(B).$$

## Probabilidad condicional

La **probabilidad condicional**

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

es la probabilidad de  $A$  asumiendo que  $B$  sucede. Por lo tanto, al calcular la probabilidad de  $A$ , solo consideramos los resultados que también pertenecen a  $B$ .

Usando los conjuntos previos,

$$P(A|B) = 1/3,$$

porque los resultados de  $B$  son  $\{1, 2, 3\}$ , y uno de ellos es par. Esta es la probabilidad de un resultado par si sabemos que el resultado está entre  $1 \dots 3$ .

## Intersección

Usando la probabilidad condicional, la probabilidad de la intersección  $A \cap B$  se puede calcular usando la fórmula

$$P(A \cap B) = P(A)P(B|A).$$

Los eventos  $A$  y  $B$  son **independientes** si

$$P(A|B) = P(A) \quad \text{y} \quad P(B|A) = P(B),$$

lo que significa que el hecho de que  $B$  suceda no cambia la probabilidad de  $A$ , y viceversa. En este caso, la probabilidad de la intersección es

$$P(A \cap B) = P(A)P(B).$$

Por ejemplo, al sacar una carta de una baraja, los eventos

$$A = \text{"el palo es trébol"}$$

y

$$B = \text{"el valor es cuatro"}$$

son independientes. Por lo tanto, el evento

$$A \cap B = \text{"la carta es el cuatro de tréboles"}$$

sucede con probabilidad

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

## 24.3 Variables aleatorias

Una **variable aleatoria** es un valor que se genera por un proceso aleatorio. Por ejemplo, al lanzar dos dados, una posible variable aleatoria es

$$X = \text{"la suma de los resultados"}.$$

Por ejemplo, si los resultados son  $[4, 6]$  (lo que significa que primero tiramos un cuatro y luego un seis), entonces el valor de  $X$  es 10.

Denotamos  $P(X = x)$  la probabilidad de que el valor de una variable aleatoria  $X$  es  $x$ . Por ejemplo, al lanzar dos dados,  $P(X = 10) = 3/36$ , porque el número total de resultados es 36 y hay tres formas posibles de obtener la suma 10:  $[4, 6]$ ,  $[5, 5]$  y  $[6, 4]$ .

## Valor esperado

El **valor esperado**  $E[X]$  indica el valor promedio de una variable aleatoria  $X$ . El valor esperado se puede calcular como la suma

$$\sum_x P(X = x)x,$$

donde  $x$  recorre todos los valores posibles de  $X$ .

Por ejemplo, al lanzar un dado, el resultado esperado es

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Una propiedad útil de los valores esperados es **linealidad**. Significa que la suma  $E[X_1 + X_2 + \dots + X_n]$  siempre es igual a la suma  $E[X_1] + E[X_2] + \dots + E[X_n]$ . Esta fórmula se cumple incluso si las variables aleatorias dependen unas de otras.

Por ejemplo, al lanzar dos dados, la suma esperada es

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Consideremos ahora un problema donde  $n$  bolas se colocan aleatoriamente en  $n$  cajas, y nuestra tarea es calcular el número esperado de cajas vacías. Cada bola tiene una probabilidad igual de ser colocada en cualquiera de las cajas. Por ejemplo, si  $n = 2$ , las posibilidades son las siguientes:



En este caso, el número esperado de cajas vacías es

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

En el caso general, la probabilidad de que a una sola caja esté vacía es

$$\left(\frac{n-1}{n}\right)^n,$$

porque ninguna bola debe ser colocada en ella. Por lo tanto, usando la linealidad, el número esperado de cajas vacías es

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

## Distribuciones

La **distribución** de una variable aleatoria  $X$  muestra la probabilidad de cada valor que  $X$  puede tener. La distribución consiste en valores  $P(X = x)$ . Por ejemplo, al lanzar dos dados, la distribución para su suma es:

$x$	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

En una **distribución uniforme**, la variable aleatoria  $X$  tiene  $n$  posibles valores  $a, a+1, \dots, b$  y la probabilidad de cada valor es  $1/n$ . Por ejemplo, al lanzar un dado,  $a = 1$ ,  $b = 6$  y  $P(X = x) = 1/6$  para cada valor  $x$ .

El valor esperado de  $X$  en una distribución uniforme es

$$E[X] = \frac{a+b}{2}.$$

En una **distribución binomial**, se realizan  $n$  intentos y la probabilidad de que un solo intento tenga éxito es  $p$ . La variable aleatoria  $X$  cuenta el número de intentos exitosos, y la probabilidad de un valor  $x$  es

$$P(X = x) = p^x(1-p)^{n-x} \binom{n}{x},$$

donde  $p^x$  y  $(1-p)^{n-x}$  corresponden a intentos exitosos y fallidos, y  $\binom{n}{x}$  es el número de formas en que podemos elegir el orden de los intentos.

Por ejemplo, al lanzar un dado diez veces, la probabilidad de obtener un seis exactamente tres veces es  $(1/6)^3(5/6)^7 \binom{10}{3}$ .

El valor esperado de  $X$  en una distribución binomial es

$$E[X] = pn.$$

En una **distribución geométrica**, la probabilidad de que un intento tenga éxito es  $p$ , y continuamos hasta que ocurre el primer éxito. La variable aleatoria  $X$  cuenta el número de intentos necesarios, y la probabilidad de un valor  $x$  es

$$P(X = x) = (1-p)^{x-1}p,$$

donde  $(1-p)^{x-1}$  corresponde a los intentos fallidos y  $p$  corresponde al primer intento exitoso.

Por ejemplo, si lanzamos un dado hasta que obtengamos un seis, la probabilidad de que el número de lanzamientos sea exactamente 4 es  $(5/6)^3 1/6$ .

El valor esperado de  $X$  en una distribución geométrica es

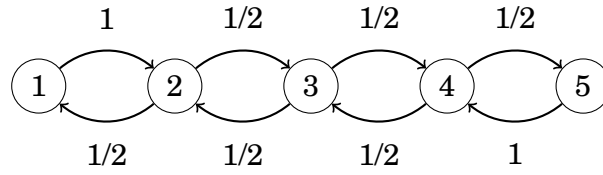
$$E[X] = \frac{1}{p}.$$

## 24.4 Cadenas de Markov

Una **cadena de Markov** es un proceso aleatorio que consiste en estados y transiciones entre ellos. Para cada estado, conocemos las probabilidades de pasar a otros estados. Una cadena de Markov puede representarse como un gráfico cuyos nodos son estados y los bordes son transiciones.

Como ejemplo, considere un problema donde estamos en el piso 1 de un edificio de  $n$  pisos. En cada paso, caminamos aleatoriamente un piso hacia arriba o un piso hacia abajo, excepto que siempre caminamos un piso hacia arriba desde el piso 1 y un piso hacia abajo desde el piso  $n$ . ¿Cuál es la probabilidad de estar en el piso  $m$  después de  $k$  pasos?

En este problema, cada piso del edificio corresponde a un estado en una cadena de Markov. Por ejemplo, si  $n = 5$ , el gráfico es el siguiente:



La distribución de probabilidad de una cadena de Markov es un vector  $[p_1, p_2, \dots, p_n]$ , donde  $p_k$  es la probabilidad de que el estado actual sea  $k$ . La fórmula  $p_1 + p_2 + \dots + p_n = 1$  siempre se cumple.

En el escenario anterior, la distribución inicial es  $[1, 0, 0, 0, 0]$ , porque siempre comenzamos en el piso 1. La siguiente distribución es  $[0, 1, 0, 0, 0]$ , porque solo podemos movernos del piso 1 al piso 2. Después de esto, podemos movernos un piso hacia arriba o un piso hacia abajo, por lo que la siguiente distribución es  $[1/2, 0, 1/2, 0, 0]$ , y así sucesivamente.

Una forma eficiente de simular la caminata en una cadena de Markov es usar la programación dinámica. La idea es mantener la distribución de probabilidad, y en cada paso recorrer todas las posibilidades de cómo podemos movernos. Usando este método, podemos simular una caminata de  $m$  pasos en  $O(n^2 m)$  tiempo.

Las transiciones de una cadena de Markov también pueden representarse como una matriz que actualiza la distribución de probabilidad. En el escenario anterior, la matriz es

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

Cuando multiplicamos una distribución de probabilidad por esta matriz, obtenemos la nueva distribución después de movernos un paso. Por ejemplo, podemos movernos desde la distribución  $[1, 0, 0, 0, 0]$  a la distribución  $[0, 1, 0, 0, 0]$  de la siguiente manera:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Calculando potencias de matrices de manera eficiente, podemos calcular la distribución después de  $m$  pasos en  $O(n^3 \log m)$  tiempo.

## 24.5 Algoritmos aleatorios

A veces podemos usar la aleatoriedad para resolver un problema, incluso si el problema no está relacionado con las probabilidades. Un **algoritmo aleatorio** es un algoritmo que se basa en la aleatoriedad.

Un **algoritmo de Monte Carlo** es un algoritmo aleatorio que a veces puede dar una respuesta incorrecta. Para que tal algoritmo sea útil, la probabilidad de una respuesta incorrecta debe ser pequeña.

Un **algoritmo de Las Vegas** es un algoritmo aleatorio que siempre da la respuesta correcta, pero su tiempo de ejecución varía aleatoriamente. El objetivo es diseñar un algoritmo que sea eficiente con alta probabilidad.

A continuación, veremos tres problemas de ejemplo que se pueden resolver utilizando la aleatoriedad.

## Estadística de orden

La  $k$ -ésima **estadística de orden** de una matriz es el elemento en la posición  $k$  después de ordenar la matriz en orden creciente. Es fácil calcular cualquier estadística de orden en tiempo  $O(n \log n)$  primero ordenando la matriz, pero ¿es realmente necesario ordenar toda la matriz solo para encontrar un elemento?

Resulta que podemos encontrar estadísticas de orden usando un algoritmo aleatorio sin ordenar la matriz. El algoritmo, llamado **quickselect**<sup>1</sup>, es un algoritmo de Las Vegas: su tiempo de ejecución suele ser  $O(n)$  pero  $O(n^2)$  en el peor de los casos.

El algoritmo elige un elemento aleatorio  $x$  de la matriz, y mueve los elementos más pequeños que  $x$  a la parte izquierda de la matriz, y todos los demás elementos a la parte derecha de la matriz. Esto lleva  $O(n)$  tiempo cuando hay  $n$  elementos. Supongamos que la parte izquierda contiene  $a$  elementos y la parte derecha contiene  $b$  elementos. Si  $a = k$ , el elemento  $x$  es la  $k$ -ésima estadística de orden. De lo contrario, si  $a > k$ , encontramos recursivamente la  $k$ -ésima orden estadística para la parte izquierda, y si  $a < k$ , encontramos recursivamente la  $r$ -ésima orden estadística para la parte derecha donde  $r = k - a$ . La búsqueda continúa de manera similar, hasta que el elemento se ha encontrado.

Cuando cada elemento  $x$  se elige aleatoriamente, el tamaño de la matriz se reduce aproximadamente a la mitad en cada paso, por lo que la complejidad temporal para encontrar la  $k$ -ésima estadística de orden es aproximadamente

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

El peor caso del algoritmo todavía requiere  $O(n^2)$  tiempo, porque es posible que  $x$  siempre se elija de tal manera que sea uno de los elementos más pequeños o más grandes de la matriz y se necesitan  $O(n)$  pasos. Sin embargo, la probabilidad de esto es tan pequeña que esto nunca sucede en la práctica.

## Verificación de la multiplicación de matrices

Nuestro siguiente problema es *verificar* si  $AB = C$  se cumple cuando  $A$ ,  $B$  y  $C$  son matrices de tamaño  $n \times n$ . Por supuesto, podemos resolver el problema calculando el producto  $AB$  de nuevo (en tiempo  $O(n^3)$  usando el algoritmo básico), pero uno

---

<sup>1</sup>En 1961, C. A. R. Hoare publicó dos algoritmos que son eficientes en promedio: **quicksort** [36] para ordenar matrices y **quickselect** [37] para encontrar estadísticas de orden.

podría esperar que la verificación de la respuesta sería más fácil que calcularla desde cero.

Resulta que podemos resolver el problema utilizando un algoritmo de Monte Carlo<sup>2</sup> cuyo la complejidad temporal es solo  $O(n^2)$ . La idea es simple: elegimos un vector aleatorio  $X$  de  $n$  elementos, y calculamos las matrices  $ABX$  y  $CX$ . Si  $ABX = CX$ , informamos que  $AB = C$ , y de lo contrario informamos que  $AB \neq C$ .

La complejidad temporal del algoritmo es  $O(n^2)$ , porque podemos calcular las matrices  $ABX$  y  $CX$  en tiempo  $O(n^2)$ . Podemos calcular la matriz  $ABX$  de manera eficiente utilizando la representación  $A(BX)$ , por lo que solo dos multiplicaciones de  $n \times n$  y  $n \times 1$  matrices de tamaño son necesarios.

El inconveniente del algoritmo es que existe una pequeña posibilidad de que el algoritmo cometa un error cuando informa que  $AB = C$ . Por ejemplo,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

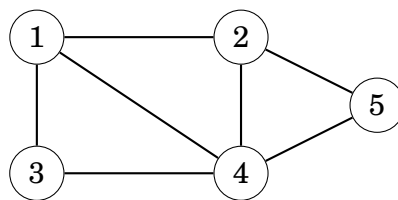
pero

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

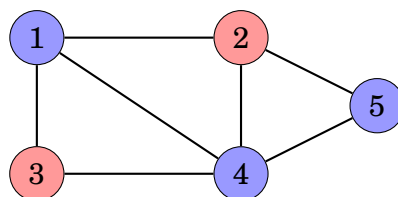
Sin embargo, en la práctica, la probabilidad de que el algoritmo cometa un error es pequeña, y podemos disminuir la probabilidad mediante la verificación del resultado usando múltiples vectores aleatorios  $X$  antes de informar que  $AB = C$ .

## Coloración de grafos

Dado un grafo que contiene  $n$  nodos y  $m$  aristas, nuestra tarea es encontrar una forma de colorear los nodos del grafo usando dos colores de modo que para al menos  $m/2$  aristas, los puntos finales tengan colores diferentes. Por ejemplo, en el grafo



una coloración válida es la siguiente:



<sup>2</sup>R. M. Freivalds publicó este algoritmo en 1977 [26], y a veces se llama **algoritmo de Freivalds**.

El grafo anterior contiene 7 aristas, y para 5 de ellas, los puntos finales tienen colores diferentes, por lo que la coloración es válida.

El problema se puede resolver usando un algoritmo de Las Vegas que genera coloraciones aleatorias hasta que se encuentra una coloración válida. En una coloración aleatoria, el color de cada nodo es elegido independientemente de modo que la probabilidad de ambos colores es  $1/2$ .

En una coloración aleatoria, la probabilidad de que los puntos finales de una sola arista tengan colores diferentes es  $1/2$ . Por lo tanto, el número esperado de aristas cuyos puntos finales tienen colores diferentes es  $m/2$ . Dado que se espera que una coloración aleatoria sea válida, encontraremos rápidamente una coloración válida en la práctica.



# Chapter 25

## Teoría de juegos

En este capítulo, nos centraremos en juegos de dos jugadores que no contienen elementos aleatorios. Nuestro objetivo es encontrar una estrategia que podamos seguir para ganar el juego sin importar lo que haga el oponente, si existe tal estrategia.

Resulta que existe una estrategia general para tales juegos, y podemos analizar los juegos utilizando la **teoría del nim**. Primero, analizaremos juegos simples donde los jugadores eliminan palos de pilas, y después de esto, generalizaremos la estrategia utilizada en esos juegos a otros juegos.

### 25.1 Estados del juego

Consideremos un juego donde inicialmente hay una pila de  $n$  palos. Los jugadores  $A$  y  $B$  se mueven alternativamente, y el jugador  $A$  comienza. En cada movimiento, el jugador tiene que eliminar 1, 2 o 3 palos de la pila, y el jugador que elimina el último palo gana el juego.

Por ejemplo, si  $n = 10$ , el juego puede proceder de la siguiente manera:

- El jugador  $A$  elimina 2 palos (quedan 8 palos).
- El jugador  $B$  elimina 3 palos (quedan 5 palos).
- El jugador  $A$  elimina 1 palo (quedan 4 palos).
- El jugador  $B$  elimina 2 palos (quedan 2 palos).
- El jugador  $A$  elimina 2 palos y gana.

Este juego consiste en estados  $0, 1, 2, \dots, n$ , donde el número del estado corresponde a la cantidad de palos restantes.

### Estados ganadores y perdedores

Un **estado ganador** es un estado donde el jugador ganará el juego si juega de manera óptima, y un **estado perdedor** es un estado donde el jugador perderá el juego si el oponente juega de manera óptima. Resulta que podemos clasificar todos los estados de un juego de modo que cada estado sea o bien un estado ganador o un estado perdedor.

En el juego anterior, el estado 0 es claramente un estado perdedor, porque el jugador no puede hacer ningún movimiento. Los estados 1, 2 y 3 son estados ganadores, porque podemos eliminar 1, 2 o 3 palos y ganar el juego. El estado 4, a su vez, es un estado perdedor, porque cualquier movimiento conduce a un estado que es un estado ganador para el oponente.

Más generalmente, si hay un movimiento que conduce desde el estado actual a un estado perdedor, el estado actual es un estado ganador, y de lo contrario el estado actual es un estado perdedor. Usando esta observación, podemos clasificar todos los estados de un juego comenzando con estados perdedores donde no hay movimientos posibles.

Los estados 0...15 del juego anterior se pueden clasificar de la siguiente manera ( $W$  denota un estado ganador y  $L$  denota un estado perdedor):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$	$L$	$W$	$W$	$W$

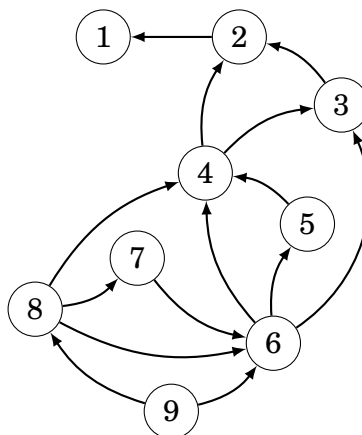
Es fácil analizar este juego: un estado  $k$  es un estado perdedor si  $k$  es divisible por 4, y de lo contrario es un estado ganador. Una forma óptima de jugar el juego es siempre elegir un movimiento después del cual la cantidad de palos en la pila sea divisible por 4. Finalmente, no quedan palos y el oponente ha perdido.

Por supuesto, esta estrategia requiere que la cantidad de palos *no* sea divisible por 4 cuando es nuestro movimiento. Si lo es, no hay nada que podamos hacer, y el oponente ganará el juego si juega de manera óptima.

## Gráfico de estado

Consideremos ahora otro juego de palos, donde en cada estado  $k$ , se permite eliminar cualquier número  $x$  de palos tal que  $x$  sea menor que  $k$  y divida  $k$ . Por ejemplo, en el estado 8 podemos eliminar 1, 2 o 4 palos, pero en el estado 7 el único movimiento permitido es eliminar 1 palo.

La siguiente imagen muestra los estados 1...9 del juego como un **gráfico de estado**, cuyos nodos son los estados y las aristas son los movimientos entre ellos:



El estado final en este juego es siempre el estado 1, que es un estado perdedor, porque no hay movimientos válidos. La clasificación de los estados 1...9 es la siguiente:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Sorprendentemente, en este juego, todos los estados con número par son estados ganadores, y todos los estados con número impar son estados perdedores.

## 25.2 Juego de Nim

El **juego de nim** es un juego simple que tiene un papel importante en la teoría de juegos, porque muchos otros juegos se pueden jugar usando la misma estrategia. Primero, nos centramos en nim, y luego generalizamos la estrategia a otros juegos.

Hay  $n$  montones en nim, y cada montón contiene un número de palitos. Los jugadores se mueven alternativamente, y en cada turno, el jugador elige un montón que aún contiene palitos y elimina cualquier número de palitos de él. El ganador es el jugador que elimina el último palito.

Los estados en nim son de la forma  $[x_1, x_2, \dots, x_n]$ , donde  $x_k$  denota el número de palitos en el montón  $k$ . Por ejemplo,  $[10, 12, 5]$  es un juego donde hay tres montones con 10, 12 y 5 palitos. El estado  $[0, 0, \dots, 0]$  es un estado perdedor, porque no es posible eliminar ningún palito, y este es siempre el estado final.

### Análisis

Resulta que podemos clasificar fácilmente cualquier estado nim calculando la **suma de nim**  $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$ , donde  $\oplus$  es la operación xor<sup>1</sup>. Los estados cuya suma de nim es 0 son estados perdedores, y todos los demás estados son estados ganadores. Por ejemplo, la suma de nim de  $[10, 12, 5]$  es  $10 \oplus 12 \oplus 5 = 3$ , por lo que el estado es un estado ganador.

¿Pero cómo se relaciona la suma de nim con el juego nim? Podemos explicar esto viendo cómo cambia la suma de nim cuando cambia el estado nim.

*Estados perdedores:* El estado final  $[0, 0, \dots, 0]$  es un estado perdedor, y su suma de nim es 0, como se esperaba. En otros estados perdedores, cualquier movimiento lleva a un estado ganador, porque cuando un solo valor  $x_k$  cambia, la suma de nim también cambia, por lo que la suma de nim es diferente de 0 después del movimiento.

*Estados ganadores:* Podemos movernos a un estado perdedor si hay algún montón  $k$  para el cual  $x_k \oplus s < x_k$ . En este caso, podemos eliminar palitos de el montón  $k$  de modo que contenga  $x_k \oplus s$  palitos, lo que conducirá a un estado perdedor. Siempre hay un montón así, donde  $x_k$  tiene un bit uno en la posición del bit uno más a la izquierda de  $s$ .

Como ejemplo, considere el estado  $[10, 12, 5]$ . Este estado es un estado ganador, porque su suma de nim es 3. Por lo tanto, tiene que haber un movimiento que conduzca a un estado perdedor. A continuación, averiguaremos tal movimiento.

La suma de nim del estado es la siguiente:

<sup>1</sup>La estrategia óptima para nim fue publicada en 1901 por C. L. Bouton [10].

10		1010
12		1100
5		0101
<hr/>		
3		0011

En este caso, el montón con 10 palitos es el único montón que tiene un bit uno en la posición del bit uno más a la izquierda de la suma de nim:

10		10 <u>1</u> 0
12		1100
5		0101
<hr/>		
3		00 <u>1</u> 1

El nuevo tamaño del montón tiene que ser  $10 \oplus 3 = 9$ , por lo que eliminaremos solo un palito. Después de esto, el estado será  $[9, 12, 5]$ , que es un estado perdedor:

9		1001
12		1100
5		0101
<hr/>		
0		0000

## Juego Misère

En un **juego misère**, el objetivo del juego es opuesto, por lo que el jugador que retira la última vara pierde el juego. Resulta que el juego de nim misère puede jugarse de forma óptima casi como el juego de nim estándar.

La idea es jugar primero al juego misère como al juego estándar, pero cambiar la estrategia al final del juego. La nueva estrategia se introducirá en una situación en la que cada pila contendría como máximo una vara después de la siguiente jugada.

En el juego estándar, debemos elegir una jugada después de la cual haya un número par de pilas con una vara. Sin embargo, en el juego misère, elegimos una jugada para que haya un número impar de pilas con una vara.

Esta estrategia funciona porque un estado en el que cambia la estrategia siempre aparece en el juego, y este estado es un estado ganador, porque contiene exactamente una pila que tiene más de una vara por lo que la suma nim no es 0.

## 25.3 Teorema de Sprague–Grundy

El **teorema de Sprague–Grundy**<sup>2</sup> generaliza la estrategia utilizada en el nim a todos los juegos que cumplen los siguientes requisitos:

- Hay dos jugadores que se turnan para mover.
- El juego consiste en estados, y las posibles jugadas en un estado no dependen de quién sea el turno.

<sup>2</sup>El teorema fue descubierto independientemente por R. Sprague [61] y P. M. Grundy [31].

- El juego termina cuando un jugador no puede hacer una jugada.
- El juego seguramente termina tarde o temprano.
- Los jugadores tienen información completa sobre los estados y las jugadas permitidas, y no hay aleatoriedad en el juego.

La idea es calcular para cada estado del juego un número de Grundy que corresponda al número de varas en una pila de nim. Cuando conocemos los números de Grundy de todos los estados, podemos jugar al juego como al juego de nim.

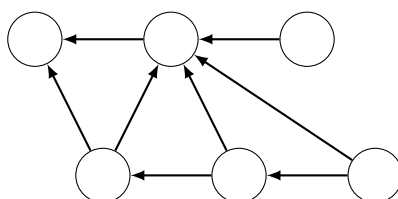
## Números de Grundy

El **número de Grundy** de un estado del juego es

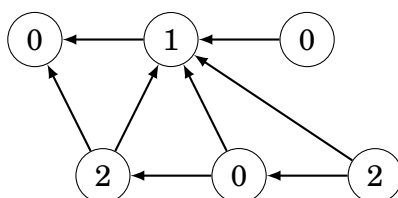
$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

donde  $g_1, g_2, \dots, g_n$  son los números de Grundy de los estados a los que podemos movernos, y la función mex da el menor número no negativo que no está en el conjunto. Por ejemplo,  $\text{mex}(\{0, 1, 3\}) = 2$ . Si no hay posibles movimientos en un estado, su número de Grundy es 0, porque  $\text{mex}(\emptyset) = 0$ .

Por ejemplo, en el gráfico de estado



los números de Grundy son los siguientes:



El número de Grundy de un estado perdedor es 0, y el número de Grundy de un estado ganador es un número positivo.

El número de Grundy de un estado corresponde a el número de varas en una pila de nim. Si el número de Grundy es 0, solo podemos movernos a estados cuyos números de Grundy sean positivos, y si el número de Grundy es  $x > 0$ , podemos movernos a estados cuyos números de Grundy incluyan todos los números  $0, 1, \dots, x - 1$ .

Como ejemplo, considera un juego donde los jugadores mueven una figura en un laberinto. Cada casilla del laberinto es o bien suelo o bien pared. En cada turno, el jugador tiene que mover la figura un número de pasos a la izquierda o hacia arriba. El ganador del juego es el jugador que hace la última jugada.

La siguiente imagen muestra un posible estado inicial del juego, donde @ denota la figura y \* denota una casilla donde puede moverse.

				*
				*
*	*	*	*	@

Los estados del juego son todos los cuadrados del suelo del laberinto. En el laberinto anterior, los números de Grundy son los siguientes:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Por lo tanto, cada estado del juego del laberinto corresponde a un montón en el juego de nim. Por ejemplo, el número de Grundy para el cuadrado inferior derecho es 2, por lo que es un estado ganador. Podemos llegar a un estado perdedor y ganar el juego moviendo ya sea cuatro pasos a la izquierda o dos pasos hacia arriba.

Tenga en cuenta que a diferencia del juego nim original, puede ser posible mover a un estado cuyo número de Grundy es mayor que el número de Grundy del estado actual. Sin embargo, el oponente siempre puede elegir una jugada que cancele esa jugada, por lo que no es posible escapar de un estado perdedor.

## Subjuegos

A continuación, asumiremos que nuestro juego consiste en subjuegos, y en cada turno, el jugador primero elige un subjuego y luego un movimiento en el subjuego. El juego termina cuando no es posible realizar ningún movimiento en ningún subjuego.

En este caso, el número de Grundy de un juego es la suma nim de los números de Grundy de los subjuegos. El juego se puede jugar como un juego de nim calculando todos los números de Grundy para los subjuegos y luego su suma nim.

Como ejemplo, considere un juego que consiste en tres laberintos. En este juego, en cada turno, el jugador elige uno de los laberintos y luego mueve la figura en el laberinto. Asuma que el estado inicial del juego es el siguiente:


Los números de Grundy para los laberintos son los siguientes:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

0	1	2	3	
1	0		0	1
2		0	1	2
3		1	2	0
4	0	2	5	3

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

En el estado inicial, la suma nim de los números de Grundy es  $2 \oplus 3 \oplus 3 = 2$ , entonces el primer jugador puede ganar el juego. Un movimiento óptimo es moverse dos pasos hacia arriba en el primer laberinto, lo que produce la suma nim  $0 \oplus 3 \oplus 3 = 0$ .

## El juego de Grundy

A veces un movimiento en un juego divide el juego en subjuegos que son independientes entre sí. En este caso, el número de Grundy del juego es

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

donde  $n$  es el número de posibles movimientos y

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

donde el movimiento  $k$  genera subjuegos con números de Grundy  $a_{k,1}, a_{k,2}, \dots, a_{k,m}$ .

Un ejemplo de tal juego es **el juego de Grundy**. Inicialmente, hay un solo montón que contiene  $n$  palitos. En cada turno, el jugador elige un montón y lo divide en dos montones no vacíos de modo que los montones sean de tamaño diferente. El jugador que realiza el último movimiento gana el juego.

Sea  $f(n)$  el número de Grundy de un montón que contiene  $n$  palitos. El número de Grundy se puede calcular pasando por todas las formas de dividir el montón en dos montones. Por ejemplo, cuando  $n = 8$ , las posibilidades son  $1 + 7$ ,  $2 + 6$  y  $3 + 5$ , entonces

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

En este juego, el valor de  $f(n)$  se basa en los valores de  $f(1), \dots, f(n-1)$ . Los casos base son  $f(1) = f(2) = 0$ , porque no es posible dividir los montones de 1 y 2 palitos. Los primeros números de Grundy son:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

El número de Grundy para  $n = 8$  es 2, entonces es posible ganar el juego. El movimiento ganador es crear montones  $1 + 7$ , porque  $f(1) \oplus f(7) = 0$ .





# Chapter 26

## Algoritmos de cadenas

Este capítulo trata sobre algoritmos eficientes para el procesamiento de cadenas. Muchos problemas de cadenas pueden resolverse fácilmente en tiempo  $O(n^2)$ , pero el desafío es encontrar algoritmos que funcionen en tiempo  $O(n)$  o  $O(n \log n)$ .

Por ejemplo, un problema fundamental de procesamiento de cadenas es el problema de **coincidencia de patrones**: dada una cadena de longitud  $n$  y un patrón de longitud  $m$ , nuestra tarea es encontrar las ocurrencias del patrón en la cadena. Por ejemplo, el patrón ABC ocurre dos veces en la cadena ABABCBABC.

El problema de coincidencia de patrones se puede resolver fácilmente en tiempo  $O(nm)$  mediante un algoritmo de fuerza bruta que prueba todas las posiciones donde el patrón puede ocurrir en la cadena. Sin embargo, en este capítulo, veremos que hay algoritmos más eficientes que requieren solo tiempo  $O(n + m)$ .

### 26.1 Terminología de cadenas

A lo largo del capítulo, asumimos que se utiliza la indexación basada en cero en las cadenas. Por lo tanto, una cadena  $s$  de longitud  $n$  consiste en caracteres  $s[0], s[1], \dots, s[n-1]$ . El conjunto de caracteres que pueden aparecer en las cadenas se denomina **alfabeto**. Por ejemplo, el alfabeto  $\{A, B, \dots, Z\}$  consiste en las letras mayúsculas del inglés.

Una **subcadena** es una secuencia de caracteres consecutivos en una cadena. Usamos la notación  $s[a \dots b]$  para referirnos a una subcadena de  $s$  que comienza en la posición  $a$  y termina en la posición  $b$ . Una cadena de longitud  $n$  tiene  $n(n+1)/2$  subcadenas. Por ejemplo, las subcadenas de ABCD son A, B, C, D, AB, BC, CD, ABC, BCD y ABCD.

Una **subsecuencia** es una secuencia de (no necesariamente consecutivos) caracteres en una cadena en su orden original. Una cadena de longitud  $n$  tiene  $2^n - 1$  subsecuencias. Por ejemplo, las subsecuencias de ABCD son A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD y ABCD.

Un **prefijo** es una subcadena que comienza al principio de una cadena, y un **sufijo** es una subcadena que termina al final de una cadena. Por ejemplo, los prefijos de ABCD son A, AB, ABC y ABCD, y los sufijos de ABCD son D, CD, BCD y ABCD.

Una **rotación** se puede generar moviendo los caracteres de una cadena uno por uno desde el principio hasta el final (o viceversa). Por ejemplo, las rotaciones de ABCD son ABCD, BCDA, CDAB y DABC.

Un **periodo** es un prefijo de una cadena tal que la cadena se puede construir repitiendo el periodo. La última repetición puede ser parcial y contener solo un prefijo del periodo. Por ejemplo, el periodo más corto de ABCABCA es ABC.

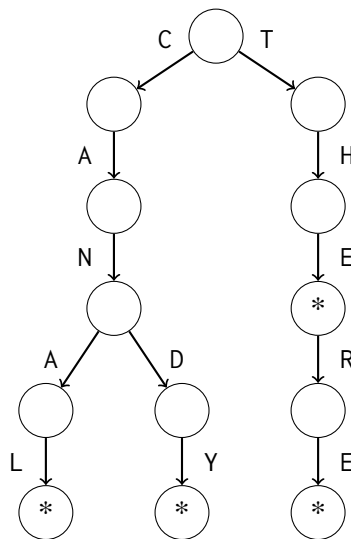
Un **borde** es una cadena que es a la vez un prefijo y un sufijo de una cadena. Por ejemplo, los bordes de ABACABA son A, ABA y ABACABA.

Las cadenas se comparan usando el **orden lexicográfico** (que corresponde al orden alfabético). Significa que  $x < y$  si  $x \neq y$  y  $x$  es un prefijo de  $y$ , o existe una posición  $k$  tal que  $x[i] = y[i]$  cuando  $i < k$  y  $x[k] < y[k]$ .

## 26.2 Estructura de trie

Un **trie** es un árbol enraizado que mantiene un conjunto de cadenas. Cada cadena del conjunto se almacena como una cadena de caracteres que comienza en la raíz. Si dos cadenas tienen un prefijo común, también tienen una cadena común en el árbol.

Por ejemplo, considere el siguiente trie:



Este trie corresponde al conjunto {CANAL, CANDY, THE, THERE}. El carácter \* en un nodo significa que una cadena en el conjunto termina en el nodo. Dicho carácter es necesario, porque una cadena puede ser un prefijo de otra cadena. Por ejemplo, en el trie anterior, THE es un prefijo de THERE.

Podemos comprobar en tiempo  $O(n)$  si un trie contiene una cadena de longitud  $n$ , porque podemos seguir la cadena que comienza en el nodo raíz. También podemos agregar una cadena de longitud  $n$  al trie en tiempo  $O(n)$  siguiendo primero la cadena y luego agregando nuevos nodos al trie si es necesario.

Usando un trie, podemos encontrar el prefijo más largo de una cadena dada tal que el prefijo pertenezca al conjunto. Además, al almacenar información

adicional en cada nodo, podemos calcular el número de cadenas que pertenecen al conjunto y tienen un cadena dada como prefijo.

Un trie se puede almacenar en una matriz

```
int trie[N][A];
```

donde  $N$  es el número máximo de nodos (la longitud total máxima de las cadenas en el conjunto) y  $A$  es el tamaño del alfabeto. Los nodos de un trie están numerados  $0, 1, 2, \dots$  de modo que el número de la raíz es 0, y  $\text{trie}[s][c]$  es el siguiente nodo en la cadena cuando nos movemos desde el nodo  $s$  usando el carácter  $c$ .

## 26.3 Hashing de cadenas

**Hashing de cadenas** es una técnica que nos permite comprobar de forma eficiente si dos cadenas son iguales<sup>1</sup>. La idea en el hashing de cadenas es comparar los valores hash de cadenas en lugar de sus caracteres individuales.

### Calculando valores hash

Un **valor hash** de una cadena es un número que se calcula a partir de los caracteres de la cadena. Si dos cadenas son iguales, sus valores hash también son iguales, lo que permite comparar cadenas basándose en sus valores hash.

Una forma habitual de implementar el hashing de cadenas es **hashing polinomial**, lo que significa que el valor hash de una cadena  $s$  de longitud  $n$  es

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

donde  $s[0], s[1], \dots, s[n-1]$  se interpretan como los códigos de los caracteres de  $s$ , y  $A$  y  $B$  son constantes preseleccionadas.

Por ejemplo, los códigos de los caracteres de ALLEY son:

A	L	L	E	Y
65	76	76	69	89

Por lo tanto, si  $A = 3$  y  $B = 97$ , el valor hash de ALLEY es

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

### Preprocesamiento

Usando el hashing polinomial, podemos calcular el valor hash de cualquier subcadena de una cadena  $s$  en tiempo  $O(1)$  después de un preprocesamiento en tiempo  $O(n)$ . La idea es construir una matriz  $h$  tal que  $h[k]$  contenga el valor hash

<sup>1</sup>La técnica fue popularizada por el algoritmo de coincidencia de patrones de Karp–Rabin [42].

del prefijo  $s[0 \dots k]$ . Los valores de la matriz se pueden calcular recursivamente como sigue:

$$\begin{aligned}h[0] &= s[0] \\h[k] &= (h[k-1]A + s[k]) \bmod B\end{aligned}$$

Además, construimos una matriz  $p$  donde  $p[k] = A^k \bmod B$ :

$$\begin{aligned}p[0] &= 1 \\p[k] &= (p[k-1]A) \bmod B.\end{aligned}$$

Construir estas matrices toma  $O(n)$  tiempo. Después de esto, el valor hash de cualquier subcadena  $s[a \dots b]$  se puede calcular en tiempo  $O(1)$  usando la fórmula

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

asumiendo que  $a > 0$ . Si  $a = 0$ , el valor hash es simplemente  $h[b]$ .

## Usando valores hash

Podemos comparar cadenas de manera eficiente usando valores hash. En lugar de comparar los caracteres individuales de las cadenas, la idea es comparar sus valores hash. Si los valores hash son iguales, las cadenas son *probablemente* iguales, y si los valores hash son diferentes, las cadenas son *ciertamente* diferentes.

Usando hashing, a menudo podemos hacer un algoritmo de fuerza bruta eficiente. Como ejemplo, considere el problema de coincidencia de patrones: dada una cadena  $s$  y un patrón  $p$ , encuentre las posiciones donde  $p$  ocurre en  $s$ . Un algoritmo de fuerza bruta recorre todas las posiciones donde  $p$  puede ocurrir y compara las cadenas carácter por carácter. La complejidad temporal de tal algoritmo es  $O(n^2)$ .

Podemos hacer que el algoritmo de fuerza bruta sea más eficiente usando hashing, porque el algoritmo compara subcadenas de cadenas. Usando hashing, cada comparación solo toma  $O(1)$  tiempo, porque solo se comparan los valores hash de las subcadenas. Esto da como resultado un algoritmo con complejidad temporal  $O(n)$ , que es la mejor complejidad temporal posible para este problema.

Combinando hashing y *búsqueda binaria*, también es posible averiguar el orden lexicográfico de dos cadenas en tiempo logarítmico. Esto se puede hacer calculando la longitud del prefijo común de las cadenas usando búsqueda binaria. Una vez que conocemos la longitud del prefijo común, solo podemos verificar el siguiente carácter después del prefijo, porque esto determina el orden de las cadenas.

## Colisiones y parámetros

Un riesgo evidente al comparar valores hash es una **colisión**, lo que significa que dos cadenas tienen contenidos diferentes pero valores hash iguales. En este caso, un algoritmo que se basa en los valores hash concluye que las cadenas son iguales, pero en realidad no lo son, y el algoritmo puede dar resultados incorrectos.

Las colisiones siempre son posibles, porque el número de cadenas diferentes es mayor que el número de valores hash diferentes. Sin embargo, la probabilidad

de una colisión es pequeña si las constantes  $A$  y  $B$  se eligen cuidadosamente. Una forma habitual es elegir constantes aleatorias cerca de  $10^9$ , por ejemplo de la siguiente manera:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Usando tales constantes, el tipo `long long` se puede usar al calcular valores hash, porque los productos  $AB$  y  $BB$  cabrán en `long long`. Pero ¿es suficiente tener alrededor de  $10^9$  valores hash diferentes?

Consideremos tres escenarios donde se puede utilizar el hashing:

*Escenario 1:* Las cadenas  $x$  e  $y$  se comparan con entre sí. La probabilidad de una colisión es  $1/B$  asumiendo que todos los valores hash son igualmente probables.

*Escenario 2:* Una cadena  $x$  se compara con cadenas  $y_1, y_2, \dots, y_n$ . La probabilidad de una o más colisiones es

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

*Escenario 3:* Todos los pares de cadenas  $x_1, x_2, \dots, x_n$  se comparan entre sí. La probabilidad de una o más colisiones es

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

La siguiente tabla muestra las probabilidades de colisión cuando  $n = 10^6$  y el valor de  $B$  varía:

constante $B$	escenario 1	escenario 2	escenario 3
$10^3$	0.001000	1.000000	1.000000
$10^6$	0.000001	0.632121	1.000000
$10^9$	0.000000	0.001000	1.000000
$10^{12}$	0.000000	0.000000	0.393469
$10^{15}$	0.000000	0.000000	0.000500
$10^{18}$	0.000000	0.000000	0.000001

La tabla muestra que en el escenario 1, la probabilidad de una colisión es insignificante cuando  $B \approx 10^9$ . En el escenario 2, una colisión es posible pero la probabilidad sigue siendo bastante pequeña. Sin embargo, en el escenario 3 la situación es muy diferente: una colisión ocurrirá casi siempre cuando  $B \approx 10^9$ .

El fenómeno en el escenario 3 se conoce como el **paradoja del cumpleaños**: si hay  $n$  personas en una habitación, la probabilidad de que *algunas* dos personas tengan el mismo cumpleaños es grande incluso si  $n$  es bastante pequeño. En hashing, en consecuencia, cuando todos los valores hash se comparan entre sí, la probabilidad de que algunos dos valores hash sean iguales es grande.

Podemos hacer que la probabilidad de una colisión sea más pequeña calculando *múltiples* valores hash usando diferentes parámetros. Es poco probable que ocurra una colisión en todos los valores hash al mismo tiempo. Por ejemplo, dos valores hash con parámetro  $B \approx 10^9$  corresponden a un valor hash con parámetro  $B \approx 10^{18}$ , lo que hace que la probabilidad de una colisión sea muy pequeña.

Algunas personas usan las constantes  $B = 2^{32}$  y  $B = 2^{64}$ , lo cual es conveniente, porque las operaciones con 32 y 64 enteros de bits se calculan módulo  $2^{32}$  y  $2^{64}$ . Sin embargo, esta *no* es una buena elección, porque es posible construir entradas que siempre generen colisiones cuando se usan constantes de la forma  $2^x$  [51].

## 26.4 Algoritmo Z

El **arreglo-Z**  $z$  de una cadena  $s$  de longitud  $n$  contiene para cada  $k = 0, 1, \dots, n-1$  la longitud de la subcadena más larga de  $s$  que comienza en la posición  $k$  y es un prefijo de  $s$ . Por lo tanto,  $z[k] = p$  nos dice que  $s[0 \dots p-1]$  es igual a  $s[k \dots k+p-1]$ . Muchos problemas de procesamiento de cadenas se pueden resolver de manera eficiente usando el arreglo-Z.

Por ejemplo, el arreglo-Z de ACBACDACBACBACDA es el siguiente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

En este caso, por ejemplo,  $z[6] = 5$ , porque la subcadena ACBAC de longitud 5 es un prefijo de  $s$ , pero la subcadena ACBACB de longitud 6 no es un prefijo de  $s$ .

### Descripción del algoritmo

A continuación describimos un algoritmo, llamado el **algoritmo-Z<sup>2</sup>**, que construye de manera eficiente el arreglo-Z en tiempo  $O(n)$ . El algoritmo calcula los valores del arreglo-Z de izquierda a derecha utilizando tanto información ya almacenada en el arreglo-Z como comparando subcadenas caracter por caracter.

Para calcular de manera eficiente los valores del arreglo-Z, el algoritmo mantiene un rango  $[x, y]$  tal que  $s[x \dots y]$  es un prefijo de  $s$  e  $y$  es lo más grande posible. Dado que sabemos que  $s[0 \dots y-x]$  y  $s[x \dots y]$  son iguales, podemos usar esta información al calcular los valores Z para las posiciones  $x+1, x+2, \dots, y$ .

En cada posición  $k$ , primero verificamos el valor de  $z[k-x]$ . Si  $k+z[k-x] < y$ , sabemos que  $z[k] = z[k-x]$ . Sin embargo, si  $k+z[k-x] \geq y$ ,  $s[0 \dots y-k]$  es igual a  $s[k \dots y]$ , y para determinar el valor de  $z[k]$  necesitamos comparar las subcadenas caracter por caracter. Aún así, el algoritmo funciona en tiempo  $O(n)$ , porque comenzamos a comparar en las posiciones  $y-k+1$  e  $y+1$ .

Por ejemplo, construyamos el siguiente arreglo-Z:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?


<sup>2</sup>El algoritmo-Z fue presentado en [32] como el método más simple conocido para la coincidencia de patrones en tiempo lineal, y la idea original fue atribuida a [50].

Después de calcular el valor  $z[6] = 5$ , el rango actual  $[x, y]$  es  $[6, 10]$ :

						$x$				$y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
—	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?	


Ahora podemos calcular los valores subsiguientes de la matriz  $Z$  de manera eficiente, porque sabemos que  $s[0 \dots 4]$  y  $s[6 \dots 10]$  son iguales. Primero, dado que  $z[1] = z[2] = 0$ , sabemos inmediatamente que también  $z[7] = z[8] = 0$ :

						$x$				$y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
-	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	



Luego, dado que  $z[3] = 2$ , sabemos que  $z[9] \geq 2$ :

						$x$				$y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	



Sin embargo, no tenemos información sobre la cadena después de la posición 10, por lo que necesitamos comparar las subcadenas caracter por caracter:

						$x$				$y$						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?	

Resulta que  $z[9] = 7$ , por lo que el nuevo rango  $[x, y]$  es  $[9, 15]$ :

									$x$				$y$		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?

Después de esto, todos los valores restantes de la matriz  $Z$  se pueden determinar utilizando la información ya almacenada en la matriz  $Z$ :

									$x$				$y$		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

## Usando el arreglo $Z$

A menudo es cuestión de gusto si usar hashing de cadenas o el algoritmo  $Z$ . A diferencia del hashing, el algoritmo  $Z$  siempre funciona y no hay riesgo de colisiones. Por otro lado, el algoritmo  $Z$  es más difícil de implementar y algunos problemas solo se pueden resolver usando hashing.

Como ejemplo, considere de nuevo el problema de coincidencia de patrones, donde nuestra tarea es encontrar las ocurrencias de un patrón  $p$  en una cadena  $s$ . Ya resolvimos este problema de manera eficiente usando hashing de cadenas, pero el algoritmo  $Z$  proporciona otra forma de resolver el problema.

Una idea habitual en el procesamiento de cadenas es construir una cadena que consista en varias cadenas separadas por caracteres especiales. En este problema, podemos construir una cadena  $p\#s$ , donde  $p$  y  $s$  están separados por un carácter especial  $\#$  que no ocurre en las cadenas. El arreglo  $Z$  de  $p\#s$  nos dice las posiciones donde  $p$  ocurre en  $s$ , porque esas posiciones contienen la longitud de  $p$ .

Por ejemplo, si  $s = \text{HATTIVATTI}$  y  $p = \text{ATT}$ , el arreglo  $Z$  es el siguiente:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	T	T	#	H	A	T	T	I	V	A	T	T	I
	–	0	0	0	0	3	0	0	0	0	3	0	0	0

Las posiciones 5 y 10 contienen el valor 3, lo que significa que el patrón  $\text{ATT}$  ocurre en las posiciones correspondientes de  $\text{HATTIVATTI}$ .

La complejidad temporal del algoritmo resultante es lineal, porque basta con construir el arreglo  $Z$  y recorrer sus valores.

## Implementación

Aquí hay una breve implementación del algoritmo  $Z$  que devuelve un vector que corresponde al arreglo  $Z$ .

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
```



```
        z[i] = max(0,min(z[i-x],y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```



# Chapter 27

## Algoritmos de raíz cuadrada

Un **algoritmo de raíz cuadrada** es un algoritmo que tiene una raíz cuadrada en su complejidad temporal. Una raíz cuadrada puede ser vista como un "logaritmo del pobre": la complejidad  $O(\sqrt{n})$  es mejor que  $O(n)$  pero peor que  $O(\log n)$ . En cualquier caso, muchos algoritmos de raíz cuadrada son rápidos y utilizables en la práctica.

Como ejemplo, consideremos el problema de crear una estructura de datos que soporte dos operaciones en un arreglo: modificar un elemento en una posición dada y calcular la suma de los elementos en el rango dado. Anteriormente hemos resuelto el problema utilizando árboles binarios indexados y segmentados, que soportan ambas operaciones en tiempo  $O(\log n)$ . Sin embargo, ahora resolveremos el problema de otra manera usando una estructura de raíz cuadrada que nos permite modificar elementos en tiempo  $O(1)$  y calcular sumas en tiempo  $O(\sqrt{n})$ .

La idea es dividir el arreglo en *bloques* de tamaño  $\sqrt{n}$  de manera que cada bloque contenga la suma de los elementos dentro del bloque. Por ejemplo, un arreglo de 16 elementos será dividido en bloques de 4 elementos como sigue:


21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

En esta estructura, es fácil modificar los elementos del arreglo, porque solo es necesario actualizar la suma de un solo bloque después de cada modificación, lo cual puede hacerse en tiempo  $O(1)$ . Por ejemplo, la siguiente imagen muestra cómo cambia el valor de un elemento y la suma del bloque correspondiente:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Luego, para calcular la suma de elementos en un rango, dividimos el rango en tres partes de manera que la suma consista en valores de elementos individuales y sumas de bloques entre ellos:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2



Dado que el número de elementos individuales es  $O(\sqrt{n})$  y el número de bloques también es  $O(\sqrt{n})$ , la consulta de suma toma tiempo  $O(\sqrt{n})$ . El propósito del tamaño del bloque  $\sqrt{n}$  es que *balancea* dos cosas: el arreglo se divide en  $\sqrt{n}$  bloques, cada uno de los cuales contiene  $\sqrt{n}$  elementos.

En la práctica, no es necesario usar el valor exacto de  $\sqrt{n}$  como parámetro, y en su lugar podemos usar los parámetros  $k$  y  $n/k$  donde  $k$  es diferente de  $\sqrt{n}$ . El parámetro óptimo depende del problema y de la entrada. Por ejemplo, si un algoritmo a menudo recorre los bloques pero rara vez inspecciona elementos individuales dentro de los bloques, puede ser una buena idea dividir el arreglo en  $k < \sqrt{n}$  bloques, cada uno de los cuales contiene  $n/k > \sqrt{n}$  elementos.

## 27.1 Combinación de algoritmos

En esta sección discutimos dos algoritmos de raíz cuadrada que se basan en combinar dos algoritmos en uno solo. En ambos casos, podríamos usar cualquiera de los algoritmos sin el otro y resolver el problema en tiempo  $O(n^2)$ . Sin embargo, al combinar los algoritmos, el tiempo de ejecución es solo  $O(n\sqrt{n})$ .

### Procesamiento de casos

Supongamos que tenemos una cuadrícula bidimensional que contiene  $n$  celdas. A cada celda se le asigna una letra, y nuestra tarea es encontrar dos celdas con la misma letra cuya distancia sea mínima, donde la distancia entre las celdas  $(x_1, y_1)$  y  $(x_2, y_2)$  es  $|x_1 - x_2| + |y_1 - y_2|$ . Por ejemplo, consideremos la siguiente cuadrícula:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

En este caso, la distancia mínima es 2 entre las dos letras 'E'.

Podemos resolver el problema considerando cada letra por separado. Con este enfoque, el nuevo problema es calcular la distancia mínima entre dos celdas con una letra *fija*  $c$ . Nos enfocamos en dos algoritmos para esto:

*Algoritmo 1:* Recorrer todas las parejas de celdas con la letra  $c$ , y calcular la distancia mínima entre tales celdas. Esto tomará tiempo  $O(k^2)$  donde  $k$  es el número de celdas con la letra  $c$ .

*Algoritmo 2:* Realizar una búsqueda en anchura que simultáneamente comienza en cada celda con la letra  $c$ . La distancia mínima entre dos celdas con la letra  $c$  se calculará en tiempo  $O(n)$ .

Una forma de resolver el problema es elegir uno de los algoritmos y usarlo para todas las letras. Si usamos el Algoritmo 1, el tiempo de ejecución es  $O(n^2)$ , porque todas las celdas pueden contener la misma letra, y en este caso  $k = n$ .

También si usamos el Algoritmo 2, el tiempo de ejecución es  $O(n^2)$ , porque todas las celdas pueden tener letras diferentes, y en este caso se necesitan  $n$  búsquedas.

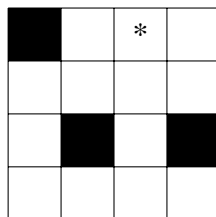
Sin embargo, podemos *combinar* los dos algoritmos y usar diferentes algoritmos para diferentes letras dependiendo de cuántas veces aparezca cada letra en la cuadrícula. Supongamos que una letra  $c$  aparece  $k$  veces. Si  $k \leq \sqrt{n}$ , usamos el Algoritmo 1, y si  $k > \sqrt{n}$ , usamos el Algoritmo 2. Resulta que al hacer esto, el tiempo total de ejecución del algoritmo es solo  $O(n\sqrt{n})$ .

Primero, supongamos que usamos el Algoritmo 1 para una letra  $c$ . Dado que  $c$  aparece como máximo  $\sqrt{n}$  veces en la cuadrícula, comparamos cada celda con la letra  $c$   $O(\sqrt{n})$  veces con otras celdas. Por lo tanto, el tiempo utilizado para procesar todas esas celdas es  $O(n\sqrt{n})$ . Luego, supongamos que usamos el Algoritmo 2 para una letra  $c$ . Hay como máximo  $\sqrt{n}$  letras de este tipo, por lo que procesar esas letras también lleva tiempo  $O(n\sqrt{n})$ .

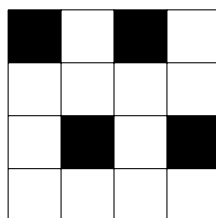
## Procesamiento por lotes

Nuestro próximo problema también trata una cuadrícula bidimensional que contiene  $n$  celdas. Inicialmente, cada celda excepto una es blanca. Realizamos  $n - 1$  operaciones, cada una de las cuales primero calcula la distancia mínima desde una celda blanca dada hasta una celda negra, y luego pinta la celda blanca de negro.

Por ejemplo, consideremos la siguiente operación:



Primero calculamos la distancia mínima desde la celda blanca marcada con \* hasta una celda negra. La distancia mínima es 2, porque podemos movernos dos pasos a la izquierda hacia una celda negra. Luego, pintamos la celda blanca de negro:



Consideremos los siguientes dos algoritmos:

*Algoritmo 1:* Usar búsqueda en anchura para calcular para cada celda blanca la distancia a la celda negra más cercana. Esto toma  $O(n)$  tiempo, y después de la búsqueda, podemos encontrar la distancia mínima desde cualquier celda blanca a una celda negra en tiempo  $O(1)$ .

*Algoritmo 2:* Mantener una lista de celdas que han sido pintadas de negro, recorrer esta lista en cada operación y luego agregar una nueva celda a la lista. Una operación toma  $O(k)$  tiempo donde  $k$  es la longitud de la lista.

Combinamos los algoritmos anteriores dividiendo las operaciones en  $O(\sqrt{n})$  lotes, cada uno de los cuales consiste en  $O(\sqrt{n})$  operaciones. Al comienzo de cada lote, realizamos el Algoritmo 1. Luego, usamos el Algoritmo 2 para procesar las operaciones en el lote. Limpiamos la lista del Algoritmo 2 entre los lotes. En cada operación, la distancia mínima a una celda negra es la distancia calculada por el Algoritmo 1 o la distancia calculada por el Algoritmo 2.

El algoritmo resultante funciona en tiempo  $O(n\sqrt{n})$ . Primero, el Algoritmo 1 se realiza  $O(\sqrt{n})$  veces, y cada búsqueda funciona en tiempo  $O(n)$ . Segundo, al usar el Algoritmo 2 en un lote, la lista contiene  $O(\sqrt{n})$  celdas (porque limpiamos la lista entre lotes) y cada operación toma tiempo  $O(\sqrt{n})$ .

## 27.2 Particiones de enteros

Algunos algoritmos de raíz cuadrada se basan en la siguiente observación: si un entero positivo  $n$  se representa como una suma de enteros positivos, tal suma siempre contiene a lo sumo  $O(\sqrt{n})$  números *distintos*. La razón de esto es que para construir una suma que contenga el máximo número de números distintos, debemos elegir números *pequeños*. Si elegimos los números  $1, 2, \dots, k$ , la suma resultante es

$$\frac{k(k+1)}{2}.$$

Así, la cantidad máxima de números distintos es  $k = O(\sqrt{n})$ . A continuación discutiremos dos problemas que pueden resolverse eficientemente usando esta observación.

### Mochila (Knapsack)

Supongamos que tenemos una lista de pesos enteros cuya suma es  $n$ . Nuestra tarea es encontrar todas las sumas que pueden formarse usando un subconjunto de los pesos. Por ejemplo, si los pesos son  $\{1, 3, 3\}$ , las sumas posibles son las siguientes:

- 0 (conjunto vacío)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Usando el enfoque estándar de la mochila (ver Capítulo 7.4), el problema puede resolverse de la siguiente manera: definimos una función  $\text{possible}(x, k)$  cuyo valor es 1 si la suma  $x$  puede formarse usando los primeros  $k$  pesos, y 0 en caso contrario. Dado que la suma de los pesos es  $n$ , hay a lo sumo  $n$  pesos

y todos los valores de la función pueden calcularse en tiempo  $O(n^2)$  utilizando programación dinámica.

Sin embargo, podemos hacer el algoritmo más eficiente usando el hecho de que hay a lo sumo  $O(\sqrt{n})$  pesos *distintos*. Así, podemos procesar los pesos en grupos que consisten en pesos similares. Podemos procesar cada grupo en tiempo  $O(n)$ , lo que da lugar a un algoritmo de tiempo  $O(n\sqrt{n})$ .

La idea es usar un arreglo que registre las sumas de pesos que pueden formarse usando los grupos procesados hasta ahora. El arreglo contiene  $n$  elementos: el elemento  $k$  es 1 si la suma  $k$  puede formarse y 0 en caso contrario. Para procesar un grupo de pesos, recorreremos el arreglo de izquierda a derecha y registramos las nuevas sumas de pesos que pueden formarse usando este grupo y los grupos anteriores.

## Construcción de cadenas

Dada una cadena  $s$  de longitud  $n$  y un conjunto de cadenas  $D$  cuya longitud total es  $m$ , consideremos el problema de contar la cantidad de formas en que  $s$  puede formarse como una concatenación de cadenas en  $D$ . Por ejemplo, si  $s = ABAB$  y  $D = \{A, B, AB\}$ , hay 4 formas:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Podemos resolver el problema utilizando programación dinámica: Sea  $\text{count}(k)$  el número de formas de construir el prefijo  $s[0 \dots k]$  usando las cadenas en  $D$ . Entonces  $\text{count}(n-1)$  da la respuesta al problema, y podemos resolver el problema en tiempo  $O(n^2)$  usando una estructura de trie.

Sin embargo, podemos resolver el problema de manera más eficiente usando el hashing de cadenas y el hecho de que hay a lo sumo  $O(\sqrt{m})$  longitudes de cadena distintas en  $D$ . Primero, construimos un conjunto  $H$  que contiene todos los valores hash de las cadenas en  $D$ . Luego, al calcular un valor de  $\text{count}(k)$ , recorreremos todos los valores de  $p$  tales que hay una cadena de longitud  $p$  en  $D$ , calculamos el valor hash de  $s[k-p+1 \dots k]$  y verificamos si pertenece a  $H$ . Dado que hay a lo sumo  $O(\sqrt{m})$  longitudes de cadena distintas, esto resulta en un algoritmo cuyo tiempo de ejecución es  $O(n\sqrt{m})$ .

## 27.3 Algoritmo de Mo

**El algoritmo de Mo**<sup>1</sup> se puede usar en muchos problemas que requieren procesar consultas de rango en una matriz *estática*, es decir, los valores de la matriz no cambian entre las consultas. En cada consulta, se nos da un rango  $[a, b]$ , y debemos calcular un valor basado en los elementos de la matriz entre las

<sup>1</sup>Según [12], este algoritmo lleva el nombre de Mo Tao, un programador competitivo chino, pero la técnica ha aparecido anteriormente en la literatura [44].

posiciones  $a$  y  $b$ . Dado que la matriz es estática, las consultas pueden procesarse en cualquier orden, y el algoritmo de Mo procesa las consultas en un orden especial que garantiza que el algoritmo funcione eficientemente.

El algoritmo de Mo mantiene un *rango activo* de la matriz, y la respuesta a una consulta sobre el rango activo se conoce en cada momento. El algoritmo procesa las consultas una por una, y siempre mueve los extremos del rango activo insertando y eliminando elementos. La complejidad temporal del algoritmo es  $O(n\sqrt{n}f(n))$  donde la matriz contiene  $n$  elementos, hay  $n$  consultas y cada inserción y eliminación de un elemento toma tiempo  $O(f(n))$ .

El truco en el algoritmo de Mo es el orden en que se procesan las consultas: La matriz se divide en bloques de  $k = O(\sqrt{n})$  elementos, y una consulta  $[a_1, b_1]$  se procesa antes que una consulta  $[a_2, b_2]$  si cualquiera de las siguientes condiciones se cumple:

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$  o
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$  y  $b_1 < b_2$ .

Así, todas las consultas cuyos extremos izquierdos están en un cierto bloque se procesan una tras otra ordenadas según sus extremos derechos. Usando este orden, el algoritmo realiza solo  $O(n\sqrt{n})$  operaciones, porque el extremo izquierdo se mueve  $O(n)$  veces  $O(\sqrt{n})$  pasos, y el extremo derecho se mueve  $O(\sqrt{n})$  veces  $O(n)$  pasos. Por lo tanto, ambos extremos se mueven un total de  $O(n\sqrt{n})$  pasos durante el algoritmo.

## Ejemplo

Como ejemplo, consideremos un problema donde se nos dan un conjunto de consultas, cada una correspondiente a un rango en un arreglo, y nuestra tarea es calcular para cada consulta el número de elementos *distintos* en el rango.

En el algoritmo de Mo, las consultas siempre se ordenan de la misma manera, pero depende del problema cómo se mantiene la respuesta a la consulta. En este problema, podemos mantener un arreglo `count` donde `count[x]` indica cuántas veces aparece el elemento  $x$  en el rango activo.

Cuando pasamos de una consulta a otra, el rango activo cambia. Por ejemplo, si el rango actual es

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

y el siguiente rango es

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

habrá tres pasos: el extremo izquierdo se mueve un paso a la derecha, y el extremo derecho se mueve dos pasos a la derecha.

Después de cada paso, el arreglo `count` necesita ser actualizado. Después de agregar un elemento  $x$ , aumentamos el valor de `count[x]` en 1, y si `count[x] = 1` después de esto, también aumentamos la respuesta a la consulta en 1. De manera



similar, después de eliminar un elemento  $x$ , disminuimos el valor de  $\text{count}[x]$  en 1, y si  $\text{count}[x] = 0$  después de esto, también disminuimos la respuesta a la consulta en 1.

En este problema, el tiempo necesario para realizar cada paso es  $O(1)$ , por lo que la complejidad temporal total del algoritmo es  $O(n\sqrt{n})$ .



## Chapter 28

# Árboles de segmentos revisitados

Un árbol de segmentos es una estructura de datos versátil que se puede utilizar para resolver una gran cantidad de problemas de algoritmos. Sin embargo, hay muchos temas relacionados con los árboles de segmentos que aún no hemos tocado. Ahora es el momento de discutir algunas variantes más avanzadas de los árboles de segmentos.

Hasta ahora, hemos implementado las operaciones de un árbol de segmentos caminando *de abajo hacia arriba* en el árbol. Por ejemplo, hemos calculado sumas de rango de la siguiente manera (Capítulo 9.3):

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Sin embargo, en los árboles de segmentos más avanzados, a menudo es necesario implementar las operaciones de otra manera, *de arriba hacia abajo*. Usando este enfoque, la función se convierte en la siguiente:

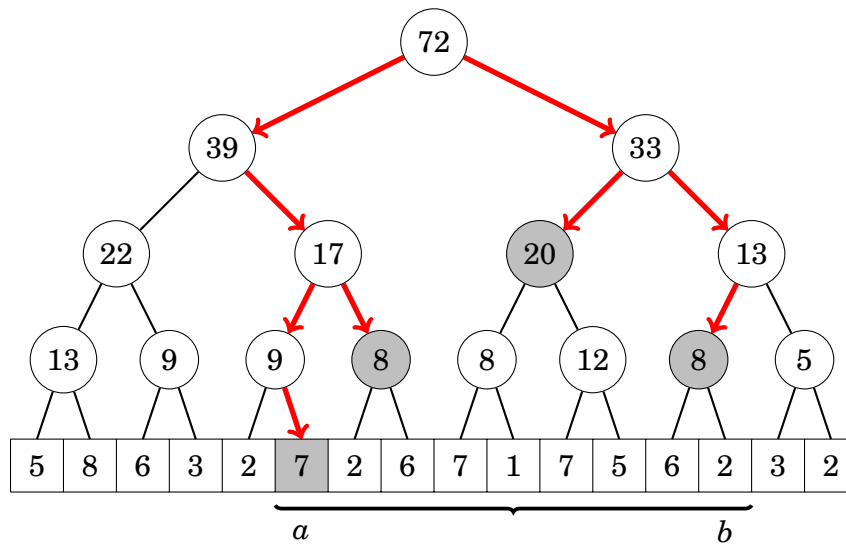
```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Ahora podemos calcular cualquier valor de  $\text{sum}_q(a, b)$  (la suma de los valores del arreglo en el rango  $[a, b]$ ) de la siguiente manera:

```
int s = sum(a, b, 1, 0, n-1);
```

El parámetro  $k$  indica la posición actual en tree. Inicialmente  $k$  es igual a 1, porque comenzamos en la raíz del árbol. El rango  $[x, y]$  corresponde a  $k$  e inicialmente es  $[0, n - 1]$ . Al calcular la suma, si  $[x, y]$  está fuera de  $[a, b]$ , la suma es 0, y si  $[x, y]$  está completamente dentro de  $[a, b]$ , la suma se puede encontrar en tree. Si  $[x, y]$  está parcialmente dentro de  $[a, b]$ , la búsqueda continúa recursivamente a la mitad izquierda y derecha de  $[x, y]$ . La mitad izquierda es  $[x, d]$  y la mitad derecha es  $[d + 1, y]$  donde  $d = \lfloor \frac{x+y}{2} \rfloor$ .

La siguiente imagen muestra cómo procede la búsqueda al calcular el valor de  $\text{sum}_q(a, b)$ . Los nodos grises indican nodos donde la recursión se detiene y la suma se puede encontrar en tree.



También en esta implementación, las operaciones toman  $O(\log n)$  tiempo, porque el número total de nodos visitados es  $O(\log n)$ .

## 28.1 Propagación perezosa

Utilizando la **propagación perezosa**, podemos construir un árbol de segmentos que admite *ambas* actualizaciones de rango y consultas de rango en  $O(\log n)$  tiempo. La idea es realizar actualizaciones y consultas de arriba hacia abajo y realizar actualizaciones *perezosamente* para que se propaguen hacia abajo del árbol solo cuando sea necesario.

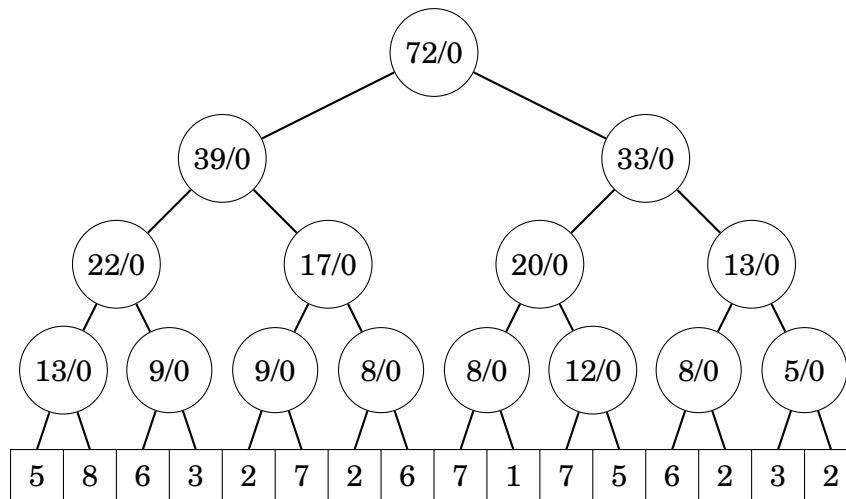
En un árbol de segmentos perezoso, los nodos contienen dos tipos de información. Como en un árbol de segmentos ordinario, cada nodo contiene la suma o algún otro valor relacionado con la submatriz correspondiente. Además, el nodo puede contener información relacionada con actualizaciones perezosas, que no ha sido propagada a sus hijos.

Hay dos tipos de actualizaciones de rango: cada valor de la matriz en el rango se *aumenta* en algún valor o se le *asigna* algún valor. Ambas operaciones se pueden implementar utilizando ideas similares, e incluso es posible construir un árbol que admita ambas operaciones al mismo tiempo.

## Árboles de segmentos perezosos

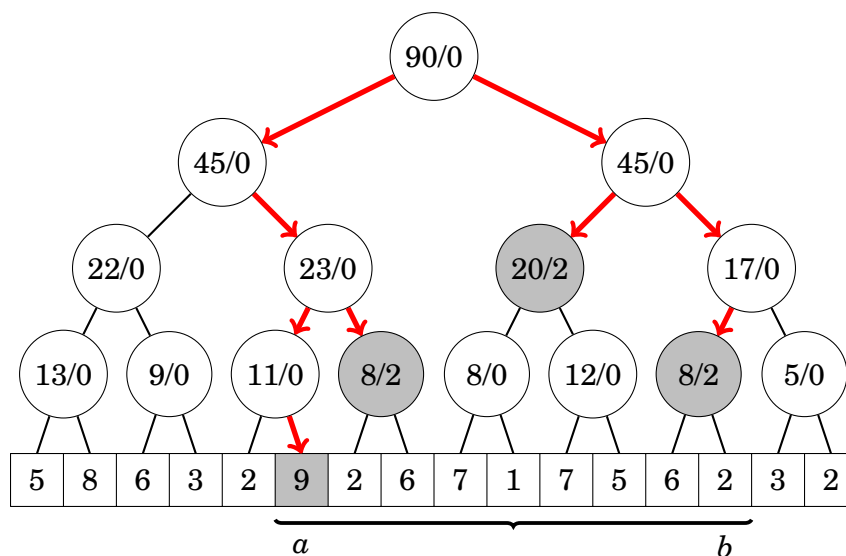
Consideremos un ejemplo donde nuestro objetivo es construir un árbol de segmentos que admita dos operaciones: aumentar cada valor en  $[a, b]$  en una constante y calcular la suma de valores en  $[a, b]$ .

Construiremos un árbol donde cada nodo tiene dos valores  $s/z$ :  $s$  denota la suma de valores en el rango, y  $z$  denota el valor de una actualización perezosa, lo que significa que todos los valores en el rango deben aumentarse en  $z$ . En el siguiente árbol,  $z = 0$  en todos los nodos, por lo que no hay actualizaciones perezosas en curso.



Cuando los elementos en  $[a, b]$  se incrementan por  $u$ , caminamos desde la raíz hacia las hojas y modificamos los nodos del árbol de la siguiente manera: Si el rango  $[x, y]$  de un nodo es completamente dentro de  $[a, b]$ , incrementamos el valor  $z$  del nodo por  $u$  y detenemos. Si  $[x, y]$  solo pertenece parcialmente a  $[a, b]$ , incrementamos el valor  $s$  del nodo por  $hu$ , donde  $h$  es el tamaño de la intersección de  $[a, b]$  y  $[x, y]$ , y continuamos nuestra caminata recursivamente en el árbol.

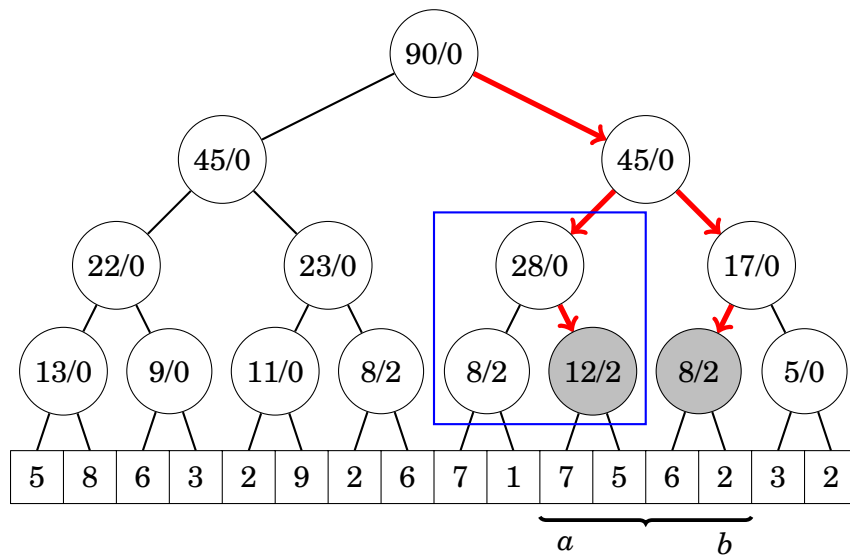
Por ejemplo, la siguiente imagen muestra el árbol después de incrementar los elementos en  $[a, b]$  por 2:



También calculamos la suma de elementos en un rango  $[a, b]$  caminando en el árbol de arriba hacia abajo. Si el rango  $[x, y]$  de un nodo pertenece completamente a  $[a, b]$ , sumamos el valor  $s$  del nodo a la suma. De lo contrario, continuamos la búsqueda recursivamente hacia abajo en el árbol.

Tanto en actualizaciones como en consultas, el valor de una actualización perezosa siempre se propaga a los hijos del nodo antes de procesar el nodo. La idea es que las actualizaciones se propaguen hacia abajo solo cuando sea necesario, lo que garantiza que las operaciones sean siempre eficientes.

La siguiente imagen muestra cómo cambia el árbol cuando calculamos el valor de  $\text{sum}_a(a, b)$ . El rectángulo muestra los nodos cuyos valores cambian, porque una actualización perezosa se propaga hacia abajo.



Tenga en cuenta que a veces es necesario combinar actualizaciones perezosas. Esto sucede cuando un nodo que ya tiene una actualización perezosa se le asigna otra actualización perezosa. Al calcular sumas, es fácil combinar actualizaciones perezosas, porque la combinación de actualizaciones  $z_1$  y  $z_2$  corresponde a una actualización  $z_1 + z_2$ .

## Actualizaciones polinómicas

Las actualizaciones perezosas se pueden generalizar para que sea posible actualizar rangos utilizando polinomios de la forma

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

En este caso, la actualización para un valor en la posición  $i$  en  $[a, b]$  es  $p(i - a)$ . Por ejemplo, agregar el polinomio  $p(u) = u + 1$  a  $[a, b]$  significa que el valor en la posición  $a$  aumenta en 1, el valor en la posición  $a + 1$  aumenta en 2, y así sucesivamente.

Para soportar actualizaciones polinómicas, a cada nodo se le asignan  $k + 2$  valores, donde  $k$  es igual al grado del polinomio. El valor  $s$  es la suma de

los elementos en el rango, y los valores  $z_0, z_1, \dots, z_k$  son los coeficientes de un polinomio que corresponde a una actualización perezosa.

Ahora, la suma de valores en un rango  $[x, y]$  es igual a

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

El valor de dicha suma se puede calcular eficientemente utilizando fórmulas de suma. Por ejemplo, el término  $z_0$  corresponde a la suma  $(y-x+1)z_0$ , y el término  $z_1 u$  corresponde a la suma

$$z_1(0+1+\dots+y-x) = z_1 \frac{(y-x)(y-x+1)}{2}.$$

Cuando se propaga una actualización en el árbol, los índices de  $p(u)$  cambian, porque en cada rango  $[x, y]$ , los valores son calculados para  $u = 0, 1, \dots, y-x$ . Sin embargo, esto no es un problema, porque  $p'(u) = p(u+h)$  es un polinomio del mismo grado que  $p(u)$ . Por ejemplo, si  $p(u) = t_2 u^2 + t_1 u - t_0$ , entonces

$$p'(u) = t_2(u+h)^2 + t_1(u+h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

## 28.2 Árboles dinámicos

Un árbol de segmentos ordinario es estático, lo que significa que cada nodo tiene una posición fija en la matriz y el árbol requiere una cantidad fija de memoria. En un **árbol de segmentos dinámico**, la memoria se asigna solo para los nodos que se acceden realmente durante el algoritmo, lo que puede ahorrar una gran cantidad de memoria.

Los nodos de un árbol dinámico se pueden representar como estructuras:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Aquí `value` es el valor del nodo, `[x,y]` es el rango correspondiente, y `left` y `right` apuntan al subárbol izquierdo y derecho.

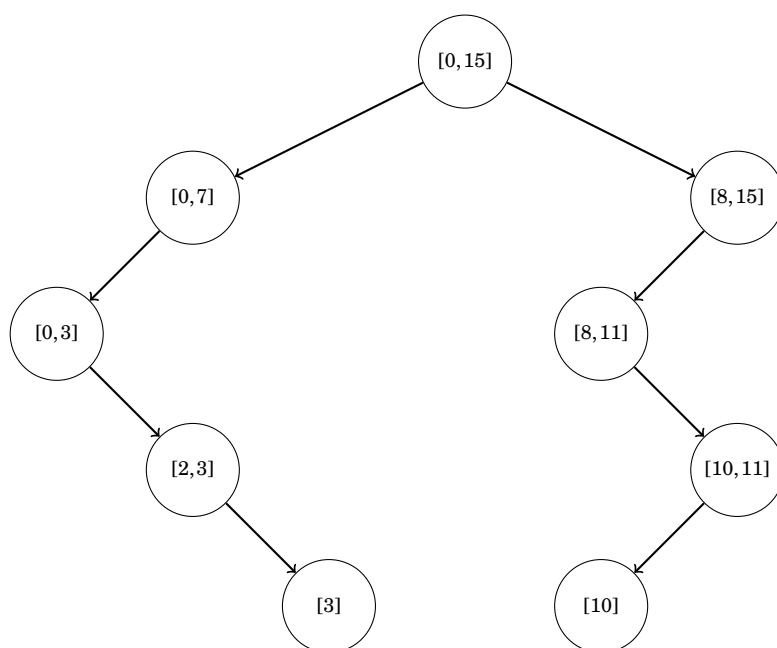
Después de esto, los nodos se pueden crear de la siguiente manera:

```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

## Árboles de segmentos dispersos

Un árbol de segmentos dinámico es útil cuando la matriz subyacente es *dispersa*, es decir, el rango  $[0, n - 1]$  de índices permitidos es grande, pero la mayoría de los valores de la matriz son ceros. Mientras que un árbol de segmentos ordinario usa  $O(n)$  memoria, un árbol de segmentos dinámico solo usa  $O(k \log n)$  memoria, donde  $k$  es el número de operaciones realizadas.

Un **árbol de segmentos disperso** inicialmente tiene solo un nodo  $[0, n - 1]$  cuyo valor es cero, lo que significa que cada valor de la matriz es cero. Después de las actualizaciones, se agregan dinámicamente nuevos nodos al árbol. Por ejemplo, si  $n = 16$  y los elementos en las posiciones 3 y 10 se han modificado, el árbol contiene los siguientes nodos:



Cualquier camino desde el nodo raíz hasta una hoja contiene  $O(\log n)$  nodos, por lo que cada operación agrega como máximo  $O(\log n)$  nuevos nodos al árbol. Por lo tanto, después de  $k$  operaciones, el árbol contiene como máximo  $O(k \log n)$  nodos.

Tenga en cuenta que si sabemos todos los elementos que se actualizarán al comienzo del algoritmo, no es necesario un árbol de segmentos dinámico, porque podemos usar un árbol de segmentos ordinario con compresión de índice (Capítulo 9.4). Sin embargo, esto no es posible cuando los índices se generan durante el algoritmo.

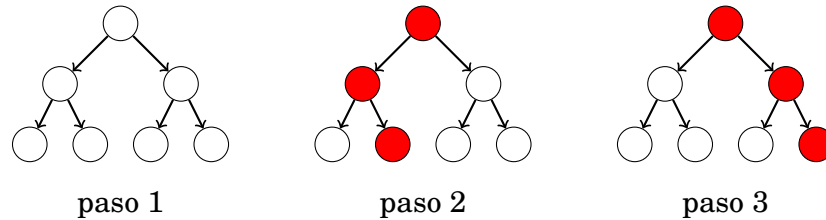
## Árboles de segmentos persistentes

Usando una implementación dinámica, también es posible crear un **árbol de segmentos persistente** que almacena el *historial de modificaciones* del árbol. En tal implementación, podemos acceder eficientemente a todas las versiones del árbol que han existido durante el algoritmo.

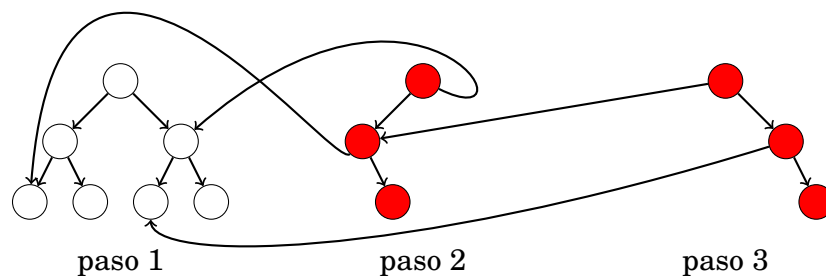


Cuando el historial de modificaciones está disponible, podemos realizar consultas en cualquier árbol anterior como en un árbol de segmentos ordinario, porque el estructura completa de cada árbol se almacena. También podemos crear nuevos árboles basados en anteriores árboles y modificarlos independientemente.

Considere la siguiente secuencia de actualizaciones, donde los nodos rojos cambian y otros nodos permanecen iguales:



Después de cada actualización, la mayoría de los nodos del árbol permanecen iguales, por lo que una forma eficiente de almacenar el historial de modificaciones es representar cada árbol en el historial como una combinación de nuevos nodos y subárboles de árboles anteriores. En este ejemplo, el historial de modificaciones puede ser almacenado de la siguiente manera:



La estructura de cada árbol anterior puede ser reconstruida siguiendo los punteros comenzando en el nodo raíz correspondiente. Dado que cada operación agrega solo  $O(\log n)$  nuevos nodos al árbol, es posible almacenar el historial completo de modificaciones del árbol.

## 28.3 Estructuras de datos

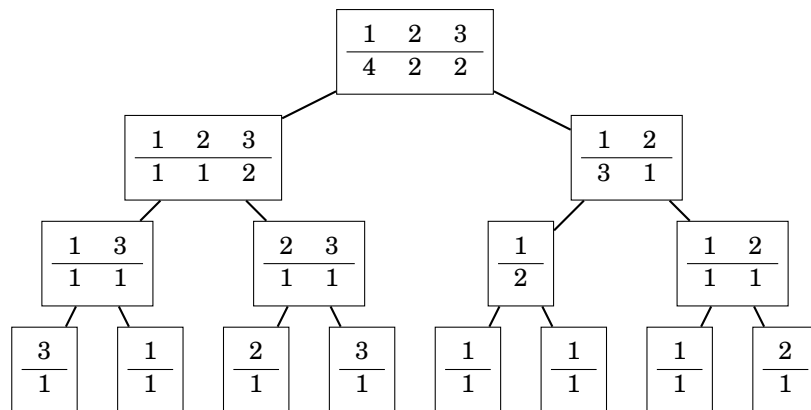
En lugar de valores únicos, los nodos en un árbol de segmentos también pueden contener *estructuras de datos* que mantienen información sobre los rangos correspondientes. En tal árbol, las operaciones toman  $O(f(n)\log n)$  tiempo, donde  $f(n)$  es el tiempo necesario para procesar un solo nodo durante una operación.

Como ejemplo, considere un árbol de segmentos que soporta consultas de la forma "¿cuántas veces aparece un elemento  $x$  en el rango  $[a, b]$ ?" Por ejemplo, el elemento 1 aparece tres veces en el siguiente rango:

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

Para soportar tales consultas, construimos un árbol de segmentos donde a cada nodo se le asigna una estructura de datos que se puede preguntar cuántas veces cualquier elemento  $x$  aparece en el rango correspondiente. Usando este árbol, la respuesta a una consulta se puede calcular combinando los resultados de los nodos que pertenecen al rango.

Por ejemplo, el siguiente árbol de segmentos corresponde a la matriz anterior:



Podemos construir el árbol de modo que cada nodo contenga una estructura map. En este caso, el tiempo necesario para procesar cada nodo es  $O(\log n)$ , por lo que la complejidad temporal total de una consulta es  $O(\log^2 n)$ . El árbol utiliza  $O(n \log n)$  memoria, porque hay  $O(\log n)$  niveles y cada nivel contiene  $O(n)$  elementos.

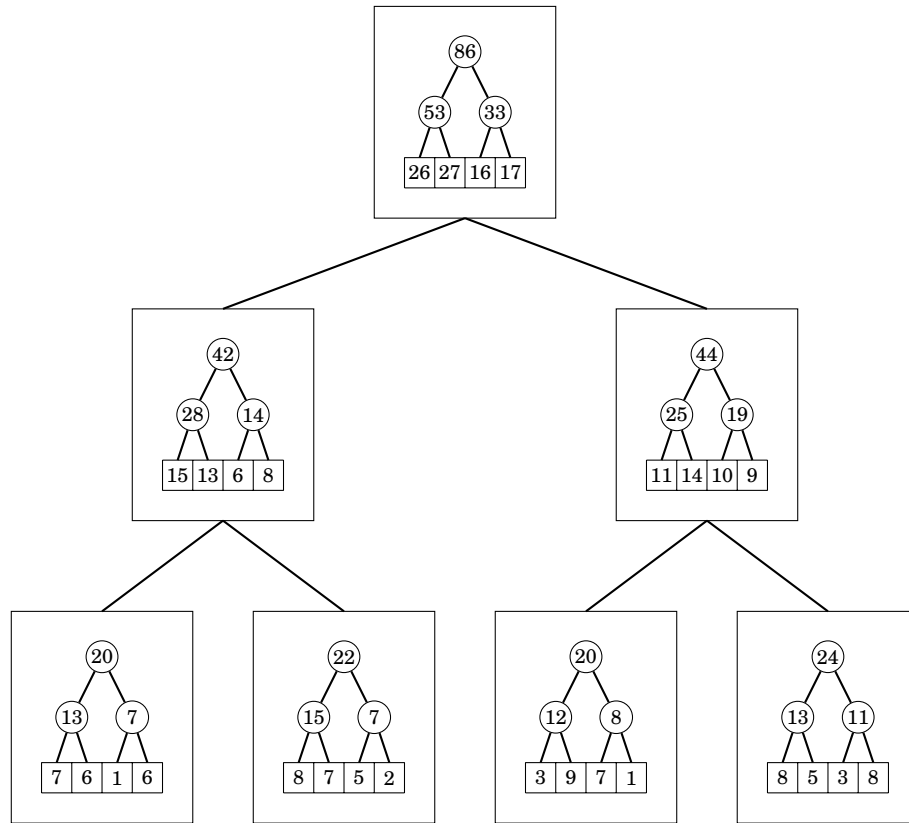
## 28.4 Bidimensionalidad

Un **árbol de segmentos bidimensional** admite consultas relacionadas con subarreglos rectangulares de una matriz bidimensional. Dicho árbol se puede implementar como árboles de segmentos anidados: un árbol grande corresponde a las filas de la matriz, y cada nodo contiene un árbol pequeño que corresponde a una columna.

Por ejemplo, en la matriz

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

la suma de cualquier submatriz se puede calcular a partir del siguiente árbol de segmentos:



Las operaciones de un árbol de segmentos bidimensional toman  $O(\log^2 n)$  tiempo, porque el árbol grande y cada árbol pequeño consisten en  $O(\log n)$  niveles. El árbol requiere  $O(n^2)$  memoria, porque cada árbol pequeño contiene  $O(n)$  valores.

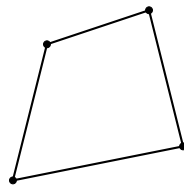


# Chapter 29

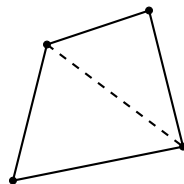
## Geometría

En problemas geométricos, a menudo es difícil encontrar una manera de abordar el problema de modo que la solución al problema pueda implementarse convenientemente y el número de casos especiales sea pequeño.

Como ejemplo, considere un problema donde se nos dan los vértices de un cuadrilátero (un polígono que tiene cuatro vértices), y nuestra tarea es calcular su área. Por ejemplo, una posible entrada para el problema es la siguiente:



Una forma de abordar el problema es dividir el cuadrilátero en dos triángulos mediante una recta entre dos vértices opuestos:

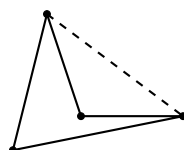


Después de esto, basta con sumar las áreas de los triángulos. El área de un triángulo se puede calcular, por ejemplo, usando la **fórmula de Herón**

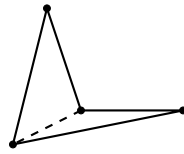
$$\sqrt{s(s-a)(s-b)(s-c)},$$

donde  $a$ ,  $b$  y  $c$  son las longitudes de los lados del triángulo y  $s = (a + b + c)/2$ .

Esta es una forma posible de resolver el problema, pero hay un escollo: ¿cómo dividir el cuadrilátero en triángulos? Resulta que a veces no podemos simplemente elegir dos vértices opuestos arbitrarios. Por ejemplo, en la siguiente situación, la línea de división está *fuera* del cuadrilátero:



Sin embargo, otra forma de dibujar la línea funciona:



Es claro para un humano cuál de las líneas es la correcta elección, pero la situación es difícil para una computadora.

Sin embargo, resulta que podemos resolver el problema usando otro método que es más conveniente para un programador. Es decir, existe una fórmula general

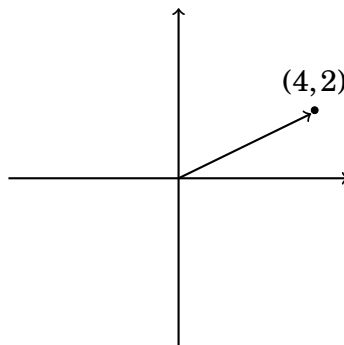
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

que calcula el área de un cuadrilátero cuyos vértices son  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  y  $(x_4, y_4)$ . Esta fórmula es fácil de implementar, no hay casos especiales y podemos incluso generalizar la fórmula a *todos* los polígonos.

## 29.1 Números complejos

Un **número complejo** es un número de la forma  $x + yi$ , donde  $i = \sqrt{-1}$  es la **unidad imaginaria**. Una interpretación geométrica de un número complejo es que representa un punto bidimensional  $(x, y)$  o un vector desde el origen hasta un punto  $(x, y)$ .

Por ejemplo,  $4 + 2i$  corresponde a el siguiente punto y vector:



La clase de número complejo de C++ `complex` es útil para resolver problemas geométricos. Usando la clase podemos representar puntos y vectores como números complejos, y la clase contiene herramientas que son útiles en geometría.

En el siguiente código, `C` es el tipo de una coordenada y `P` es el tipo de un punto o un vector. Además, el código define las macros `X` y `Y` que se pueden usar para referirse a las coordenadas `x` e `y`.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

Por ejemplo, el siguiente código define un punto  $p = (4,2)$  y imprime sus coordenadas  $x$  e  $y$ :

```
P p = {4,2};  
cout << p.X << " " << p.Y << "\n"; // 4 2
```

El siguiente código define vectores  $v = (3,1)$  y  $u = (2,2)$ , y después calcula la suma  $s = v + u$ .

```
P v = {3,1};  
P u = {2,2};  
P s = v+u;  
cout << s.X << " " << s.Y << "\n"; // 5 3
```

En la práctica, un tipo de coordenada adecuado es usualmente `long long` (entero) o `long double` (número real). Es una buena idea usar enteros siempre que sea posible, porque los cálculos con enteros son exactos. Si se necesitan números reales, se deben tener en cuenta los errores de precisión al comparar números. Una forma segura de comprobar si los números reales  $a$  y  $b$  son iguales es compararlos usando  $|a - b| < \epsilon$ , donde  $\epsilon$  es un número pequeño (por ejemplo,  $\epsilon = 10^{-9}$ ).

## Funciones

En los siguientes ejemplos, el tipo de coordenada es `long double`.

La función `abs(v)` calcula la longitud  $|v|$  de un vector  $v = (x, y)$  usando la fórmula  $\sqrt{x^2 + y^2}$ . La función también se puede usar para calcular la distancia entre puntos  $(x_1, y_1)$  y  $(x_2, y_2)$ , porque esa distancia es igual a la longitud del vector  $(x_2 - x_1, y_2 - y_1)$ .

El siguiente código calcula la distancia entre los puntos  $(4,2)$  y  $(3,-1)$ :

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.16228
```

La función `arg(v)` calcula el ángulo de un vector  $v = (x, y)$  con respecto al eje  $x$ . La función da el ángulo en radianes, donde  $r$  radianes es igual a  $180r/\pi$  grados. El ángulo de un vector que apunta hacia la derecha es 0, y los ángulos disminuyen en el sentido de las agujas del reloj y aumentan en sentido contrario a las agujas del reloj.

La función `polar(s, a)` construye un vector cuya longitud es  $s$  y que apunta a un ángulo  $a$ . Un vector se puede rotar por un ángulo  $a$  multiplicándolo por un vector con longitud 1 y ángulo  $a$ .

El siguiente código calcula el ángulo de el vector  $(4,2)$ , lo rota  $1/2$  radianes en sentido contrario a las agujas del reloj, y luego calcula el ángulo nuevamente:

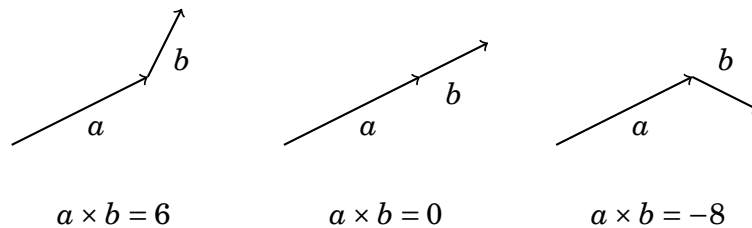
```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648
```

```
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; // 0.963648
```

## 29.2 Puntos y líneas

El **producto cruz**  $a \times b$  de vectores  $a = (x_1, y_1)$  y  $b = (x_2, y_2)$  se calcula usando la fórmula  $x_1y_2 - x_2y_1$ . El producto cruz nos dice si  $b$  gira a la izquierda (valor positivo), no gira (cero) o gira a la derecha (valor negativo) cuando se coloca directamente después de  $a$ .

La siguiente imagen ilustra los casos anteriores:



Por ejemplo, en el primer caso  $a = (4, 2)$  y  $b = (1, 2)$ . El siguiente código calcula el producto cruz usando la clase complex:

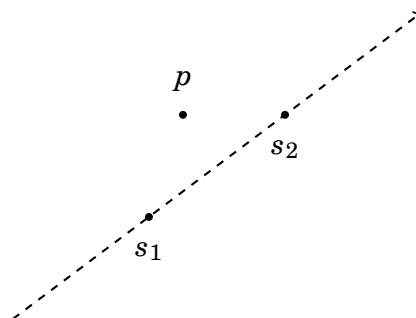
```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

El código anterior funciona, porque la función `conj` niega la coordenada y de un vector, y cuando los vectores  $(x_1, -y_1)$  y  $(x_2, y_2)$  se multiplican, la coordenada y del resultado es  $x_1y_2 - x_2y_1$ .

### Ubicación del punto

Los productos cruzados se pueden usar para probar si un punto está ubicado a la izquierda o derecha de una línea. Asumimos que la línea pasa por los puntos  $s_1$  y  $s_2$ , estamos mirando desde  $s_1$  a  $s_2$  y el punto es  $p$ .

Por ejemplo, en la siguiente imagen,  $p$  está a la izquierda de la línea:



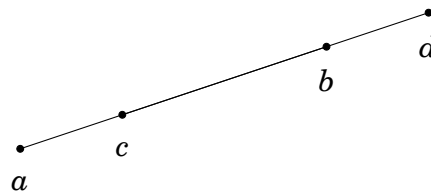
El producto cruz  $(p - s_1) \times (p - s_2)$  nos dice la ubicación del punto  $p$ . Si el producto cruz es positivo,  $p$  está ubicado a la izquierda, y si el producto cruz es



negativo,  $p$  está ubicado a la derecha. Finalmente, si el producto cruz es cero, los puntos  $s_1$ ,  $s_2$  y  $p$  están en la misma línea.

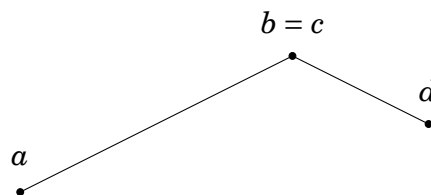
## Intersección de segmentos de línea

A continuación, consideramos el problema de probar si dos segmentos de línea  $ab$  y  $cd$  se cruzan. Los posibles casos son: *Caso 1*: Los segmentos de línea están en la misma línea y se superponen entre sí. En este caso, hay un número infinito de puntos de intersección. Por ejemplo, en la siguiente imagen, todos los puntos entre  $c$  y  $b$  son puntos de intersección:



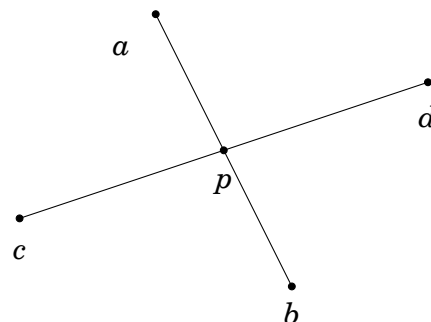
En este caso, podemos usar productos cruzados para comprobar si todos los puntos están en la misma línea. Después de esto, podemos ordenar los puntos y verificar si los segmentos de línea se superponen entre sí.

*Caso 2*: Los segmentos de línea tienen un vértice común que es el único punto de intersección. Por ejemplo, en la siguiente imagen el punto de intersección es  $b = c$ :



Este caso es fácil de comprobar, porque solo hay cuatro posibilidades para el punto de intersección:  $a = c$ ,  $a = d$ ,  $b = c$  y  $b = d$ .

*Caso 3*: Hay exactamente un punto de intersección que no es un vértice de ningún segmento de línea. En la siguiente imagen, el punto  $p$  es el punto de intersección:



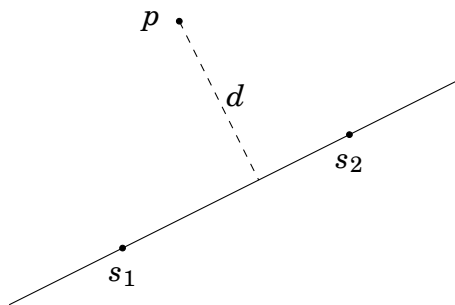
En este caso, los segmentos de línea se intersecan exactamente cuando ambos puntos  $c$  y  $d$  son en lados diferentes de una línea a través de  $a$  y  $b$ , y los puntos  $a$  y  $b$  están en diferentes lados de una línea a través de  $c$  y  $d$ . Podemos usar productos cruzados para comprobar esto.

## Distancia de un punto a una línea

Otra característica de los productos cruzados es que el área de un triángulo se puede calcular usando la fórmula

$$\frac{|(a - c) \times (b - c)|}{2},$$

donde  $a$ ,  $b$  y  $c$  son los vértices del triángulo. Usando este hecho, podemos derivar una fórmula para calcular la distancia más corta entre un punto y una línea. Por ejemplo, en la siguiente imagen  $d$  es la distancia más corta entre el punto  $p$  y la línea que está definida por los puntos  $s_1$  y  $s_2$ :

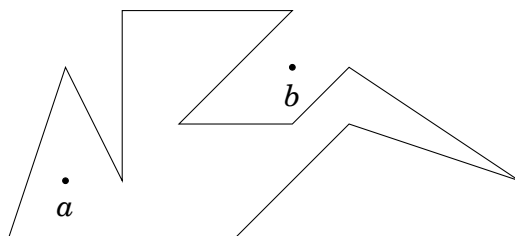


El área del triángulo cuyos vértices son  $s_1$ ,  $s_2$  y  $p$  se puede calcular de dos maneras: es tanto  $\frac{1}{2}|s_2 - s_1|d$  como  $\frac{1}{2}((s_1 - p) \times (s_2 - p))$ . Por lo tanto, la distancia más corta es

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

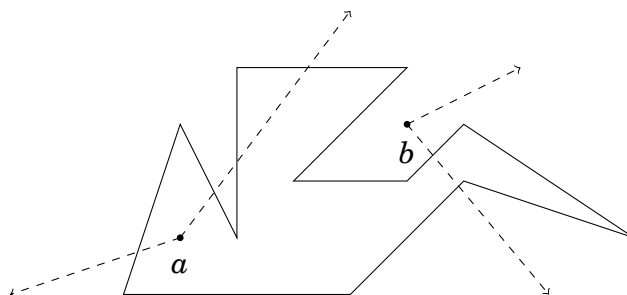
## Punto dentro de un polígono

Consideremos ahora el problema de probar si un punto está ubicado dentro o fuera de un polígono. Por ejemplo, en la siguiente imagen el punto  $a$  está dentro del polígono y el punto  $b$  está fuera del polígono.



Una forma conveniente de resolver el problema es enviar un *rayo* desde el punto a una dirección arbitraria y calcular el número de veces que toca el límite del polígono. Si el número es impar, el punto está dentro del polígono, y si el número es par, el punto está fuera del polígono.

Por ejemplo, podríamos enviar los siguientes rayos:



Los rayos desde  $a$  tocan 1 y 3 veces el límite del polígono, por lo que  $a$  está dentro del polígono. Correspondientemente, los rayos desde  $b$  tocan 0 y 2 veces el límite del polígono, por lo que  $b$  está fuera del polígono.

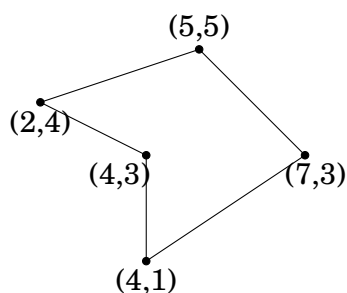
## 29.3 Área del polígono

Una fórmula general para calcular el área de un polígono, a veces llamada la fórmula del **cordón de zapatos**, es la siguiente:

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

Aquí los vértices son  $p_1 = (x_1, y_1)$ ,  $p_2 = (x_2, y_2)$ , ...,  $p_n = (x_n, y_n)$  en tal orden que  $p_i$  y  $p_{i+1}$  son vértices adyacentes en el límite del polígono, y el primer y último vértice es el mismo, es decir,  $p_1 = p_n$ .

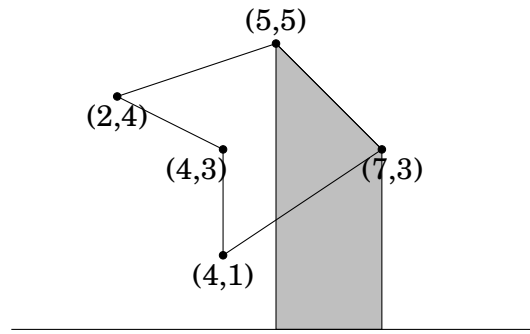
Por ejemplo, el área del polígono



es

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

La idea de la fórmula es recorrer los trapecios cuyo un lado es un lado del polígono, y otro lado yace en la línea horizontal  $y = 0$ . Por ejemplo:



El área de tal trapecio es

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

donde los vértices del polígono son  $p_i$  y  $p_{i+1}$ . Si  $x_{i+1} > x_i$ , el área es positiva, y si  $x_{i+1} < x_i$ , el área es negativa.

El área del polígono es la suma de áreas de todos los trapecios, lo que produce la fórmula

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Tenga en cuenta que se toma el valor absoluto de la suma, porque el valor de la suma puede ser positivo o negativo, dependiendo de si caminamos en sentido horario o antihorario a lo largo del límite del polígono.

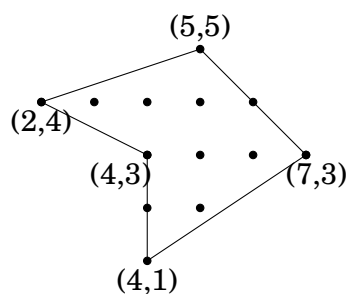
## Teorema de Pick

El **Teorema de Pick** proporciona otra forma de calcular el área de un polígono siempre que todos los vértices del polígono tengan coordenadas enteras. De acuerdo con el teorema de Pick, el área del polígono es

$$a + b/2 - 1,$$

donde  $a$  es el número de puntos enteros dentro del polígono y  $b$  es el número de puntos enteros en el límite del polígono.

Por ejemplo, el área del polígono



es  $6 + 7/2 - 1 = 17/2$ .

## 29.4 Funciones de distancia

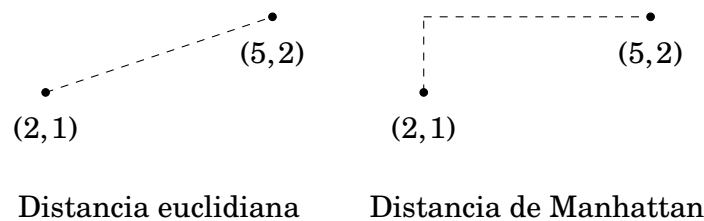
Una **función de distancia** define la distancia entre dos puntos. La función de distancia habitual es la **distancia euclidiana** donde la distancia entre los puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  es

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Una función de distancia alternativa es la **distancia de Manhattan** donde la distancia entre los puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  es

$$|x_1 - x_2| + |y_1 - y_2|.$$

Por ejemplo, considere la siguiente imagen:



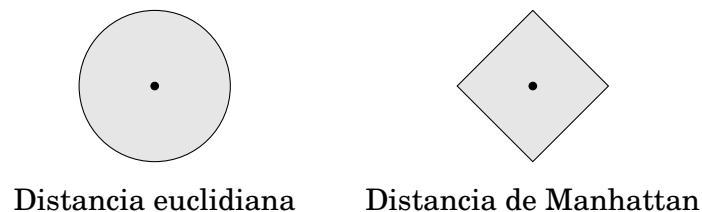
La distancia euclidiana entre los puntos es

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

y la distancia de Manhattan es

$$|5 - 2| + |2 - 1| = 4.$$

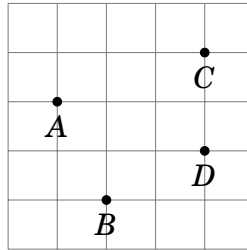
La siguiente imagen muestra regiones que están dentro de una distancia de 1 del punto central, usando las distancias euclidiana y de Manhattan:



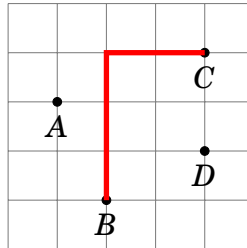
### Rotación de coordenadas

Algunos problemas son más fáciles de resolver si se usan distancias de Manhattan en lugar de distancias euclidianas. Como ejemplo, considere un problema donde se nos dan  $n$  puntos en el plano bidimensional y nuestra tarea es calcular la distancia de Manhattan máxima entre dos puntos cualesquiera.

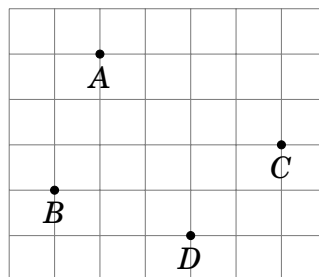
Por ejemplo, considere el siguiente conjunto de puntos:



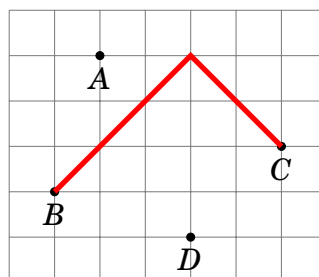
La distancia de Manhattan máxima es 5 entre los puntos  $B$  y  $C$ :



Una técnica útil relacionada con las distancias de Manhattan es rotar todas las coordenadas 45 grados para que un punto  $(x, y)$  se convierta en  $(x + y, y - x)$ . Por ejemplo, después de rotar los puntos anteriores, el resultado es:



Y la distancia máxima es la siguiente:



Considere dos puntos  $p_1 = (x_1, y_1)$  y  $p_2 = (x_2, y_2)$  cuyas coordenadas giradas son  $p'_1 = (x'_1, y'_1)$  y  $p'_2 = (x'_2, y'_2)$ . Ahora hay dos formas de expresar la distancia de Manhattan entre  $p_1$  y  $p_2$ :

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

Por ejemplo, si  $p_1 = (1, 0)$  y  $p_2 = (3, 3)$ , las coordenadas giradas son  $p'_1 = (1, -1)$  y  $p'_2 = (6, 0)$  y la distancia de Manhattan es

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

Las coordenadas giradas proporcionan una forma sencilla de operar con distancias de Manhattan, porque podemos considerar las coordenadas  $x$  e  $y$  por separado. Para maximizar la distancia de Manhattan entre dos puntos, deberíamos encontrar dos puntos cuyas coordenadas giradas maximicen el valor de

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

Esto es fácil, porque o la diferencia horizontal o la vertical de las coordenadas giradas tiene que ser máxima.





## Chapter 30

# Algoritmos de línea de barrido

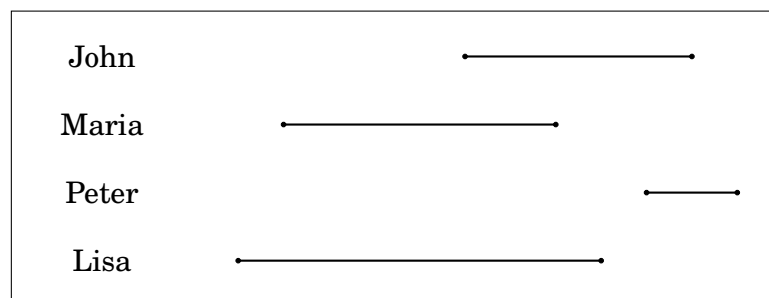
Muchos problemas geométricos se pueden resolver utilizando **algoritmos de línea de barrido**. La idea en estos algoritmos es representar una instancia del problema como un conjunto de eventos que corresponden a puntos en el plano. Los eventos se procesan en orden creciente de acuerdo con sus coordenadas  $x$  o  $y$ .

Como ejemplo, considere el siguiente problema: Hay una empresa que tiene  $n$  empleados, y conocemos para cada empleado su hora de llegada y su hora de salida en un día determinado. Nuestra tarea es calcular el número máximo de empleados que estaban en la oficina al mismo tiempo.

El problema se puede resolver modelando la situación de modo que cada empleado reciba dos eventos que correspondan a su hora de llegada y su hora de salida. Después de ordenar los eventos, los revisamos y hacemos un seguimiento del número de personas en la oficina. Por ejemplo, la tabla

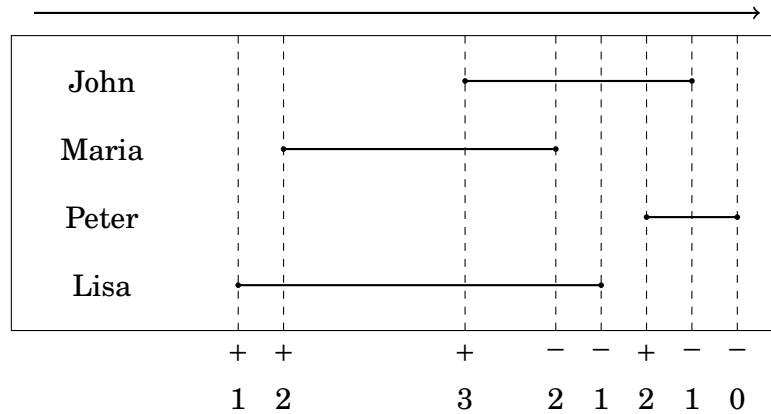
persona	hora de llegada	hora de salida
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

corresponde a los siguientes eventos:



Revisamos los eventos de izquierda a derecha y mantenemos un contador. Siempre que una persona llega, aumentamos el valor del contador en uno, y cuando una persona se va, disminuimos el valor del contador en uno. La respuesta al problema es el máximo valor del contador durante el algoritmo.

En el ejemplo, los eventos se procesan de la siguiente manera:

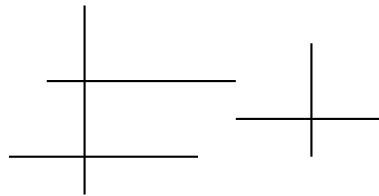


Los símbolos + y - indican si el valor del contador aumenta o disminuye, y el valor del contador se muestra debajo. El valor máximo del contador es 3 entre la llegada de John y la salida de María.

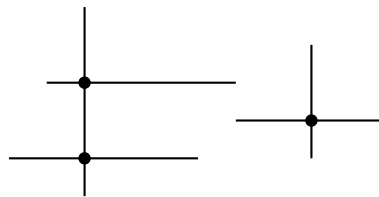
El tiempo de ejecución del algoritmo es  $O(n \log n)$ , porque ordenar los eventos lleva  $O(n \log n)$  tiempo y el resto del algoritmo lleva  $O(n)$  tiempo.

### 30.1 Puntos de intersección

Dado un conjunto de  $n$  segmentos de línea, cada uno de ellos siendo horizontal o vertical, considere el problema de contar el número total de puntos de intersección. Por ejemplo, cuando los segmentos de línea son



hay tres puntos de intersección:



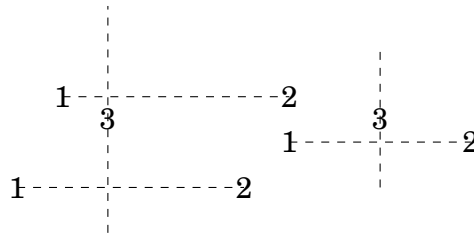
Es fácil resolver el problema en tiempo  $O(n^2)$ , porque podemos recorrer todas las parejas posibles de segmentos de línea y verificar si se intersecan. Sin embargo, podemos resolver el problema de manera más eficiente en tiempo  $O(n \log n)$  usando un algoritmo de línea de barrido y una estructura de datos de consulta de rango.

La idea es procesar los puntos finales de la línea segmentos de izquierda a derecha y centrarse en tres tipos de eventos:

- (1) segmento horizontal comienza
- (2) segmento horizontal termina

### (3) segmento vertical

Los siguientes eventos corresponden al ejemplo:



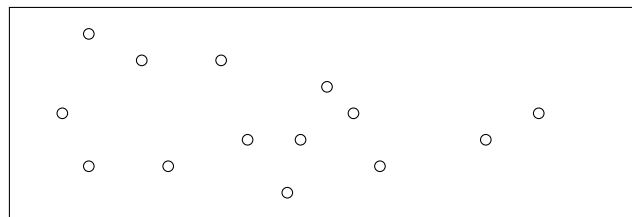
Recorremos los eventos de izquierda a derecha y usamos una estructura de datos que mantiene un conjunto de coordenadas y donde hay un segmento horizontal activo. En el evento 1, agregamos la coordenada y del segmento al conjunto, y en el evento 2, quitamos la coordenada y del conjunto.

Los puntos de intersección se calculan en el evento 3. Cuando hay un segmento vertical entre los puntos  $y_1$  e  $y_2$ , contamos el número de segmentos horizontales activos cuya coordenada y está entre  $y_1$  e  $y_2$ , y agregamos este número al total número de puntos de intersección.

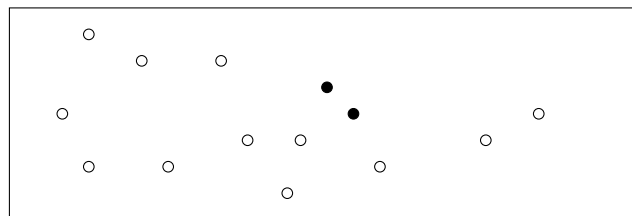
Para almacenar las coordenadas y de los segmentos horizontales, podemos usar un árbol binario indexado o un árbol de segmentos, posiblemente con compresión de índices. Cuando se utilizan tales estructuras, el procesamiento de cada evento lleva  $O(\log n)$  tiempo, por lo que el tiempo total de ejecución del algoritmo es  $O(n \log n)$ .

## 30.2 Problema del par más cercano

Dado un conjunto de  $n$  puntos, nuestro siguiente problema es encontrar dos puntos cuya distancia euclidiana sea mínima. Por ejemplo, si los puntos son



deberíamos encontrar los siguientes puntos:



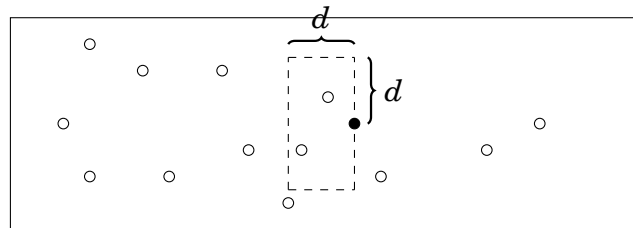
Este es otro ejemplo de un problema que se puede resolver en tiempo  $O(n \log n)$  usando un algoritmo de línea de barrido<sup>1</sup>. Recorremos los puntos de izquierda a

<sup>1</sup>Además de este enfoque, también hay un algoritmo de dividir y conquistar de tiempo  $O(n \log n)$  [56] que divide los puntos en dos conjuntos y recursivamente resuelve el problema para ambos conjuntos.

derecha y mantenemos un valor  $d$ : la distancia mínima entre dos puntos vistos hasta ahora. En cada punto, encontramos el punto más cercano a la izquierda. Si la distancia es menor que  $d$ , es la nueva distancia mínima y actualizamos el valor de  $d$ .

Si el punto actual es  $(x, y)$  y hay un punto a la izquierda a una distancia menor que  $d$ , la coordenada  $x$  de dicho punto debe estar entre  $[x - d, x]$  y la coordenada  $y$  debe estar entre  $[y - d, y + d]$ . Por lo tanto, basta con considerar puntos que están ubicados en esos rangos, lo que hace que el algoritmo sea eficiente.

Por ejemplo, en la siguiente imagen, la región marcada con líneas discontinuas contiene los puntos que pueden estar a una distancia de  $d$  del punto activo:



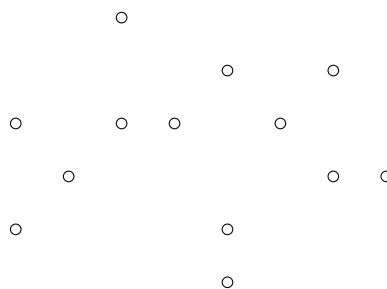
La eficiencia del algoritmo se basa en el hecho de que la región siempre contiene solo  $O(1)$  puntos. Podemos recorrer esos puntos en tiempo  $O(\log n)$  manteniendo un conjunto de puntos cuya coordenada  $x$  está entre  $[x - d, x]$ , en orden creciente según sus coordenadas  $y$ .

La complejidad temporal del algoritmo es  $O(n \log n)$ , porque recorreremos  $n$  puntos y encontramos para cada punto el punto más cercano a la izquierda en tiempo  $O(\log n)$ .

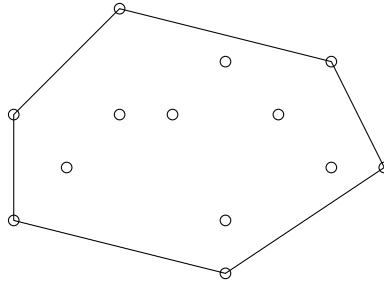
### 30.3 Problema de la envolvente convexa

Una **envolvente convexa** es el polígono convexo más pequeño que contiene todos los puntos de un conjunto dado. La convexidad significa que un segmento de línea entre dos vértices cualesquiera del polígono está completamente dentro del polígono.

Por ejemplo, para los puntos



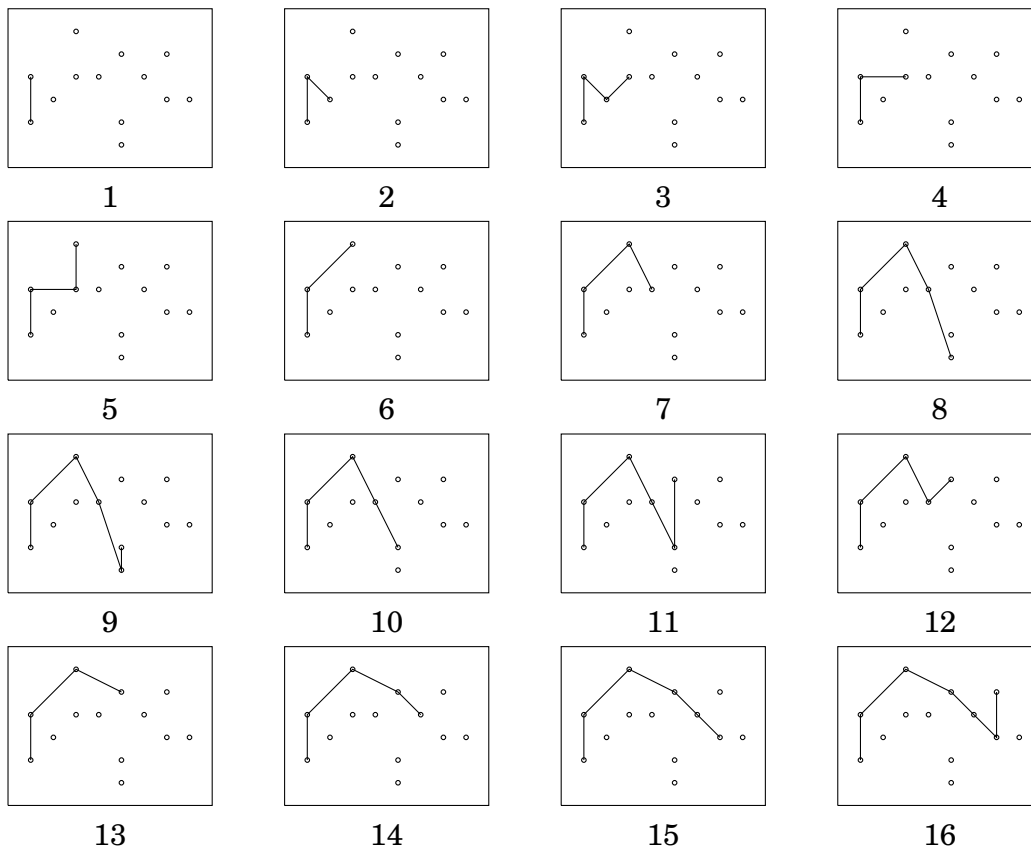
la envolvente convexa es la siguiente:

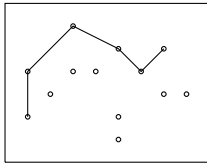


**Algoritmo de Andrew** [3] proporciona una forma sencilla de construir la envolvente convexa para un conjunto de puntos en tiempo  $O(n \log n)$ . El algoritmo primero localiza los puntos más a la izquierda y más a la derecha, y luego construye la envolvente convexa en dos partes: primero la envolvente superior y luego la envolvente inferior. Ambas partes son similares, por lo que podemos concentrarnos en construir la envolvente superior.

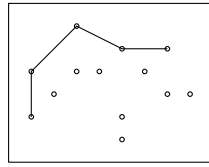
Primero, ordenamos los puntos principalmente según las coordenadas  $x$  y secundariamente según las coordenadas  $y$ . Después de esto, recorreremos los puntos y agregamos cada punto a la envolvente. Siempre después de agregar un punto a la envolvente, nos aseguramos de que el último segmento de línea en la envolvente no gire a la izquierda. Mientras gire a la izquierda, eliminamos repetidamente el penúltimo punto de la envolvente.

Las siguientes imágenes muestran cómo funciona el algoritmo de Andrew:

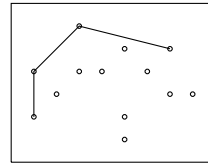




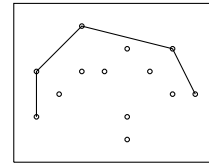
17



18



19



20

# Bibliography

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).
- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>
- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.



- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).
- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).

- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An  $O(n \log n)$  algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.



# Index

- 3SAT problem, 178
- 3SUM problem, 87
- alfabeto, 263
- Algoritmo de Andrew, 307
- Algoritmo de Bellman–Ford, 135
- Algoritmo de Dijkstra, 138, 167
- Algoritmo de Edmonds–Karp, 202
- Algoritmo de escalado, 203
- algoritmo de Euclides, 218
- algoritmo de Euclides extendido, 222
- algoritmo de Floyd, 170
- Algoritmo de Floyd–Warshall, 141
- Algoritmo de Ford–Fulkerson, 200
- algoritmo de Freivalds, 253
- Algoritmo de Hierholzer, 193
- algoritmo de Kosaraju, 174
- Algoritmo de Kruskal, 154
- Algoritmo de Las Vegas, 252
- Algoritmo de Mo, 277
- Algoritmo de Monte Carlo, 251
- Algoritmo de Prim, 159
- algoritmo de raíz cuadrada, 273
- Algoritmo SPFA, 138
- algoritmo voraz, 63
- Algoritmo Z, 268
- ancestro, 179
- antepasado común más bajo, 183
- análisis amortizado, 85
- arithmetic progression, 11
- aritmética modular, 7
- aritmética modular, 219
- arreglo de recorrido de árbol, 180
- arreglo dinámico, 39
- Arreglo Z, 268
- backtracking, 56
- Binet’s formula, 14
- binomial distribution, 250
- bitset, 46
- borde, 264
- búsqueda binaria, 34
- búsqueda en amplitud, 129
- búsqueda en profundidad, 127
- cadena, 40, 211, 263
- Camino Euleriano, 191
- camino hamiltoniano, 195
- camino más corto, 135
- ciclo, 131, 163, 169
- ciclo negativo, 137
- Circuito Euleriano, 192
- circuito hamiltoniano, 195
- Codificación de Huffman, 69
- coeficiente binomial, 226
- coeficiente multinomial, 228
- cofactor, 239
- coincidencia de patrones, 263
- cola, 48
- cola de prioridad, 48
- colisión, 266
- coloración, 122, 253
- combinatoria, 225
- complejidad temporal, 19
- complement, 12
- complex, 292
- complex number, 292
- complexity classes, 22
- component, 120
- componente fuertemente conexo, 173
- compresión de datos, 68
- conjetura de Goldbach, 217
- conjetura de Legendre, 217
- conjunction, 13
- conjunto, 41

connected graph, 120  
 constant factor, 23  
 constant-time algorithm, 22  
 consulta de máximo, 91  
 consulta de mínimo, 91  
 consulta de rango, 91  
 consulta de suma, 91  
 consulta de árbol, 179  
 coprimo, 219  
 corte, 200  
 Corte mínimo, 203  
 corte mínimo, 200  
 criba de Eratóstenes, 218  
 cubic algorithm, 22  
 cubierta de nodos, 208  
 cubierta de nodos mínima, 208  
 cycle, 119  
 código binario, 68  
 Código de Prüfer, 234  
  
 deque, 47  
 desarreglo, 232  
 descomposición prima, 215  
 desplazamiento de bits, 105  
 detección de ciclos, 169  
 determinant, 239  
 difference, 12  
 difference array, 102  
 disjunction, 13  
 distancia de edición, 80  
 Distancia de Hamming, 108  
 distancia de Levenshtein, 80  
 distancia de Manhattan, 299  
 distancia euclidiana, 299  
 distribución, 249  
 distribución uniforme, 250  
 divisibilidad, 215  
 divisor, 215  
 diámetro, 147  
  
 ecuación diofántica, 222  
 edge, 119  
 El juego de Grundy, 261  
 emparejamiento, 205  
 emparejamiento máximo, 205  
 emparejamiento perfecto, 207  
 en orden, 151  
  
 encuentro en el medio, 61  
 entero, 6  
 equivalence, 13  
 estado ganador, 255  
 estado perdedor, 255  
 estadística de orden, 252  
 estructura de datos, 39  
 estructura de unión-búsqueda, 157  
 expresión de paréntesis, 229  
  
 factor, 215  
 factorial, 14  
 Faulhaber's formula, 11  
 Fibonacci number, 14, 240  
 flujo, 199  
 flujo máximo, 199  
 función de comparación, 34  
 función de distancia, 299  
 Función mex, 259  
 Función totiente de Euler, 219  
 Fórmula de Cayley, 233  
 Fórmula de Euclides, 224  
 fórmula del cordón de zapatos, 297  
  
 geometric distribution, 250  
 geometric progression, 11  
 geometry, 291  
 grado, 121  
 grado de entrada, 121  
 grado de salida, 121  
 grafo bipartito, 132  
 grafo conectado, 131  
 grafo de componentes, 173  
 grafo fuertemente conexo, 173  
 grafo funcional, 168  
 grafo simple, 122  
 grafo sucesor, 168  
 graph, 119  
 gráfico bipartito, 122  
 gráfico completo, 121  
 gráfico dirigido, 120  
 gráfico ponderado, 121  
 gráfico regular, 121  
  
 harmonic sum, 12  
 hashing, 265  
 hashing de cadenas, 265

- hashing polinomial, 265
- Heron's formula, 291
- heurística, 198
- hijo, 145
- hoja, 145
- identity matrix, 238
- implication, 13
- inclusión-exclusión, 231
- independencia, 248
- independent set, 208
- index compression, 101
- input y output, 4
- intersección de segmentos de línea, 295
- intersection, 12
- inverse matrix, 240
- inversión, 28
- inverso modular, 221
- iterator, 43
- juego de nim, 257
- juego misère, 258
- Kadane's algorithm, 25
- Lema de Burnside, 232
- lenguaje de programación, 3
- linear algorithm, 22
- linear recurrence, 240
- lista de adyacencia, 123
- lista de aristas, 125
- logarithm, 15
- logarithmic algorithm, 22
- logic, 13
- línea de barrido, 303
- macro, 9
- map, 42
- Markov chain, 250
- matrix, 237
- matrix multiplication, 238
- matrix power, 239
- matriz de adyacencia, 124
- matriz de suma de prefijos, 92
- matriz laplaciana, 244
- maximum independent set, 208
- maximum subarray sum, 24
- memorización, 73
- merge sort, 29
- mochila, 79
- montículo, 48
- multiplicación de matrices, 252
- máximo común divisor, 218
- método de dos punteros, 85
- mínimo común múltiplo, 218
- mínimo de ventana deslizante, 89
- número de punto flotante, 7
- natural logarithm, 15
- nearest smaller elements, 87
- negation, 13
- next\_permutation, 55
- node, 119
- NP-hard problem, 23
- número de Catalan, 229
- número de Fibonacci, 224
- Número de Grundy, 259
- número perfecto, 216
- operación no, 105
- operación o, 105
- operación xor, 105
- operación y, 104
- operador de comparación, 33
- orden lexicográfico, 264
- ordenación por conteo, 31
- ordenación topológica, 163
- ordenamiento, 27
- ordenamiento por burbuja, 27
- padre, 145
- pair, 33
- palabra de código, 68
- par más cercano, 305
- paradoja del cumpleaños, 267
- path, 119
- path cover, 209
- periodo, 264
- permutación, 55
- pila, 47
- point, 292
- polynomial algorithm, 23
- posorden, 151
- predicate, 13

prefijo, 263  
 preorden, 151  
 primo, 215  
 primo gemelo, 217  
 probabilidad, 245  
 probabilidad condicional, 247  
 Problema 2SAT, 176  
 problema 2SUM, 86  
 problema de la reina, 56  
 producto cruz, 294  
 programación dinámica, 71  
 propagación perezosa, 282  
 punto de intersección, 304  
  
 quadratic algorithm, 22  
 quantifier, 13  
 quickselect, 252  
 quicksort, 252  
  
 random\_shuffle, 44  
 randomized algorithm, 251  
 raíz, 145  
 recorrido del caballo, 197  
 Regla de Warnsdorf, 198  
 representación de bits, 103  
 resto, 7  
 reverse, 44  
 rotación, 263  
  
 Secuencia de De Bruijn, 196  
 set, 12  
 set theory, 12  
 sort, 32, 44  
 square matrix, 237  
 subcadena, 263  
 subconjunto, 53  
 subsecuencia, 263  
 subsecuencia creciente más larga, 76  
 subset, 12  
 subárbol, 145  
 sufijo, 263  
 suma armónica, 218  
 suma de nim, 257  
  
 tabla dispersa, 93  
 teorema chino del resto, 223  
 Teorema de Dilworth, 211  
  
 Teorema de Dirac, 196  
 Teorema de Euler, 220  
 Teorema de Fermat, 220  
 Teorema de Hall, 207  
 Teorema de Kirchhoff, 243  
 Teorema de König, 208  
 teorema de Lagrange, 223  
 Teorema de Ore, 196  
 Teorema de Pick, 298  
 Teorema de Sprague–Grundy, 258  
 Teorema de Wilson, 224  
 teorema de Zeckendorf, 224  
 teoría de números, 215  
 transpose, 237  
 tree, 120  
 trie, 264  
 Triple pitagórico, 224  
 Triángulo de Pascal, 227  
 tuple, 33  
 typedef, 8  
 técnica de recorrido de Euler, 184  
  
 union, 12  
 universal set, 12  
  
 valor esperado, 249  
 valor hash, 265  
 variable aleatoria, 248  
 vecino, 121  
 vector, 39, 237, 292  
 ventana deslizante, 89  
  
 árbol, 145  
 árbol binario, 151  
 árbol de expansión, 153, 243  
 árbol de expansión máximo, 154  
 árbol de expansión mínimo, 153  
 árbol de Fenwick, 94  
 árbol de segmentos, 97, 281  
 árbol de segmentos bidimensional, 288  
 árbol de segmentos dinámico, 285  
 árbol de segmentos disperso, 286  
 árbol de segmentos perezoso, 282  
 árbol de segmentos persistente, 286  
 árbol enraizado, 145  
 árbol indexado binario, 94