

Manual de Programación Competitiva

Antti Laaksonen

Borrador April 17, 2023

Contents

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

6	Algoritmos voraces	63
6.1	Problema de monedas	63
6.2	Planificación	64
6.3	Tareas y tiempos límite	66
6.4	Minimizando sumas	67
6.5	Compresión de datos	68
7	Programación dinámica	71
7.1	Problema de las monedas	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 formas de colocar baldosas	82
8	Análisis amortizado	85
8.1	Método de dos punteros	85
8.2	Elementos menores más cercanos	87
8.3	Mínimo de ventana deslizante	89
9	Consultas de rango	91
9.1	Consultas de arreglos estáticos	92
9.2	Árbol binario indexado	94
9.3	Árbol de segmentos	97
9.4	Técnicas adicionales	101
10	Manipulación de bits	103
10.1	Representación de bits	103
10.2	Operaciones de bits	104
10.3	Representación de conjuntos	106
10.4	Optimizaciones de bits	108
10.5	Programación dinámica	110
II	Graph algorithms	115
11	Basics of graphs	117
11.1	Graph terminology	117
11.2	Graph representation	121
12	Graph traversal	125
12.1	Depth-first search	125
12.2	Breadth-first search	127
12.3	Applications	129

13 Shortest paths	131
13.1 Bellman–Ford algorithm	131
13.2 Dijkstra’s algorithm	134
13.3 Floyd–Warshall algorithm	137
14 Tree algorithms	141
14.1 Tree traversal	142
14.2 Diameter	143
14.3 All longest paths	145
14.4 Binary trees	147
15 Spanning trees	149
15.1 Kruskal’s algorithm	150
15.2 Union-find structure	153
15.3 Prim’s algorithm	155
16 Directed graphs	157
16.1 Topological sorting	157
16.2 Dynamic programming	159
16.3 Successor paths	162
16.4 Cycle detection	163
17 Strong connectivity	165
17.1 Kosaraju’s algorithm	166
17.2 2SAT problem	168
18 Tree queries	171
18.1 Finding ancestors	171
18.2 Subtrees and paths	172
18.3 Lowest common ancestor	175
18.4 Offline algorithms	178
19 Paths and circuits	181
19.1 Eulerian paths	181
19.2 Hamiltonian paths	185
19.3 De Bruijn sequences	186
19.4 Knight’s tours	187
20 Flows and cuts	189
20.1 Ford–Fulkerson algorithm	190
20.2 Disjoint paths	194
20.3 Maximum matchings	195
20.4 Path covers	198

III	Advanced topics	203
21	Number theory	205
21.1	Primes and factors	205
21.2	Modular arithmetic	209
21.3	Solving equations	212
21.4	Other results	213
22	Combinatorics	215
22.1	Binomial coefficients	216
22.2	Catalan numbers	218
22.3	Inclusion-exclusion	220
22.4	Burnside's lemma	222
22.5	Cayley's formula	223
23	Matrices	225
23.1	Operations	225
23.2	Linear recurrences	228
23.3	Graphs and matrices	230
24	Probability	233
24.1	Calculation	233
24.2	Events	234
24.3	Random variables	236
24.4	Markov chains	238
24.5	Randomized algorithms	239
25	Game theory	243
25.1	Game states	243
25.2	Nim game	245
25.3	Sprague–Grundy theorem	246
26	String algorithms	251
26.1	String terminology	251
26.2	Trie structure	252
26.3	String hashing	253
26.4	Z-algorithm	255
27	Square root algorithms	259
27.1	Combining algorithms	260
27.2	Integer partitions	262
27.3	Mo's algorithm	263
28	Segment trees revisited	265
28.1	Lazy propagation	266
28.2	Dynamic trees	269
28.3	Data structures	271
28.4	Two-dimensionality	272

29 Geometry	273
29.1 Complex numbers	274
29.2 Points and lines	276
29.3 Polygon area	279
29.4 Distance functions	280
30 Sweep line algorithms	283
30.1 Intersection points	284
30.2 Closest pair problem	285
30.3 Convex hull problem	286
Bibliography	289

Prefacio

El propósito de este libro es proporcionar una introducción completa a la programación competitiva. Se asume que ya conoces lo básico de la programación, pero no se requiere experiencia previa en programación competitiva.

El libro está especialmente diseñado para 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.

Lleva mucho tiempo convertirse en una persona con buenas habilidades en programación competitiva, pero también es una oportunidad para aprender mucho. Puedes estar seguro de que obtendrás una buena comprensión general de los algoritmos si dedicas tiempo a leer el libro, resolver problemas y participar en concursos.

El libro está en constante desarrollo. Siempre puedes enviar comentarios sobre el libro a ahslaaks@cs.helsinki.fi.

Helsinki, agosto de 2019
Antti Laaksonen

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 de manera creativa. Un algoritmo para resolver un problema debe ser tanto correcto como eficiente, y el núcleo del problema a menudo trata de inventar un algoritmo eficiente.

El conocimiento teórico de los algoritmos es importante para las personas que participan en la programación competitiva. Típicamente, una solución a un problema es una combinación de técnicas conocidas y nuevos conocimientos. Las técnicas que aparecen en la programación competitiva también forman la base para la investigación científica de los algoritmos.

La **implementación de algoritmos** requiere buenas habilidades de programación. En la programación competitiva, las soluciones se califican mediante la prueba de un algoritmo implementado usando un conjunto de casos de prueba. Por lo tanto, no es suficiente 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 (generalmente como máximo unas pocas cientos de líneas de código) y no necesitan mantenimiento después del concurso.

1.1 Lenguajes de programación

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

Muchas personas piensan que C++ es la mejor opción para una persona que participa en programación competitiva, y C++ está casi siempre disponible en sistemas de competencias. Los beneficios de usar C++ son que es un lenguaje

muy eficiente y su biblioteca estándar contiene una gran colección de estructuras de datos y algoritmos.

Por otro lado, es bueno dominar varios lenguajes y comprender sus fortalezas. Por ejemplo, si se necesitan enteros grandes en el problema, Python puede ser una buena opción, porque contiene operaciones integradas para calcular con enteros grandes. Aún así, la mayoría de los problemas en concursos de programación están establecidos de tal manera que usar un lenguaje de programación específico no representa una ventaja injusta.

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

Plantilla de código C++

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

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

using namespace std;

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

La línea `#include` al principio del código es una característica del compilador `g++` que nos permite incluir toda la biblioteca estándar. Por lo tanto, 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 se pueden utilizar directamente en el código. Sin la línea `using`, tendríamos que escribir, por ejemplo, `std::cout`, pero ahora es suficiente con escribir `cout`.

El código se puede compilar utilizando 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 Entrada y salida

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

La entrada para el programa generalmente 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, suponiendo que haya al menos un espacio o salto de línea entre cada elemento en la entrada. Por ejemplo, el código anterior puede leer ambas entradas siguientes:

```
123 456 mono
```

```
123    456  
mono
```

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

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

La entrada y salida a veces es un cuello de botella en el programa. Las siguientes líneas al comienzo del código hacen que la entrada y salida sean más eficientes:

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

Tenga en cuenta que la línea nueva "\n" funciona más rápido que endl, porque endl siempre provoca una operación de vaciado.

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

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

El siguiente código imprime dos enteros:

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

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

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

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

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

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

En algunos sistemas de competencia, se utilizan archivos para entrada y salida. Una solución fácil para esto es escribir el código como de costumbre utilizando flujos estándar, pero agregue las siguientes líneas al comienzo del código:

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

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

1.3 Trabajando con números

Enteros

El tipo de entero más utilizado en 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 utilizar 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 de número es `long long`.

Un error común al usar el tipo `long long` es que el tipo `int` todavía se usa en alguna parte en el 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 de tipo `long long`, ambos números en la expresión `a*a` son de tipo `int` y el resultado es también de tipo `int`. Debido a esto, la variable `b` contendrá un resultado incorrecto. El problema se puede solucionar cambiando el tipo de `a` a `long long` o cambiando la expresión a `(long long)a*a`.

Por lo general, los problemas de concurso se establecen de tal manera que el tipo `long long` es suficiente. Aún 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 concurso.

Aritmética modular

Denotamos por $x \bmod m$ el residuo 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 darlo "módulo m ", es decir, el residuo 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 utilizar los tipos `int` y `long long`.

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

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

Así, podemos tomar el residuo después de cada operación y los números nunca serán demasiado grandes.

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

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

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

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

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

Números de punto flotante

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

La precisión requerida de la respuesta generalmente se proporciona en el enunciado del problema. Una forma fácil de mostrar la respuesta es utilizar la

función `printf` e indicar el número de decimales en la cadena de formato. Por ejemplo, el siguiente código imprime el valor de x con 9 decimales:

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

Una dificultad al usar números de punto flotante es que algunos números no 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 sería 1.

Es arriesgado comparar números de punto flotante con el operador `==`, porque es posible que los valores deban ser iguales pero no lo son debido a errores de precisión. Una mejor manera de comparar números de punto flotante es suponer que dos números son iguales si la diferencia entre ellos es menor que ε , donde ε 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
}
```

Tenga en cuenta que, aunque los números de punto flotante son inexactos, los enteros hasta cierto límite aún pueden ser representados con precisión. Por ejemplo, utilizando `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 programación competitiva, porque los programas deben escribirse lo más rápido posible. Por ello, los programadores competitivos suelen definir 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";
```

puede acortarse de la siguiente manera:

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

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

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

Macros

Otra forma de acortar el código es definir **macros**. Un macro significa que ciertas cadenas en el código se cambiarán 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, considere el siguiente macro que calcula el cuadrado de un número:

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

Este macro *no* siempre funciona como se esperaba. Por ejemplo, el código

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

corresponde al código

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

Una mejor versión 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 una persona exitosa en programación competitiva sin tener buenas habilidades matemáticas. Esta sección trata algunos conceptos y fórmulas matemáticas importantes que serán necesarios 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 número entero positivo, tiene una fórmula cerrada que es un polinomio de grado $k + 1$. Por ejemplo¹,

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

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 proporción 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 proporción 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

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

Al multiplicar ambos lados por k , obtenemos

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

¹ Existe incluso 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 $\log_2(n) + 1$. Es decir, podemos modificar cada término $1/k$ de modo que k se convierta en la potencia de dos más cercana que no exceda 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, la cantidad 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 elementos que están tanto en A como en B . Por ejemplo, si $A = \{1, 2, 5\}$ y $B = \{2, 4\}$, entonces $A \cap B = \{2\}$.
- La **unión** $A \cup B$ consiste en elementos que están en A o en 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 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 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,

incluido 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 utilizados con frecuencia son \mathbb{N} (números naturales), \mathbb{Z} (enteros), \mathbb{Q} (números racionales) y \mathbb{R} (números reales). El conjunto \mathbb{N} se puede definir de dos 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 números 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 tanto A como 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 siempre que A es verdadero, también lo es B . 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 se denotan generalmente 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 con los elementos de un conjunto. Los cuantificadores más importantes son \forall (**para todo**) y \exists (**existe**). Por ejemplo,

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

significa que para cada elemento x en el conjunto, hay un elemento y en el conjunto tal que y es menor que x . Esto es cierto en el conjunto de los números enteros, pero falso en el conjunto de los números naturales.

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

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

significa que si un número x es mayor que 1 y no es un número primo, entonces 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** $n!$ puede definirse

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

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 existe 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 como $\log_k(x)$, donde k es la base del logaritmo. Según la definición, $\log_k(x) = a$ exactamente cuando $k^a = x$.

Una propiedad útil de los logaritmos es que $\log_k(x)$ es igual al número de veces que 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, por sus siglas en inglés) es un concurso anual de programación para estudiantes de secundaria. Cada país puede enviar un equipo de cuatro estudiantes al concurso. Suele haber unos 300 participantes de 80 países.

La IOI consta de dos concursos de cinco horas de duración. En ambos concursos, se pide a los participantes que resuelvan tres tareas de algoritmos de dificultad variable. 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 por 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 Centro Europea de Informática (CEOI) y la Olimpiada de Informática Asia-Pacífico (APIO).

Algunos países organizan concursos de práctica en línea para futuros participantes de la IOI, como la Competencia Abierta Croata de Informática [11] y la Olimpiada de Computación de EE. UU. [68]. Además, hay una gran colección de problemas de concursos polacos disponibles en línea [60].

ICPC

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

El ICPC consta de varias etapas y, finalmente, los mejores equipos son invitados a la Final Mundial. Aunque hay decenas de miles de participantes en el concurso, solo hay una pequeña cantidad² de plazas finales disponibles, por lo que incluso avanzar a las finales es un gran logro en algunas regiones.

En cada concurso del ICPC, los equipos tienen cinco horas para resolver alrededor de diez problemas de algoritmos. Una solución a un problema es aceptada solo si resuelve todos los casos de prueba de manera eficiente. Durante el concurso, los competidores pueden ver los resultados de otros equipos, pero durante la última hora, el marcador 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 habilidades matemáticas adicionales.

Concursos en línea

También hay muchos concursos en línea abiertos para todos. Actualmente, 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 utilizan esos concursos para contratar: tener un buen desempeño en un concurso es una buena manera de demostrar habilidades.

²El número exacto de plazas finales varía de un año a otro; en 2017, había 133 plazas finales.

Libros

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

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim and F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]
- K. Diks et al.: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

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

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

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

Chapter 2

Complejidad temporal

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

La **complejidad temporal** de un algoritmo estima cuánto tiempo utilizará el algoritmo para alguna entrada. La idea es representar la eficiencia como una función cuyo parámetro es el tamaño de la entrada. Al calcular la complejidad temporal, 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. Por lo general, la variable n denota el tamaño de la entrada. Por ejemplo, si la entrada es un conjunto de números, n será el tamaño del conjunto, y si la entrada es una cadena, n será la longitud de la cadena.

Bucles

Una razón común por la que un algoritmo es lento es que contiene muchos bucles que recorren la entrada. Cuanto 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++) {  
    // código  
}
```

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

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

```
}
```

Orden de magnitud

Una complejidad temporal no 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 consta de fases consecutivas, la complejidad temporal total es la mayor complejidad temporal de una sola fase. La razón de esto es que la fase más lenta suele ser el cuello de botella del código.

Por ejemplo, el siguiente código consta de tres fases con complejidades temporales $O(n)$, $O(n^2)$ y $O(n)$. Por 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++) {  
    // código  
}
```

Varias variables

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

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

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

Recursión

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

Por ejemplo, considere la siguiente función:

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

La llamada $f(n)$ 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, considere la siguiente función:

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

En este caso, cada llamada a la función genera dos otras llamadas, excepto para $n = 1$. Veamos qué sucede 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}

Basándonos 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 los 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 reduce 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 es a menudo la mejor complejidad temporal posible, ya que 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 ordenamiento eficientes es $O(n \log n)$. Otra posibilidad es que el algoritmo utilice una estructura de datos en la que cada operación tarda $O(\log n)$ en tiempo.
- $O(n^2)$ Un algoritmo **cuadrático** a menudo contiene dos bucles anidados. Es posible recorrer todos los pares de los elementos de entrada en tiempo $O(n^2)$.
- $O(n^3)$ Un algoritmo **cúbico** a menudo contiene tres bucles anidados. Es posible recorrer todos los tríos de los 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 como máximo $O(n^k)$ 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-difíciles** son un conjunto importante de problemas, para los cuales no se conoce ningún algoritmo polinomial¹.

2.3 Estimación de la eficiencia

Al calcular la complejidad temporal de un algoritmo, es posible verificar, antes de implementar el algoritmo, que es suficientemente eficiente para el problema. El punto de partida para las estimaciones es el hecho de que una computadora moderna puede realizar varios 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 unas 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 del 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 generarían un algoritmo con una peor complejidad temporal.

¹Un 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].

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

2.4 Suma máxima de subarreglo

A menudo hay varios algoritmos posibles para resolver un problema de manera que sus complejidades temporales sean diferentes. Esta sección analiza 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 arreglo². El problema es interesante cuando puede haber valores negativos en el arreglo. Por ejemplo, en el arreglo

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

el siguiente subarreglo produce la suma máxima 10:

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

Asumimos que se permite un subarreglo vacío, así que la suma de subarreglo máximo es siempre al menos 0.

Algoritmo 1

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

```
int 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";
```

²El libro de J. Bentley *Programming Pearls* [8] hizo popular el problema.

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

La complejidad en tiempo 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 un bucle de él. Esto es posible calculando la suma al mismo tiempo que se mueve el extremo derecho del subarreglo. 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 en tiempo es $O(n^2)$.

Algoritmo 3

Sorprendentemente, es posible resolver el problema en tiempo $O(n)$ ³, lo que significa que solo se necesita un bucle. La idea es calcular, para cada posición del arreglo, la suma máxima de un subarreglo que termina en esa posición. Después de esto, la respuesta al problema es el máximo de esas sumas.

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

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

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

El siguiente código implementa el algoritmo:

³En [8], este algoritmo de tiempo lineal se atribuye a J. B. Kadane, y el algoritmo a veces se llama **Algoritmo de Kadane**.

```

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

```

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

Comparación de eficiencia

Es interesante estudiar qué tan 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 al azar. El tiempo necesario para leer la entrada no fue medido.

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 la entrada es pequeño, pero las entradas más grandes resaltan 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 al instante.

Chapter 3

Ordenamiento

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

Por ejemplo, el problema "¿un arreglo contiene dos elementos iguales?" es fácil de resolver usando ordenamiento. Si el arreglo contiene dos elementos iguales, estarán uno al lado del otro después de ordenar, así 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 ordenar, y también son 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 ordenamiento como una subrutina también tienen esta complejidad de tiempo.

3.1 Teoría del 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 ascendente.

Por ejemplo, el arreglo

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

será como sigue después de ordenar:

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

Algoritmos $O(n^2)$

Los algoritmos simples para ordenar un arreglo funcionan en tiempo $O(n^2)$. Estos algoritmos son cortos y generalmente consisten en dos bucles anidados.

Un famoso algoritmo de ordenamiento en tiempo $O(n^2)$ es el **ordenamiento burbuja**, donde los elementos "burbujean" en el arreglo según sus valores.

Bubble sort consta de n rondas. En cada ronda, el algoritmo itera a través de los elementos del arreglo. Siempre que 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 bubble sort intercambia elementos como sigue:

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



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



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



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



Inversiones

Bubble sort es un ejemplo de un algoritmo de ordenamiento que siempre intercambia elementos *consecutivos* en el arreglo. Resulta que la complejidad temporal de tal algoritmo es *siempre* al menos $O(n^2)$, porque en el peor caso, se requieren $O(n^2)$ intercambios para ordenar el arreglo.

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

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 el arreglo. Un arreglo está completamente ordenado cuando no hay inversiones. Por otro lado, si los elementos del arreglo 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 están en el orden incorrecto elimina exactamente una inversión del arreglo. Por lo tanto, si un algoritmo de ordenamiento solo puede intercambiar elementos consecutivos, cada intercambio elimina como máximo una inversión, y la complejidad en tiempo del algoritmo es al menos $O(n^2)$.

$O(n \log n)$ algoritmos

Es posible ordenar un arreglo de manera eficiente en tiempo $O(n \log n)$ utilizando algoritmos que no se limitan a intercambiar elementos consecutivos. Un algoritmo de este tipo es el **ordenamiento por mezcla**¹, que se basa en la recursión.

El ordenamiento por mezcla ordena un subarreglo $\text{array}[a \dots b]$ de la siguiente manera:

1. Si $a = b$, no hacer nada, porque el subarreglo ya está ordenado.
2. Calcular la posición del elemento medio: $k = \lfloor (a + b)/2 \rfloor$.
3. Ordenar recursivamente el subarreglo $\text{array}[a \dots k]$.
4. Ordenar recursivamente el subarreglo $\text{array}[k + 1 \dots b]$.
5. *Mezclar* los subarreglos ordenados $\text{array}[a \dots k]$ y $\text{array}[k + 1 \dots b]$ en un subarreglo ordenado $\text{array}[a \dots b]$.

El ordenamiento por mezcla es un algoritmo eficiente, ya que reduce a la mitad el tamaño del subarreglo en cada paso. La recursión consta de $O(\log n)$ niveles, y procesar cada nivel toma $O(n)$ tiempo. Mezclar los subarreglos $\text{array}[a \dots k]$ y $\text{array}[k + 1 \dots b]$ es posible en tiempo lineal, porque ya están ordenados.

Por ejemplo, considere ordenar el siguiente arreglo:

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

¹Según [47], el ordenamiento por mezcla fue inventado por J. von Neumann en 1945.

El arreglo se dividirá en dos subarreglos de la siguiente manera:

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

Entonces, los subarreglos se ordenarán recursivamente de la siguiente manera:

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

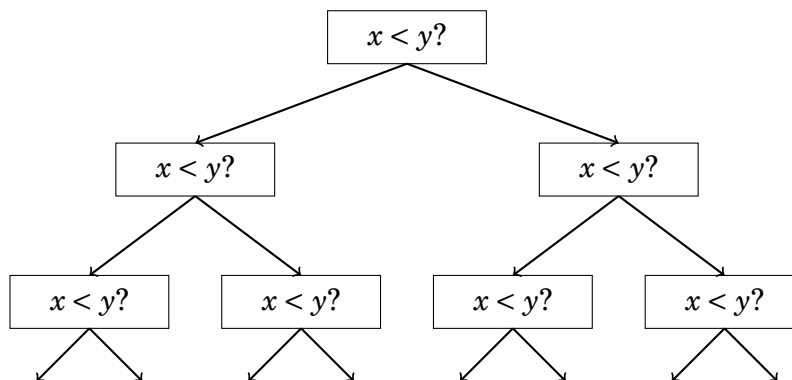
Finalmente, el algoritmo combina los subarreglos ordenados y crea el arreglo ordenado final:

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

Cota inferior de ordenamiento

¿Es posible ordenar un arreglo más rápido que en tiempo $O(n \log n)$? Resulta que esto *no* es posible cuando nos limitamos a algoritmos de ordenamiento que se basan en comparar elementos del arreglo.

La cota inferior para la complejidad del tiempo se puede demostrar considerando el ordenamiento como un proceso en el que cada comparación de dos elementos proporciona más información sobre el contenido del arreglo. El proceso crea el siguiente árbol:



Aquí " $x < y$?" significa que algunos elementos x y y son comparados. Si $x < y$, el proceso continúa a la izquierda, y en caso contrario a la derecha. Los resultados del proceso son las posibles maneras de ordenar el arreglo, 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 una cota inferior para esta suma al elegir los últimos $n/2$ elementos y cambiar 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 algoritmo de ordenamiento en el peor caso es al menos $n \log n$.

Ordenamiento por conteo

La cota inferior $n \log n$ no se aplica a algoritmos que no comparan elementos del arreglo sino que utilizan alguna otra información. Un ejemplo de tal algoritmo es **ordenamiento por conteo** que ordena un arreglo en tiempo $O(n)$ suponiendo que cada elemento en el arreglo es un entero entre $0 \dots c$ y $c = O(n)$.

El algoritmo crea un arreglo de *registro*, cuyos índices son elementos del arreglo original. El algoritmo itera a través del arreglo original y calcula cuántas veces aparece cada elemento en el arreglo.

Por ejemplo, el arreglo

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

corresponde al siguiente arreglo de registro:

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

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

La construcción del arreglo de registro toma un tiempo de $O(n)$. Después de esto, el arreglo ordenado se puede crear en tiempo $O(n)$ porque el número de ocurrencias de cada elemento se puede obtener del arreglo de registro. Por lo tanto, la complejidad total en tiempo del ordenamiento por conteo es $O(n)$.

El ordenamiento por conteo es un algoritmo muy eficiente pero solo se puede usar cuando la constante c es lo suficientemente pequeña, de modo que los elementos del arreglo puedan usarse como índices en el arreglo de registro.

3.2 Ordenamiento en C++

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

Hay muchos beneficios al usar una función de biblioteca. En primer lugar, 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 ordenamiento casera sea mejor.

En esta sección veremos cómo utilizar 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 del ordenamiento, 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 un arreglo ordinario de la siguiente manera:

```
int n = 7; // tamaño del arreglo  
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 se ordenan. 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 dato de los elementos a ordenar. Al ordenar, este operador se utilizará siempre que sea necesario averiguar el orden de dos elementos.

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

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.²:

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

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

Structs definidos por el usuario

Los structs definidos por el usuario no tienen un operador de comparación automáticamente. El operador debe definirse dentro del struct como una función operator<, cuyo parámetro es otro elemento del mismo tipo. El operador debe devolver true si el elemento es menor que el parámetro, y false en caso contrario.

²Cabe mencionar que en algunos compiladores 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}).

Por ejemplo, el siguiente struct P contiene las coordenadas x e y de un punto. El operador de comparación se define de tal manera que los puntos se ordenan 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 proporcionar una **función de comparación** externa a la función sort como una función de retrollamada. Por ejemplo, la siguiente función de comparación comp ordena las cadenas de texto principalmente por longitud y en segundo lugar 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 de texto 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 itere 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 en tiempo de este enfoque es $O(n)$, porque en el peor caso, es necesario verificar 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 deberíamos 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 tiempo $O(\log n)$.

Método 1

La forma habitual de implementar la búsqueda binaria se asemeja a buscar una palabra en un diccionario. La búsqueda mantiene una región activa en el array, que inicialmente contiene todos los elementos del array. Luego, se realizan una serie de pasos, cada uno de los cuales divide por la mitad el tamaño de la región.

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

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

En esta implementación, la región activa es $a \dots b$, y al principio 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 del vector. La idea es hacer saltos y disminuir la velocidad cuando nos acercamos al elemento objetivo.

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

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 actual del salto. La complejidad temporal del algoritmo es $O(\log n)$, porque el código en el bucle `while` se realiza 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 del vector cuyo valor es al menos x .
- `upper_bound` devuelve un puntero al primer elemento del vector cuyo valor es mayor que x .
- `equal_range` devuelve ambos punteros.

Las funciones asumen que el vector está ordenado. Si no hay tal elemento, el puntero apunta al elemento después del último elemento del vector. Por ejemplo, el siguiente código averigua si un vector 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 la cantidad 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";
```

Encontrando la solución más pequeña

Un uso importante para la búsqueda binaria es encontrar la posición en la que el valor de una *función* cambia. Supongamos que queremos encontrar el valor más pequeño k que es una solución válida para un problema. Se nos da una función $ok(x)$ que devuelve `true` si x es una solución válida y `false` en caso 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	\dots	$k-1$	k	$k+1$	\dots
$ok(x)$	false	false	\dots	false	true	true	\dots

Ahora, el valor de k se puede encontrar usando 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 mayor valor 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 del salto inicial z debe ser suficientemente grande, por ejemplo, algún valor para el cual sepamos de antemano que $ok(z)$ es true.

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

Encontrando el valor máximo

La búsqueda binaria también se puede utilizar para encontrar el valor máximo de una función que primero aumenta y luego disminuye. 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 utilizar la búsqueda binaria para encontrar el mayor valor 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 está permitido 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 adecuada para un problema, ya que 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 de la biblioteca estándar de C++. Es una buena idea utilizar 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 puede cambiar durante la ejecución del programa. El arreglo dinámico más popular en C++ es la estructura `vector`, que se puede utilizar casi como un arreglo ordinario.

El siguiente código crea un vector vacío y añade 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 proporcionar 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 la complejidad de tiempo promedio de `push_back` es $O(1)$.

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

El siguiente código presenta algunas operaciones de cadenas:

```
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 conjuntos

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

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

La elección de qué implementación de conjunto utilizar a menudo es una cuestión de preferencia. 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 utilizar principalmente como un vector, pero no es posible acceder a los elementos utilizando 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 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 presente. 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 funcionan de manera similar a `set` y `unordered_set` pero pueden contener múltiples instancias de un elemento. Por ejemplo, en el siguiente código, se agregan las tres instancias del número 5 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 cual se puede hacer de la siguiente manera:

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

4.3 Estructuras de mapas

Un **mapa** es un arreglo generalizado que consiste en pares clave-valor. Mientras que las claves en un arreglo ordinario son siempre los enteros consecutivos $0, 1, \dots, n-1$, donde n es el tamaño del arreglo, las claves en un mapa pueden ser de cualquier tipo de dato y no tienen que ser valores consecutivos.

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

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

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

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

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

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

```
if (m.count("aybaltu")) {
    // la 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 a menudo utilizados `begin` y `end` definen un rango que contiene todos los elementos en una estructura de datos. El iterador `begin` apunta al 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()
```

Tome nota de 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 *semiabierto*.

Trabajando con rangos

Los iteradores se utilizan en 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 proporcionan 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 mezcla 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 utilizar 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 conjuntos

Los iteradores se utilizan a menudo para acceder a 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 al elemento anterior en el 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 en el conjunto cuyo valor es *al menos* `x`, y la función `upper_bound(x)` devuelve un iterador al elemento más pequeño en el conjunto cuyo valor es *mayor que* `x`. En ambas funciones, si dicho elemento no existe, el valor devuelto 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 pasa por todos los casos posibles usando 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 el elemento que corresponde a `it` o el elemento anterior.

4.5 Otras estructuras

Bitset

Un **bitset** es un array cuyos valores son 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 los arrays ordinarios, porque cada elemento en un `bitset` solo usa un bit de memoria. For example, if n bits are stored in an `int` array, $32n$ bits of memory will be used, but a corresponding `bitset` only requires n bits of memory. Además, los valores de un `bitset` pueden manipularse eficientemente usando operadores de bits, lo que hace posible optimizar algoritmos que usan conjuntos de bits.

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 de bits:

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

Deque

Un **deque** es un arreglo dinámico cuyo tamaño puede cambiarse eficientemente en ambos extremos del arreglo. 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 usar 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 un deque es más compleja que la de un vector, y por esta razón, un deque es más lento que un vector. Aún así, tanto agregar como eliminar elementos toman tiempo $O(1)$ en promedio en ambos extremos.

Stack

Un **stack** es una estructura de datos que proporciona dos operaciones en tiempo $O(1)$: agregar un elemento en la parte superior, y quitar un elemento de la parte superior. Sólo 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 en tiempo $O(1)$: agregar un elemento al final de la cola, y eliminar el primer elemento de la cola. Solo es posible acceder al primer y último elemento de una cola.

El siguiente código muestra cómo se puede utilizar 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 admitidas son inserción y, dependiendo del tipo de cola, recuperación y eliminación de ya sea el elemento mínimo o máximo. La inserción y eliminación toman un tiempo de $O(\log n)$, y la recuperación toma un tiempo de $O(1)$.

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 usando una estructura de montículo que es mucho más simple que un árbol binario equilibrado utilizado en un conjunto ordenado.

Por defecto, los elementos en una cola de prioridad en C++ se ordenan en orden decreciente, y es posible encontrar y eliminar el elemento más grande de 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 encontrar y eliminar el 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 forman parte de la biblioteca estándar de C++. Dichas estructuras se denominan estructuras de datos *basadas en políticas*. Para utilizar estas estructuras, se deben agregar las siguientes líneas 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 un arreglo. La definición para valores de tipo `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 un arreglo ordenado. 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 ordenación

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

Consideremos un problema en el que se nos dan dos listas A y B que contienen n elementos cada una. Nuestra tarea es calcular la cantidad 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 directa al problema es recorrer todos los pares de elementos en tiempo $O(n^2)$, 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 de tiempo 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 utilizar la estructura `unordered_set`. Esta es una forma fácil de hacer que el algoritmo sea más eficiente, porque solo tenemos que cambiar la estructura de datos subyacente. La complejidad de tiempo del nuevo algoritmo es $O(n)$.

Algoritmo 3

En lugar de estructuras de datos, podemos usar ordenamiento. 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 de tiempo del ordenamiento es $O(n \log n)$, y el resto del algoritmo funciona en tiempo $O(n)$, entonces la complejidad de tiempo total es $O(n \log n)$.

Comparación de eficiencia

La siguiente tabla muestra cuán 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 utilizan diferentes estructuras de conjunto. 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 ordenamiento. Solo utiliza la mitad del tiempo en comparación con el Algoritmo 2. Curiosamente, la complejidad de tiempo 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 el ordenamiento es un procedimiento simple y se realiza solo una vez al comienzo del Algoritmo 3, y el resto del algoritmo funciona en tiempo lineal. Por otro lado, el Algoritmo 1 mantiene un árbol binario equilibrado complejo durante todo el algoritmo.

Chapter 5

Búsqueda exhaustiva

Búsqueda exhaustiva es un método general que se puede utilizar para resolver casi cualquier problema algorítmico. La idea es generar todas las posibles soluciones al problema mediante fuerza bruta, y luego seleccionar la mejor solución o contar el número de soluciones, dependiendo del problema.

La búsqueda exhaustiva es una buena técnica si hay suficiente tiempo para revisar todas las soluciones, ya que la búsqueda suele ser fácil de implementar y siempre proporciona la respuesta correcta. Si la búsqueda exhaustiva es demasiado lenta, pueden ser necesarias otras técnicas, como algoritmos voraces o programación dinámica.

5.1 Generando 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 aprovechar la representación en bits de los enteros.

Método 1

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

```
void busqueda(int k) {
    if (k == n) {
        // procesar subconjunto
    } else {
        busqueda(k+1);
        subconjunto.push_back(k);
        busqueda(k+1);
        subconjunto.pop_back();
    }
}
```

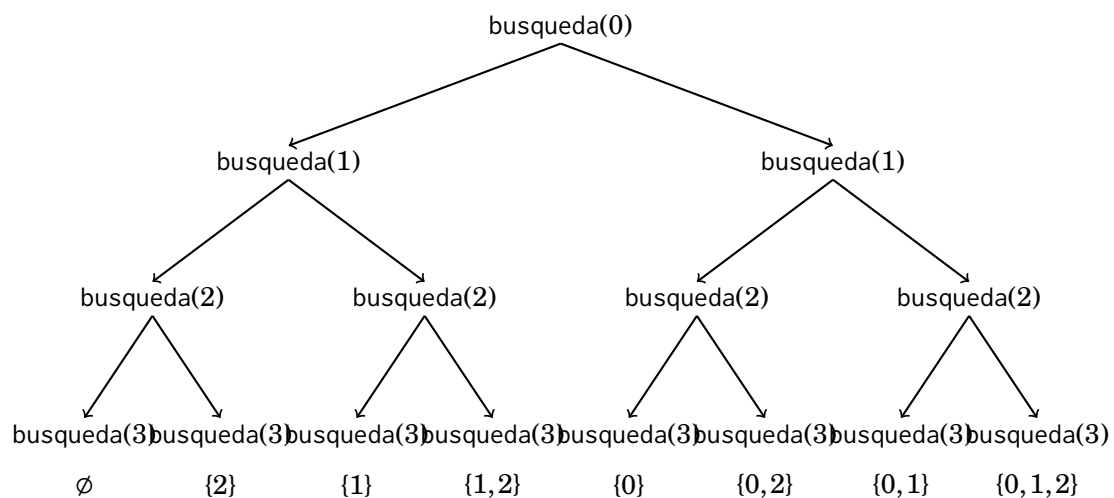
```

    }
}

```

Cuando se llama a la función `busqueda` con el parámetro k , decide si incluir o no el elemento k en el subconjunto, y en ambos casos, luego se llama a sí misma con el parámetro $k + 1$. Sin embargo, si $k = n$, la función se da cuenta de que todos los elementos han sido procesados 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 de bits de los enteros. Cada subconjunto de un conjunto de n elementos puede representarse 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 se incluyen 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 de 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++) {
    // procesar subconjunto
}

```

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

```



```

        if (b&(1<<i)) subconjunto.push_back(i);
    }
}

```

5.2 Generando 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 de forma iterativa.

Método 1

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

```

void buscar() {
    if (permutacion.size() == n) {
        // procesar permutación
    } else {
        for (int i = 0; i < n; i++) {
            if (elegido[i]) continue;
            elegido[i] = true;
            permutacion.push_back(i);
            buscar();
            elegido[i] = false;
            permutacion.pop_back();
        }
    }
}

```

Cada llamada a la función añade un nuevo elemento a permutacion. El array elegido indica cuáles elementos ya están incluidos en la permutación. Si el tamaño de permutacion 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 usar repetidamente una función que construye la siguiente permutación en orden creciente. La biblioteca estándar de C++ contiene la función `next_permutation` que se puede utilizar para esto:

```

vector<int> permutacion;

```

```

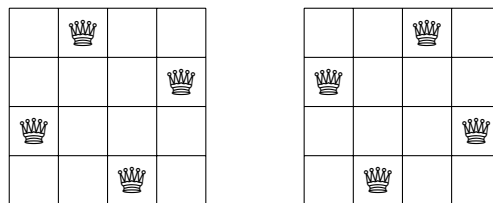
for (int i = 0; i < n; i++) {
    permutacion.push_back(i);
}
do {
    // procesar permutacion
} while (next_permutation(permutacion.begin(), permutacion.end()));

```

5.3 Backtracking

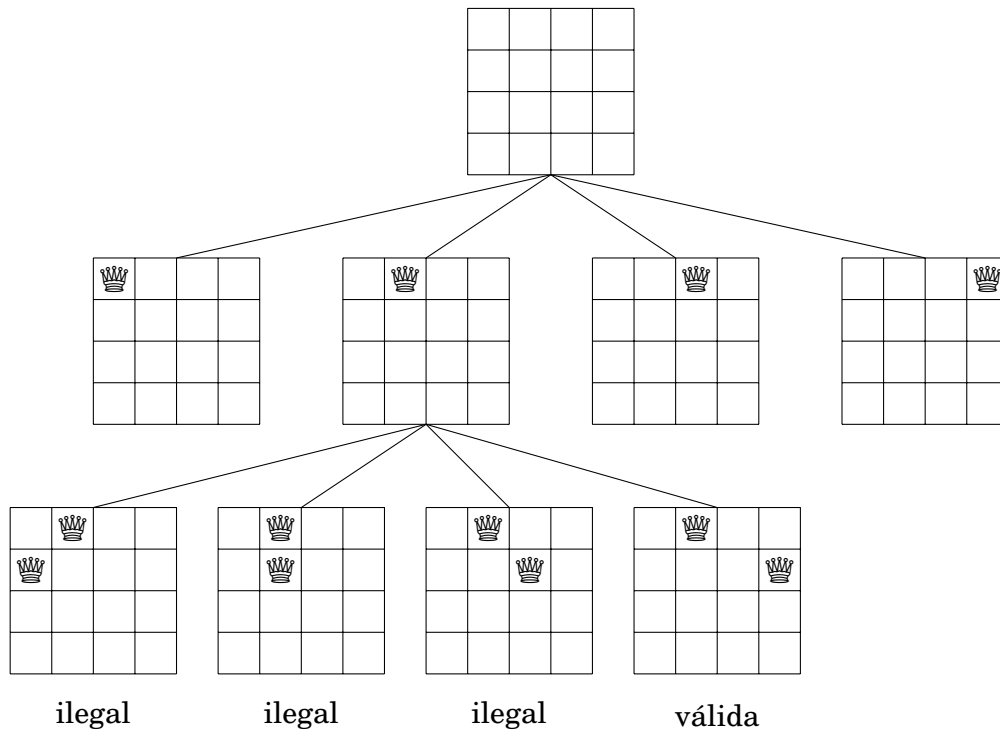
Un algoritmo de **backtracking** comienza con una solución vacía y extiende la solución paso a paso. La búsqueda recursivamente recorre todas las diferentes formas en que se puede construir una solución.

Como ejemplo, considere el problema de calcular el número de formas en que n reinas pueden ser colocadas en un tablero de ajedrez $n \times n$ de tal manera que ninguna reina ataque a las demás. Por ejemplo, cuando $n = 4$, hay dos soluciones posibles:



El problema se puede resolver mediante backtracking colocando reinas en el tablero fila por fila. Más precisamente, se colocará exactamente una reina en cada fila de tal manera que ninguna reina ataque a ninguna de las reinas colocadas anteriormente. Se ha encontrado una solución cuando todas las n reinas han sido colocadas en el tablero.

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



En el nivel más bajo, las tres primeras configuraciones son ilegales, porque las reinas se atacan entre sí. Sin embargo, la cuarta configuración es válida y se puede extender 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 de $n \times n$, y el código calcula el número de soluciones en `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 $y + 1$. Entonces, 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 del tablero de 4×4 están numeradas 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

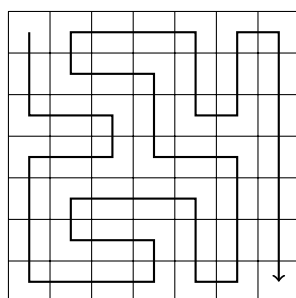
diag2

Sea $q(n)$ el número de formas de colocar n reinas en un tablero de ajedrez de $n \times n$. El algoritmo de backtracking 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 $q(16) = 14772512$ usando el algoritmo anterior ya toma aproximadamente un minuto en una computadora moderna¹.

5.4 Poda de la búsqueda

A menudo podemos optimizar backtracking podando el árbol de búsqueda. La idea es agregar "inteligencia" al algoritmo para que pueda notar lo más rápido posible si una solución parcial no puede extenderse a una solución completa. Dichas 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 tal manera que el camino visite cada cuadrado exactamente una vez. Por ejemplo, en una cuadrícula de 7×7 , hay 111712 de estos caminos. Uno de los caminos es el siguiente:



Nos enfocamos en el caso de 7×7 , porque su nivel de dificultad es apropiado para nuestras necesidades. Comenzamos con un algoritmo de backtracking sencillo, y luego lo optimizamos paso a paso utilizando observaciones sobre 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.

¹No se conoce ninguna forma eficiente de calcular valores más grandes de $q(n)$. El récord actual es $q(27) = 234907967154122528$, calculado en 2016 [55].

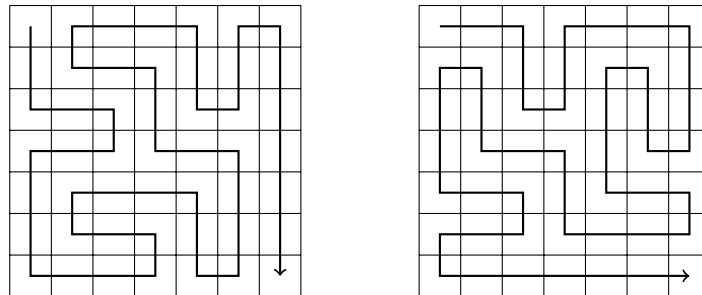
Algoritmo básico

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

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

Optimización 1

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

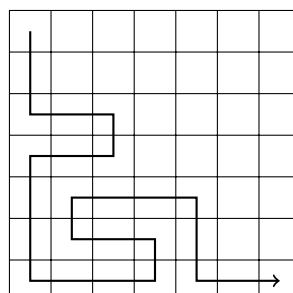


Por lo tanto, podemos decidir que siempre primero nos movemos un paso hacia abajo (o hacia la derecha), y finalmente multiplicamos el número de soluciones por dos.

- 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, está claro que no será posible completar la solución. Un ejemplo de esto es el siguiente camino:

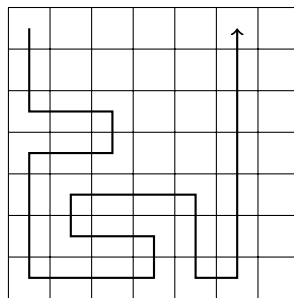


Usando esta observación, podemos terminar la búsqueda inmediatamente si llegamos a la casilla inferior derecha demasiado pronto.

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

Optimización 3

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

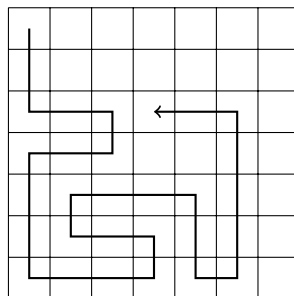


En este caso, ya no podemos visitar todas las casillas, por lo que podemos terminar la búsqueda. Esta optimización es muy útil:

- 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 hacia la izquierda o hacia la derecha, la cuadrícula se divide en dos partes que contienen casillas no visitadas en ambas partes. 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 común en backtracking, porque el árbol de búsqueda suele ser grande y incluso observaciones simples pueden podar efectivamente la búsqueda. Son especialmente útiles las optimizaciones que ocurren 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 donde el espacio de búsqueda se divide en dos partes de aproximadamente igual tamaño. Se realiza una búsqueda separada para ambas partes, y finalmente los resultados de las búsquedas se combinan.

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 donde se nos da una lista de n números y un número x , y queremos averiguar si es posible elegir algunos números de la lista de modo 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 verificar si la suma de alguno de los subconjuntos es x . El tiempo de ejecución de dicho algoritmo es $O(2^n)$, porque hay 2^n subconjuntos. Sin embargo, usando 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 . Correspondientemente, la segunda búsqueda crea una lista S_B a partir de B . Después de esto, basta con verificar 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 $B = [5, 9]$. Luego, creamos las listas $S_A = [0, 2, 4, 6]$ y

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

$S_B = [0, 5, 9, 14]$. 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 manera que su complejidad de tiempo sea $O(2^{n/2})$. Primero, generamos listas *ordenadas* S_A y S_B , lo cual se puede hacer en tiempo $O(2^{n/2})$ utilizando una técnica similar al merge. Después de esto, dado que las listas están ordenadas, podemos verificar 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 siempre tomando una elección que parece ser la mejor en ese momento. Un algoritmo voraz nunca revierte sus elecciones, 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 elecciones localmente óptimas en un algoritmo voraz también deben ser óptimas a nivel global. A menudo es difícil argumentar que un algoritmo voraz funciona.

6.1 Problema de monedas

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 $\text{monedas} = \{c_1, c_2, \dots, c_k\}$, y cada moneda se puede usar tantas veces como queramos. ¿Cuál es el número mínimo de monedas necesario?

Por ejemplo, si las monedas son 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 las 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 haya 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, ¿siempre funciona este algoritmo?

Resulta que si las monedas son 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 de la siguiente manera:

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

Usando estas observaciones, podemos demostrar para cada moneda x que no es posible construir de manera óptima una suma x o cualquier suma mayor utilizando únicamente monedas que sean más pequeñas que x . Por ejemplo, si $x = 100$, la suma óptima más grande usando 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* necesariamente produce una solución óptima.

Podemos demostrar que un algoritmo voraz no funciona mostrando un contraejemplo donde el algoritmo proporciona 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 las monedas se puede resolver utilizando algún algoritmo voraz¹. Sin embargo, como veremos en el Capítulo 7, en algunos casos, el problema general se puede resolver de manera eficiente usando un algoritmo de programación dinámica que siempre proporciona la respuesta correcta.

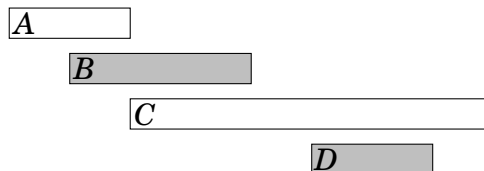
6.2 Planificación

Muchos problemas de planificación se pueden resolver utilizando algoritmos voraces. Un problema clásico es el siguiente: Dado n eventos con sus horas de inicio y finalización, encuentra un horario que incluya la mayor cantidad de eventos posible. No es posible seleccionar un evento parcialmente. Por ejemplo, considera los siguientes eventos:

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

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

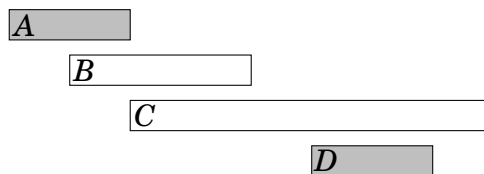
En este caso, el número máximo de eventos es dos. Por ejemplo, podemos seleccionar los eventos *B* y *D* de la siguiente manera:



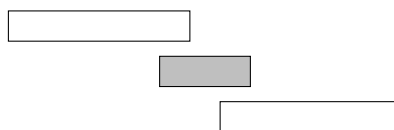
Es posible inventar varios algoritmos voraces para el problema, pero ¿cuál de ellos funciona en todos los casos?

Algoritmo 1

La primera idea es seleccionar eventos lo más *cortos* posible. En el caso de ejemplo, este algoritmo selecciona los siguientes eventos:



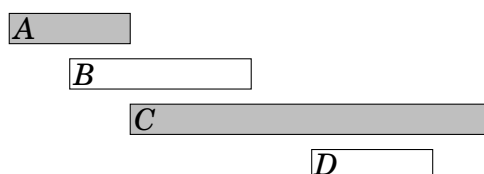
Sin embargo, seleccionar eventos cortos no siempre es una estrategia correcta. Por ejemplo, el algoritmo falla en el siguiente caso:



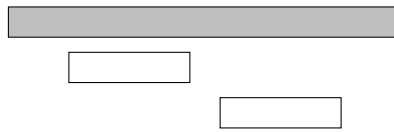
Si seleccionamos el evento corto, solo podemos seleccionar un evento. Sin embargo, sería posible seleccionar ambos eventos largos.

Algoritmo 2

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



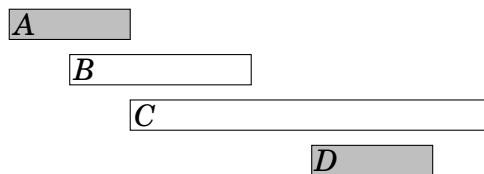
Sin embargo, podemos encontrar un contraejemplo también para este algoritmo. Por ejemplo, en el siguiente caso, el algoritmo solo selecciona un evento:



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 *termina* 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 elección óptima seleccionar primero un evento que termine lo más temprano posible. Después de esto, es una elección 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 a lo sumo un número igual de opciones de cómo podemos seleccionar el siguiente evento. Por lo tanto, seleccionar un evento que termine más tarde nunca puede generar una solución mejor, y el algoritmo voraz es correcto.

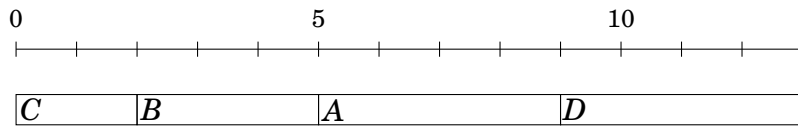
6.3 Tareas y tiempos límite

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

Por ejemplo, supongamos que las tareas son las siguientes:

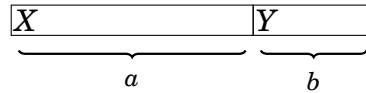
tarea	duración	tiempo límite
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

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

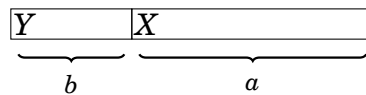


En esta solución, C produce 5 puntos, B produce 0 puntos, A produce -7 puntos y D produce -8 puntos, entonces el puntaje total es -10 .

Sorprendentemente, la solución óptima al problema no depende de los tiempos límite en absoluto, pero una estrategia voraz 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 tras otra de tal manera que la primera tarea tarda más que la segunda tarea, podemos obtener una mejor solución si intercambiamos las tareas. Por ejemplo, considere el siguiente horario:



Aquí $a > b$, entonces deberíamos intercambiar las tareas:



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

6.4 Minimizando 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 enfocamos 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 ordenar. Por ejemplo, la lista $[1, 2, 9, 2, 6]$ se convierte en $[1, 2, 2, 6, 9]$ después de ordenar, así que la mediana es 2.

La mediana es una elección óptima, porque si x es menor que la mediana, la suma se vuelve más pequeña al aumentar x , y si x es mayor que la mediana, la suma se vuelve 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, ambas medianas y 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 el *promedio* de los números. En el ejemplo, el promedio 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 , así 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 el promedio de las raíces $x = s/n$, es decir, el promedio de los números a_1, a_2, \dots, a_n .

6.5 Compresión de datos

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

carácter	palabra clave
A	00
B	01
C	10
D	11

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

000001001011001000

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

carácter	palabra clave
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,

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

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

carácter	palabra clave
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

La **Codificación de Huffman**² 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 el código de cada carácter se puede leer siguiendo un camino desde la raíz hasta el nodo correspondiente. Un movimiento a la izquierda corresponde al bit 0, y un movimiento a la derecha corresponde al bit 1.

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

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

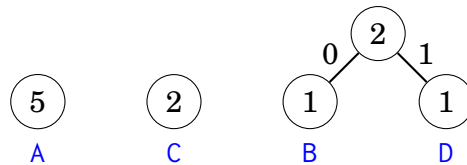
paso, se combinan dos nodos con pesos mínimos creando un nuevo nodo cuyo peso es la suma de los pesos de los nodos originales. El proceso continúa hasta que todos los nodos se hayan combinado.

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:

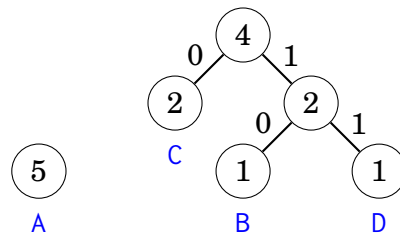


El nodo que representa el carácter A tiene peso 5 porque el carácter A aparece 5 veces en la cadena. Los otros pesos se han calculado de la misma manera.

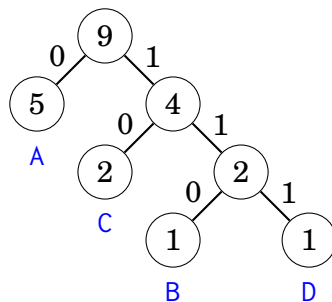
El primer paso es combinar los nodos que corresponden a los caracteres B y D, ambos con peso 1. El resultado es:



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



Finalmente, se combinan los dos nodos restantes:



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 correctitud 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 de forma independiente.

Hay dos usos para la programación dinámica:

- **Encontrar una solución óptima:** Queremos encontrar una solución que sea lo más grande posible 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.

Entender 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 las monedas

Primero nos enfocamos en un problema que ya hemos visto en el Capítulo 6: Dado un conjunto de valores de monedas $\text{monedas} = \{c_1, c_2, \dots, c_k\}$ 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 moneda más grande posible. El algoritmo voraz funciona, por ejemplo, cuando las monedas son las monedas de euro, pero en el caso general, el algoritmo voraz no necesariamente produce una solución óptima.

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

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

Formulación recursiva

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

Denotemos $\text{resolver}(x)$ como 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 $\text{monedas} = \{1, 3, 4\}$, los primeros valores de la función son los siguientes:

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

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

La propiedad esencial de resolver 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, lo cual es un subproblema del problema original. Por supuesto, lo mismo se aplica a las monedas 3 y 4. Así, podemos usar la siguiente fórmula recursiva para calcular el mínimo número de monedas:

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

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

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

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

mínimo número de monedas necesarias para formar una suma x :

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

Primero, si $x < 0$, el valor es ∞ , 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 recorre todas las posibilidades de cómo elegir la primera moneda de la suma.

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

```
int resolver(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int mejor = INF;
    for (auto c : monedas) {
        mejor = min(mejor, resolver(x-c)+1);
    }
    return mejor;
}
```

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

Usando memoización

La idea de la programación dinámica es utilizar **memoización** para calcular eficientemente los valores de una función recursiva. Esto significa que los valores de la función se almacenan en un arreglo 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 del arreglo.

En este problema, usamos los arreglos

```
bool listo[N];
int valor[N];
```

donde `listo[x]` indica si se ha calculado el valor de `resolver(x)`, y si lo está, `valor[x]` contiene este valor. La constante N se ha elegido de manera que todos los valores requeridos quepan en los arreglos.

Ahora la función se puede implementar de manera eficiente de la siguiente forma:

```
int resolver(int x) {
```

```

    if (x < 0) return INF;
    if (x == 0) return 0;
    if (listo[x]) return valor[x];
    int mejor = INF;
    for (auto c : monedas) {
        mejor = min(mejor, resolver(x-c)+1);
    }
    valor[x] = mejor;
    listo[x] = true;
    return mejor;
}

```

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

Esta función funciona de manera eficiente, porque la respuesta para cada parámetro x se calcula recursivamente solo una vez. Después de que se haya almacenado un valor de `resolver(x)` en `valor[x]`, se puede recuperar de manera eficiente cada vez que la función sea llamada nuevamente con el parámetro x . La complejidad en tiempo del algoritmo es $O(nk)$, donde n es la suma objetivo y k es el número de monedas.

También podemos construir *iterativamente* el arreglo `valor` usando un bucle que simplemente calcule todos los valores de `resolver` para los parámetros $0 \dots n$:

```

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

```

De hecho, la mayoría de los programadores competitivos prefieren esta implementación, porque es más corta y tiene factores constantes más bajos. A partir de ahora, también utilizamos implementaciones iterativas en nuestros ejemplos. Aun 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 encontrar el valor de una solución óptima y dar un ejemplo de cómo se puede construir dicha solución. En el problema de las monedas, por ejemplo, podemos declarar otro arreglo que indique para cada suma de dinero la primera moneda en una solución óptima:

```
int primero[N];
```

Luego, podemos modificar el algoritmo de la siguiente manera:

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

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

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

Contando el número de soluciones

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

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

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

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

Entonces, la función recursiva general es la siguiente:

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

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 $\text{resolver}(x - c)$ donde c está en monedas.

El siguiente código construye un array conteo tal que $\text{conteo}[x]$ es igual al valor de $\text{resolver}(x)$ para $0 \leq x \leq n$:

```

conteo[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : monedas) {
        if (x-c >= 0) {
            conteo[x] += conteo[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 modificando el código de manera que todos los cálculos se realicen módulo m . En el código anterior, basta con agregar la línea

```

conteo[x] %= m;

```

después de la línea

```

conteo[x] += conteo[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 revisaremos 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 en 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



Denotemos $\text{largo}(k)$ como el largo de la subsecuencia creciente más larga que termina en la posición k . Así, si calculamos todos los valores de $\text{largo}(k)$ donde $0 \leq k \leq n-1$, descubriremos el largo de la subsecuencia creciente más larga. Por ejemplo, los valores de la función para el arreglo anterior son los siguientes:

$\text{largo}(0)$	$=$	1
$\text{largo}(1)$	$=$	1
$\text{largo}(2)$	$=$	2
$\text{largo}(3)$	$=$	1
$\text{largo}(4)$	$=$	3
$\text{largo}(5)$	$=$	2
$\text{largo}(6)$	$=$	4
$\text{largo}(7)$	$=$	2

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

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

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

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

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

7.3 Caminos en una cuadrícula

Nuestro siguiente problema es encontrar un camino desde la esquina superior izquierda hasta la esquina inferior derecha de una cuadrícula de $n \times n$, de tal manera que sólo nos movemos hacia abajo y hacia la derecha. Cada cuadro contiene un número entero positivo, y el camino debe construirse de tal manera 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

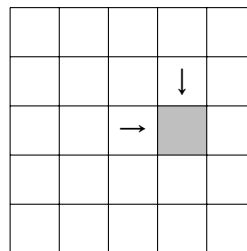
La suma de los valores en el camino es 67, y esta es la suma más grande posible en un camino desde la esquina superior izquierda hasta la esquina inferior derecha.

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

Podemos calcular recursivamente las sumas de la siguiente manera:

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

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



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

Dado que la función suma tiene dos parámetros, el array de programación dinámica también tiene dos dimensiones. Por ejemplo, podemos usar un array

```
int suma[N][N];
```

y calcular la suma como sigue:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        suma[y][x] = max(suma[y][x-1], suma[y-1][x]) + valor[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 ciertas propiedades. Los problemas de la mochila a menudo se pueden resolver usando programación dinámica.

En esta sección, nos enfocamos en el siguiente problema: Dada una lista de pesos $[w_1, w_2, \dots, w_n]$, determine todas las sumas que se pueden construir usando 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 enfocamos en subproblemas donde solo usamos los primeros k pesos para construir sumas. Sea $\text{posible}(x, k) = \text{verdadero}$ si podemos construir una suma x usando los primeros k pesos, y de lo contrario $\text{posible}(x, k) = \text{falso}$. Los valores de la función se pueden calcular recursivamente de la siguiente manera:

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

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,

$$\text{posible}(x, 0) = \begin{cases} \text{verdadero} & x = 0 \\ \text{falso} & x \neq 0 \end{cases}$$

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, $\text{posible}(x, n)$ nos indica si podemos construir una suma x utilizando *todos* los pesos.

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

```

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

```

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

```

posible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (posible[x]) posible[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 pesos el valor máximo suma de un subconjunto.

7.5 Distancia de edición

La **distancia de edición** o **distancia de Levenshtein**¹ 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 número mínimo posible de operaciones, porque está claro que una sola operación no es suficiente.

Supongamos 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 $distancia(a, b)$ que proporciona la distancia

¹La distancia lleva el nombre de V. I. Levenshtein, quien la estudió en relación con los códigos binarios [49].

de edición entre los prefijos $x[0 \dots a]$ y $y[0 \dots b]$. Por lo tanto, usando esta función, la distancia de edición entre x y y es igual a $\text{distancia}(n-1, m-1)$.

Podemos calcular los valores de distancia de la siguiente manera:

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

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

- $\text{distancia}(a, b-1)$: insertar un carácter al final de x
- $\text{distancia}(a-1, b)$: eliminar el último carácter de x
- $\text{distancia}(a-1, b-1)$: 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 coincidir los últimos caracteres sin editar, y en caso contrario se necesita una operación de edición (modificar).

La siguiente tabla muestra los valores de distancia en el caso de ejemplo:

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

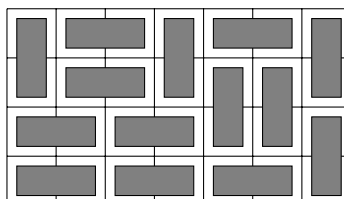
La esquina inferior derecha de la tabla nos indica que la distancia de edición entre LOVE y MOVIE es 2. La tabla también muestra cómo construir la secuencia más corta de operaciones de edición. En este caso, el camino es el siguiente:

		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 una unidad mayor que la distancia de edición entre LOV y MOV, etc.

7.6 Contando formas de colocar baldosas

A veces los estados de una solución de programación dinámica son más complejos que combinaciones fijas de números. Como ejemplo, considera el problema de calcular el número de formas distintas de llenar una cuadrícula de $n \times m$ usando baldosas de tamaño 1×2 y 2×1 . Por ejemplo, una solución válida para la cuadrícula de 4×7 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 formas de construir una solución para las filas $1 \dots k$ de la cuadrícula de tal manera que la cadena x corresponda a la fila k . Es posible utilizar programación dinámica aquí, porque el estado de una fila está limitado sólo 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 filas distintas es como máximo 4^m . Por lo tanto, la complejidad del tiempo de la solución es $O(n4^{2m})$ porque podemos recorrer los $O(4^m)$ 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 de modo que el lado más corto tenga longitud m , porque el factor 4^{2m} domina la complejidad del tiempo.

Es posible hacer que la solución sea más eficiente utilizando una representación más compacta para las filas. Resulta que es suficiente saber en qué columnas de la fila anterior se encuentra el cuadrado superior de una baldosa vertical. Así, podemos representar una fila usando únicamente caracteres \sqcap y \sqcup , donde \sqcup es una combinación de caracteres \sqcup , \sqsubset y \sqsupset . Usando esta representación, hay solamente 2^m filas distintas y la complejidad de tiempo es $O(n2^{2m})$.

Como nota final, también hay una sorprendente fórmula directa para calcular el número de forma de colocar baldosas²:

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

Esta fórmula es muy eficiente, ya que calcula el número de baldosas 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 de manera precisa.

²Sorprendentemente, 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 realizan los bucles. Sin embargo, a veces un análisis directo no proporciona una imagen real de la eficiencia del algoritmo.

El **Análisis amortizado** se puede utilizar para analizar algoritmos que contienen operaciones cuya complejidad temporal varía. La idea es estimar el tiempo total utilizado en todas estas 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 del arreglo. Ambos punteros pueden moverse en una dirección solamente, lo que garantiza que el algoritmo funcione de manera eficiente. A continuación, discutimos dos problemas que se pueden resolver utilizando el método de dos punteros.

Suma de subarreglo

Como primer ejemplo, consideremos 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 $O(n)$ tiempo utilizando el método de dos punteros. La idea es mantener punteros que apuntan al primer y último valor de un subarreglo. En cada turno, el puntero izquierdo se mueve un paso a la

derecha, y el puntero derecho se mueve a 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.

Por 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 a 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
---	---	---	---	---	---	---	---

De nuevo, el puntero izquierdo se mueve un paso a la derecha, y esta vez el puntero derecho se mueve tres pasos a la derecha. La suma del subarreglo es $2 + 5 + 1 = 8$, entonces 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 el puntero derecho se mueva. Si bien no hay un límite superior útil sobre cuántos pasos puede moverse el puntero en un *único* 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 usando el método de dos punteros es el siguiente problema, también conocido como el problema **2SUM**: dado un arreglo de n números y una suma objetivo x , encuentre dos valores del arreglo tales que su suma sea x , o informe que no existen tales valores.

Para resolver el problema, primero ordenamos los valores del arreglo en orden creciente. Después de eso, iteramos a través del arreglo usando dos punteros. El puntero izquierdo comienza en el primer valor y se mueve un paso a la derecha en cada turno. 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 sea como máximo x . Si la suma es exactamente x , se ha encontrado una solución.

Por ejemplo, considere el siguiente arreglo 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 nuevamente. El puntero derecho no se mueve y se encuentra una solución $5 + 7 = 12$.

1	4	5	6	7	9	9	10
		↑		↑			

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

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

Un problema más difícil es el **problema 3SUM** que pregunta por encontrar *tres* valores del arreglo cuya suma sea x . Usando la idea del algoritmo anterior, este problema se puede resolver en tiempo $O(n^2)$ ¹. ¿Puedes ver cómo?

8.2 Elementos menores más cercanos

El análisis amortizado se utiliza a menudo para estimar el número de operaciones realizadas en una estructura de datos. Las operaciones pueden distribuirse de manera desigual de modo que la mayoría de las operaciones ocurran durante una fase específica del algoritmo, pero el número total de operaciones está limitado.

Como ejemplo, considere el problema de encontrar para cada elemento del arreglo el **elemento menor más cercano**, es decir, el primer elemento menor que precede al elemento en el arreglo. Es posible que no exista tal elemento, en cuyo caso el algoritmo debe informar esto. A continuación, veremos cómo el problema puede ser resuelto de manera eficiente utilizando una estructura de pila.

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

Recorremos el arreglo de izquierda a derecha y mantenemos una pila de elementos del arreglo. En cada posición del arreglo, 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 menor más 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 el siguiente arreglo:

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 menor más cercano de 4 es 3, y el elemento menor más 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 menor más cercano es 1:

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

1 → 2

Luego, el elemento 5 es mayor que el elemento 2, así que se agregará a la pila, y su elemento menor más 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 son eliminados 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 el número total de operaciones en la pila. Si el elemento actual es más grande 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. Aun 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 causa $O(1)$ operaciones en la pila, y el algoritmo funciona en tiempo $O(n)$.

8.3 Mínimo de ventana deslizante

Una **ventana deslizante** es un subarreglo de tamaño constante que se mueve de izquierda a derecha a través del arreglo. En cada posición de la ventana, queremos calcular cierta información sobre los elementos dentro de la ventana. En esta sección, nos enfocamos en el problema de mantener el **mínimo de ventana deslizante**, lo que significa que debemos informar el valor más pequeño dentro de cada ventana.

El mínimo de ventana deslizante se puede calcular usando una idea similar a la que utilizamos para calcular los elementos menores más cercanos. 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 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 quede vacía. También eliminamos el primer elemento de la cola si ya no está dentro de la ventana. Finalmente, agregamos el nuevo elemento de la ventana al final de la cola.

Como ejemplo, considera el siguiente arreglo:

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 agrega 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 nuevamente, y el elemento más pequeño 1 ya no pertenece a la ventana. Por lo tanto, se elimina de la cola y el más pequeño valor es ahora 3. También se agrega el nuevo elemento 4 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, se eliminan todos los elementos de la cola y solo contendrá el elemento 1:

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

1

Finalmente, la ventana llega a su última posición. El elemento 2 se agrega 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 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 un subarreglo de un arreglo. Las consultas de rango típicas son:

- $\text{suma}_q(a, b)$: calcular la suma de 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, considera el rango $[3, 6]$ en el siguiente arreglo:

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

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

Una forma simple de procesar consultas de rango es usar un bucle que recorre todos los valores del arreglo en el rango. Por ejemplo, la siguiente función se puede usar para procesar consultas de suma en un arreglo:

```
int suma(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 del arreglo. Por lo tanto, podemos procesar q consultas en tiempo $O(nq)$ 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 arreglos estáticos

Primero nos enfocamos en una situación donde el arreglo es *estático*, es decir, los valores del arreglo nunca se actualizan entre las consultas. En este caso, basta con construir una estructura de datos estática que nos indique la respuesta para cualquier consulta posible.

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

El arreglo de suma de prefijos correspondiente es el siguiente:

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

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

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

Definiendo $\text{suma}_q(0, -1) = 0$, 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 $\text{suma}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Esta suma se puede calcular a partir de dos valores del arreglo de suma de prefijos:

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

Así, $\text{suma}_q(3, 6) = \text{suma}_q(0, 6) - \text{suma}_q(0, 2) = 27 - 8 = 19$.

También es posible generalizar esta idea a dimensiones superiores. Por ejemplo, podemos construir un arreglo de suma de prefijos bidimensional que se puede utilizar para calcular la suma de cualquier subarreglo rectangular en tiempo $O(1)$. Cada suma en dicho 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 de mínimo

Las consultas de mínimo son más difíciles de procesar que las consultas de suma. Sin embargo, hay un método de preprocesamiento bastante simple de tiempo $O(n \log n)$ después del cual podemos responder a cualquier consulta de mínimo en tiempo $O(1)$ ¹. Tenga en cuenta que, dado que las consultas de mínimo y máximo se pueden procesar de manera similar, podemos centrarnos en las consultas de mínimo.

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						

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 un tiempo de $O(n \log n)$.

Después de esto, cualquier valor de $\min_q(a, b)$ se puede calcular en tiempo $O(1)$ como el mínimo de dos valores precalculados. Sea k la mayor potencia de dos que no exceda $b - a + 1$. 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 $[a, b]$ se representa como la unión de los rangos $[a, a + k - 1]$ y $[b - k + 1, b]$, ambos de longitud k .

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

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

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

La longitud del rango es 6, y la mayor potencia de dos que no exceda 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 binario indexado

Un **árbol binario indexado** o un **árbol de Fenwick**² puede verse como una variante dinámica de un arreglo de suma de prefijos. Admite dos operaciones en tiempo $O(\log n)$ en un arreglo: procesar una consulta de suma de rango y actualizar un valor.

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

Estructura

Aunque el nombre de la estructura es un árbol binario indexado, generalmente se representa como un arreglo. En esta sección asumimos que todos los arreglos tienen índice base uno, ya que facilita la implementación.

Sea $p(k)$ la mayor potencia de dos que divide a k . Almacenamos un árbol binario indexado como un arreglo *arbol* tal que

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

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

Por ejemplo, considere el siguiente arreglo:

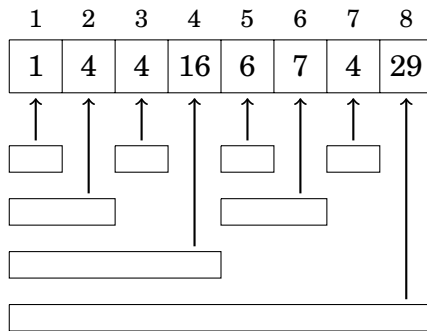
1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

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

El árbol binario indexado correspondiente es el siguiente:

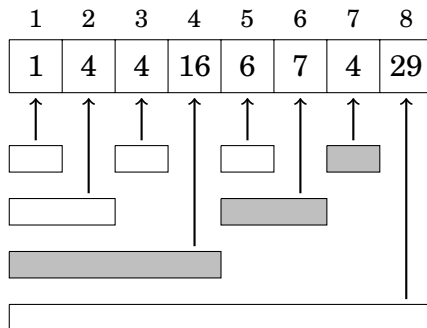
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

La siguiente imagen muestra de manera más clara cómo cada valor en el árbol binario indexado corresponde a un rango en el arreglo original:



Usando un árbol binario indexado, cualquier valor de $\text{suma}_q(1, k)$ se puede calcular en $O(\log n)$ tiempo, 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 $[1, 7]$ consta de los siguientes rangos:



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

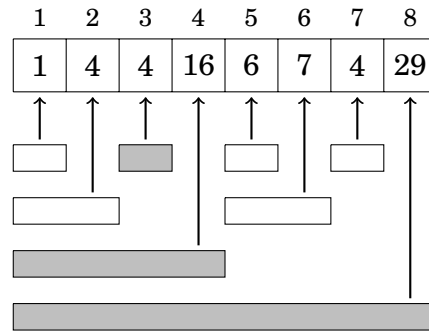
$$\text{suma}_q(1, 7) = \text{suma}_q(1, 4) + \text{suma}_q(5, 6) + \text{suma}_q(7, 7) = 16 + 7 + 4 = 27$$

Para calcular el valor de $\text{suma}_q(a, b)$ donde $a > 1$, podemos usar el mismo truco que utilizamos con arreglos de sumas de prefijos:

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

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

Luego, después de actualizar un valor en el arreglo original, varios valores en el árbol binario indexado deben ser actualizados. Por ejemplo, si el valor en la posición 3 cambia, las sumas de los siguientes rangos cambian:



Dado que cada elemento del arreglo pertenece a $O(\log n)$ rangos en el árbol binario indexado, basta con actualizar $O(\log n)$ valores en el árbol.

Implementación

Las operaciones de un árbol binario indexado pueden ser implementadas de manera eficiente usando operaciones de bits. El hecho clave necesario es que podemos calcular cualquier valor de $p(k)$ usando la fórmula

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

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

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

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

```
void agregar(int k, int x) {
    while (k <= n) {
        arbol[k] += x;
        k += k&-k;
    }
}
```

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

9.3 Árbol de segmentos

Un **árbol de segmentos**³ es una estructura de datos que soporta dos operaciones: procesar una consulta de rango y actualizar un valor del arreglo. Los árboles de segmentos pueden soportar consultas de suma, consultas de mínimo y máximo, y muchas otras consultas de modo que ambas operaciones funcionen en $O(\log n)$ tiempo.

En comparación con un árbol binario indexado, la ventaja de un árbol de segmentos es que es una estructura de datos más general. Mientras que los árboles binarios indexados solo admiten consultas de suma⁴, 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

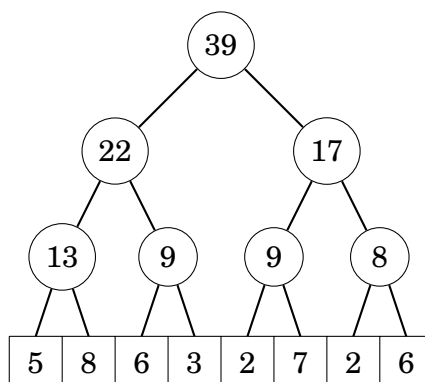
Un árbol de segmentos es un árbol binario tal que los nodos en el nivel inferior del árbol corresponden a los elementos del arreglo, y los otros nodos contienen información necesaria para procesar consultas de rango.

En esta sección, asumimos que el tamaño del arreglo es una potencia de dos y se utiliza indexación basada 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.

Primero discutiremos 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:



³La implementación de abajo hacia arriba en este capítulo corresponde a la que aparece en [62]. Estructuras similares se utilizaron a finales de la década de 1970 para resolver problemas geométricos [9].

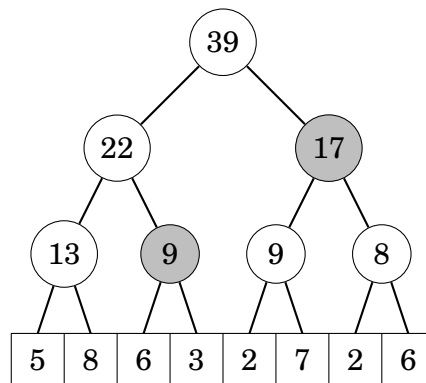
⁴De hecho, utilizando *dos* árboles binarios indexados es posible soportar consultas de mínimo [16], pero esto es más complicado que usar un árbol de segmentos.

Cada nodo interno del árbol corresponde a un rango de 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 correspondiente, y se puede calcular como la suma de los valores de su nodo hijo izquierdo y derecho.

Resulta que cualquier rango $[a, b]$ puede dividirse 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 siguientes dos nodos del árbol corresponden al rango:

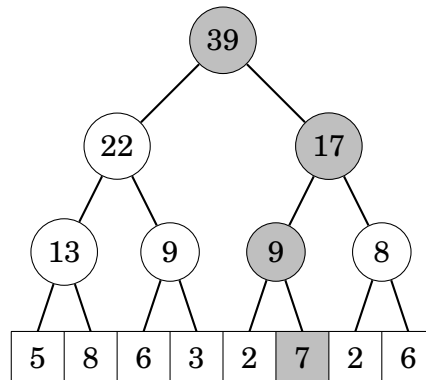


Por lo tanto, otra forma de calcular la suma es $9 + 17 = 26$.

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

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

La siguiente imagen muestra qué nodos del árbol cambian si cambia el valor 7 del arreglo:

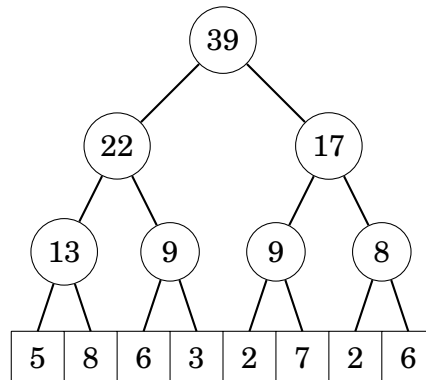


El camino de abajo hacia arriba siempre consta de $O(\log n)$ nodos, por lo que cada actualización cambia $O(\log n)$ nodos en el árbol.

Implementación

Almacenamos un árbol de segmentos como un arreglo de $2n$ elementos donde n es el tamaño de el arreglo original y una potencia de dos. Los nodos del árbol se almacenan de arriba hacia abajo: `arbol[1]` es el nodo superior, `arbol[2]` y `arbol[3]` son sus hijos, y así sucesivamente. Finalmente, los valores desde `arbol[n]` hasta `arbol[2n - 1]` corresponden a los valores del arreglo 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

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

```
int suma(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += arbol[a++];
        if (b%2 == 0) s += arbol[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

La función mantiene un rango que inicialmente es $[a + n, b + n]$. 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 suman a la suma.

La siguiente función incrementa el valor del arreglo en la posición k en x :

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

```

    }
}

```

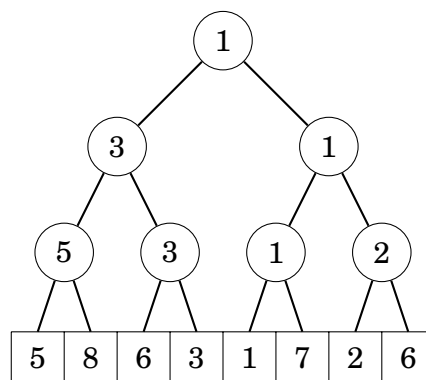
Primero, la función actualiza el valor en el nivel inferior del árbol. Después de esto, la función actualiza los valores de todos los nodos internos del árbol, hasta que llega al nodo superior del árbol.

Ambas funciones trabajan en tiempo $O(\log n)$, porque un árbol de segmentos de n elementos consiste en $O(\log n)$ 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 eficientemente las respuestas. Ejemplos de tales consultas son mínimo y máximo, máximo común divisor, y operaciones de bits and, or y xor.

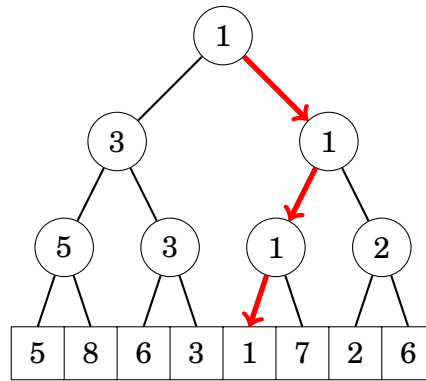
Por ejemplo, el siguiente árbol de segmentos admite consultas de mínimos:



En este caso, cada nodo del árbol contiene el valor más pequeño en el rango correspondiente del arreglo. El nodo superior del árbol contiene el valor más pequeño en todo el arreglo. Las operaciones se pueden implementar como antes, pero en lugar de sumas, se calculan los mínimos.

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

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



9.4 Técnicas adicionales

Compresión de índices

Una limitación en las estructuras de datos que se basan en un arreglo es que los elementos se indexan usando enteros consecutivos. Se presentan dificultades cuando se necesitan índices grandes. Por ejemplo, si deseamos usar el índice 10^9 , el arreglo debería contener 10^9 elementos, lo que requeriría demasiada memoria.

Sin embargo, a menudo podemos eludir esta limitación utilizando **compresión de índices**, donde los índices originales son reemplazados con índices 1, 2, 3, etc. Esto se puede hacer si conocemos todos los índices necesarios 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 individuales. Consideremos ahora una situación opuesta, donde debemos actualizar rangos y recuperar valores individuales. 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 un **arreglo de diferencias** cuyos valores indican las diferencias entre valores consecutivos en el arreglo original. Por lo tanto, el arreglo original es el arreglo de suma de prefijos del arreglo de diferencias. Por ejemplo, considere el siguiente arreglo:

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

El arreglo de diferencias para el arreglo anterior es el 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 en el arreglo original corresponde a la suma $3 - 2 + 4 - 3 = 2$ en el arreglo de diferencias.

La ventaja del arreglo de diferencias es que podemos actualizar un rango en el arreglo original cambiando solo dos elementos en el arreglo de diferencias. Por ejemplo, si queremos aumentar los valores del arreglo original entre las posiciones 1 y 4 en 5, basta con aumentar el valor del arreglo 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 $[a, b]$ por 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 individuales y procesar consultas de suma, así que podemos usar un árbol binario indexado o un árbol de segmentos.

Un problema más difícil es admitir tanto consultas de rango como 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 se pueden usar números no negativos, 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 $2^{32} - 1$.

Existe una conexión entre las 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 número sin signo $y = 2^{32} - 43$:

```
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, considere 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`, entonces el siguiente número después de $2^{31} - 1$ es -2^{31} .

10.2 Operaciones de bits

Operación and

La operación **and** x & y produce un número que tiene bits uno en las posiciones donde ambos x y y tienen bits uno. Por ejemplo, $22 \text{ \& } 26 = 18$, porque

$$\begin{array}{rcl} & 10110 & (22) \\ \& & 11010 & (26) \\ \hline = & 10010 & (18) \end{array}$$

Usando la operación `and`, podemos verificar si un número x es par porque $x \& 1 = 0$ si x es par, y $x \& 1 = 1$ si x es impar. Más en general, x es divisible por 2^k exactamente cuando $x \& (2^k - 1) = 0$.

Operación or

La operación **or** $x \mid y$ produce un número que tiene bits uno en posiciones donde al menos uno de x y y tienen bits uno. Por ejemplo, $22 \mid 26 = 30$, porque

$$\begin{array}{r} 10110 \quad (22) \\ \wedge \quad 11010 \quad (26) \\ \hline = \quad 01100 \quad (12) \end{array}$$

Operación not

La operación **not** $\sim x$ produce un número donde todos los bits de x han sido invertidos. La fórmula $\sim x = -x - 1$ se mantiene, por ejemplo, $\sim 29 = -30$.

El resultado de la operación not a nivel de bit 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 $x \ll k$ añade k bits de cero al número, y el desplazamiento de bits a la derecha $x \gg k$ elimina los últimos k 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 de uno en la posición k y todos los demás bits son cero, por lo que podemos usar tales números para acceder a bits individuales de los números. En particular, el bit k 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 individuales de números usando ideas similares. Por ejemplo, la fórmula $x \mid (1 \ll k)$ establece el bit k de x en uno, la fórmula $x \& \sim(1 \ll k)$ establece el bit k de x en cero, y la fórmula $x \wedge (1 \ll k)$ invierte el bit k 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. Además, se debe tener 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)`: la cantidad de ceros al principio del número
- `__builtin_ctz(x)`: la cantidad de ceros al final del número
- `__builtin_popcount(x)`: la cantidad de unos en el número
- `__builtin_parity(x)`: la paridad (par o impar) de la cantidad de unos

Las funciones se pueden utilizar de la siguiente manera:

```
int x = 5328; // 000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

10.3 Representación de conjuntos

Cada subconjunto de un conjunto $\{0, 1, 2, \dots, n - 1\}$ se puede representar como un número entero de n bits cuyos bits uno indican qué elementos pertenecen al subconjunto. Esta es una forma eficiente de representar conjuntos, porque cada elemento requiere solo un bit de memoria, y las operaciones de conjuntos 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 una variable `int 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 conjuntos

Las operaciones de conjuntos se pueden implementar como operaciones de bits de la siguiente manera:

	sintaxis de conjuntos	sintaxis de bits
intersección	$a \cap b$	$a \& b$
unión	$a \cup b$	$a \mid b$
complemento	\bar{a}	$\sim a$
diferencia	$a \setminus b$	$a \& (\sim b)$

Por ejemplo, el siguiente código primero construye los conjuntos $x = \{1, 3, 4, 8\}$ y $y = \{3, 6, 8, 9\}$, 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
```

Iteración a través de subconjuntos

El siguiente código recorre los subconjuntos de $\{0, 1, \dots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // procesar subconjunto b
}
```

El siguiente código recorre los subconjuntos con exactamente k elementos:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // procesar subconjunto b
    }
}
```

```
}
}
```

El siguiente código recorre los subconjuntos de un conjunto x :

```
int b = 0;
do {
    // procesar subconjunto b
} while (b=(b-x)&x);
```

10.4 Optimizaciones de bits

Muchos algoritmos se pueden optimizar utilizando operaciones de bits. Estas optimizaciones no cambian la complejidad del tiempo 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: Dado una lista de n cadenas de bits, cada una de longitud k , calcule la distancia mínima de Hamming 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 directa de resolver el problema es recorrer todos los pares de cadenas y calcular sus distancias de Hamming, lo que produce un algoritmo de tiempo $O(n^2k)$. La siguiente función se puede utilizar para calcular 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 almacenar las cadenas como valores `int` y usar la siguiente función para calcular distancias:

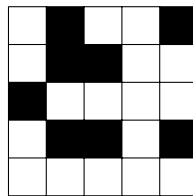
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

En la función anterior, la operación xor construye una cadena de bits que tiene unos en posiciones donde a y b difieren. Luego, se calcula el número de bits usando la función `__builtin_popcount`.

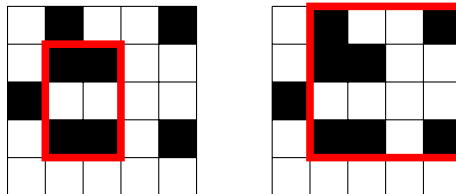
Para comparar las implementaciones, generamos una lista de 10000 cadenas de bits aleatorias de longitud 30. Usando el primer enfoque, la búsqueda tomó 13.5 segundos, y después de la optimización de bits, sólo tomó 0.5 segundos. Por lo tanto, el código optimizado de bits fue casi 30 veces más rápido que el código original.

Contando subcuadrículas

Como otro ejemplo, considere el siguiente problema: Dado una cuadrícula de $n \times n$ cuyo cada cuadrado es negro (1) o blanco (0), calcule el número de subcuadrículas cuyas cuatro esquinas son negras. Por ejemplo, la cuadrícula



contiene dos de estas subcuadrículas:



Existe un algoritmo de tiempo $O(n^3)$ para resolver el problema: recorre todos los $O(n^2)$ pares de filas y para cada par (a, b) calcula el número de columnas que contienen un cuadrado negro en ambas filas en tiempo $O(n)$. El siguiente código asume que `color[y][x]` denota el color en la fila y y columna x :

```
int conteo = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) conteo++;
}
```

Entonces, esas columnas representan $\text{conteo}(\text{conteo} - 1)/2$ subcuadrículas con esquinas negras, porque podemos elegir cualquiera de los dos para formar una subcuadrícula.

Para optimizar este algoritmo, dividimos la cuadrícula en bloques de columnas de tal manera que cada bloque consta de 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 de bits. En el siguiente código, `color[y][k]` representa un bloque de N colores como bits.

```
int conteo = 0;
for (int i = 0; i <= n/N; i++) {
    conteo += __builtin_popcount(color[a][i]&color[b][i]);
}
```

El algoritmo resultante funciona en tiempo $O(n^3/N)$.

Generamos una cuadrícula aleatoria de tamaño 2500×2500 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 $N = 32$ (números `int`) y 1.7 segundos con $N = 64$ (números `long long`).

10.5 Programación dinámica

Las operaciones de bits proporcionan una forma eficiente y conveniente de implementar algoritmos de programación dinámica cuyos estados contienen subconjuntos de elementos, porque tales estados se pueden almacenar como enteros. A continuación, discutimos ejemplos de combinación de operaciones de bits 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 se nos permite comprar un producto como máximo 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
product 0	6	9	5	2	8	9	1	6
product 1	8	2	6	2	7	5	7	2
product 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
product 0	6	9	5	2	8	9	1	6
product 1	8	2	6	2	7	5	7	2
product 2	5	3	9	7	3	5	1	4

Deje que `precio[x][d]` denote el precio del producto x en el día d . Por ejemplo, en el escenario anterior `precio[2][3] = 7`. Luego, que `total(S, d)` denote el precio

total mínimo para comprar un subconjunto S de productos en 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 $\text{total}(\{x\}, 0) = \text{precio}[x][0]$, porque hay una forma de comprar un producto en el primer día. Luego, se puede utilizar la siguiente recurrencia:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{precio}[x][d]))$$

Esto significa que no compramos ningún producto en el día d o compramos un producto x que pertenece a S . En este último caso, quitamos x de S y añadimos 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 adecuadamente grandes. La primera dimensión de la matriz corresponde a una representación de bits 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] = precio[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] + precio[x][d]);
            }
        }
    }
}
```

La complejidad temporal del algoritmo es $O(n2^k k)$.

De permutaciones a subconjuntos

Usando programación dinámica, a menudo es posible cambiar una iteración sobre permutaciones a una iteración sobre subconjuntos¹. 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 $2^n \approx 10^6$. Así, para

¹Esta técnica fue introducida en 1962 por M. Held y R. M. Karp [34].

ciertos valores de n , podemos recorrer eficientemente los subconjuntos 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 última planta. ¿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 utilizar programación dinámica para obtener un algoritmo más eficiente en 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 $\text{peso}[p]$ el peso de la persona p . Definimos dos funciones: $\text{viajes}(S)$ es el número mínimo de viajes para un subconjunto S , y $\text{ultimo}(S)$ es el peso mínimo del último viaje. Por ejemplo, en el escenario anterior

$$\text{viajes}(\{1, 3, 4\}) = 2 \quad \text{y} \quad \text{ultimo}(\{1, 3, 4\}) = 5,$$

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 $\text{viajes}(\{0 \dots n-1\})$.

Podemos calcular los valores de las funciones de forma recursiva y luego aplicar programación dinámica. La idea es recorrer todas las personas que pertenecen a S y escoger de manera óptima la última persona p que entra en el ascensor. Cada una de estas elecciones genera un subproblema para un subconjunto más pequeño de personas. Si $\text{ultimo}(S \setminus p) + \text{peso}[p] \leq x$, podemos agregar p al último viaje. De lo contrario, debemos reservar un nuevo viaje que inicialmente solo contiene p .

Para implementar programación dinámica, declaramos un array

```
pair<int,int> mejor[1<<N];
```

que contiene para cada subconjunto S un par $(\text{viajes}(S), \text{ultimo}(S))$. Establecemos el valor para un grupo vacío de la siguiente manera:

```
mejor[0] = {1,0};
```

Entonces, podemos llenar el array de la siguiente manera:

```

for (int s = 1; s < (1<<n); s++) {
    // valor inicial: se necesitan n+1 viajes
    mejor[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto opcion = mejor[s^(1<<p)];
            if (opcion.second+peso[p] <= x) {
                // agregar p a un viaje existente
                opcion.second += peso[p];
            } else {
                // reservar un nuevo viaje para p
                opcion.first++;
                opcion.second = peso[p];
            }
            mejor[s] = min(mejor[s], opcion);
        }
    }
}

```

Tenga en cuenta que el bucle anterior garantiza que para cualquier dos subconjuntos S_1 y S_2 tal que $S_1 \subset S_2$, procesamos S_1 antes de S_2 . Por lo tanto, los valores de 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 $\text{valor}[S]$. Nuestra tarea es calcular para cada S

$$\text{suma}(S) = \sum_{A \subset S} \text{valor}[A],$$

es decir, la suma de los valores de los subconjuntos de S .

Por ejemplo, suponga que $n = 3$ y los valores son los siguientes:

- $\text{valor}[\emptyset] = 3$
- $\text{valor}[\{0\}] = 1$
- $\text{valor}[\{1\}] = 4$
- $\text{valor}[\{0,1\}] = 5$
- $\text{valor}[\{2\}] = 5$
- $\text{valor}[\{0,2\}] = 1$
- $\text{valor}[\{1,2\}] = 3$
- $\text{valor}[\{0,1,2\}] = 3$

En este caso, por ejemplo,

$$\begin{aligned} \text{suma}(\{0,2\}) &= \text{valor}[\emptyset] + \text{valor}[\{0\}] + \text{valor}[\{2\}] + \text{valor}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

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 programación dinámica, podemos resolver el problema en tiempo $O(2^n n)$. La idea es

centrarse en las sumas donde los elementos que pueden eliminarse de S están restringidos.

Denotemos a $\text{parcial}(S, k)$ como la suma de valores de subconjuntos de S con la restricción de que solo los elementos $0 \dots k$ pueden eliminarse de S . Por ejemplo,

$$\text{parcial}(\{0, 2\}, 1) = \text{valor}[\{2\}] + \text{valor}[\{0, 2\}],$$

porque solo podemos eliminar elementos $0 \dots 1$. Podemos calcular valores de suma usando valores de parcial , porque

$$\text{suma}(S) = \text{parcial}(S, n - 1).$$

Los casos base para la función son

$$\text{parcial}(S, -1) = \text{valor}[S],$$

porque en este caso no se pueden eliminar elementos de S . Luego, en el caso general, podemos usar la siguiente recurrencia:

$$\text{parcial}(S, k) = \begin{cases} \text{parcial}(S, k - 1) & k \notin S \\ \text{parcial}(S, k - 1) + \text{parcial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

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 especialmente ingeniosa de implementar el cálculo de sumas. Podemos declarar un arreglo

```
int suma[1<<N];
```

que contendrá la suma de cada subconjunto. El arreglo se inicializa de la siguiente manera:

```
for (int s = 0; s < (1<<n); s++) {
    suma[s] = valor[s];
}
```

Luego, podemos llenar el arreglo de la siguiente manera:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s & (1<<k)) suma[s] += suma[s ^ (1<<k)];
    }
}
```

Este código calcula los valores de $\text{parcial}(S, k)$ para $k = 0 \dots n - 1$ en el arreglo suma . Dado que $\text{parcial}(S, k)$ siempre se basa en $\text{parcial}(S, k - 1)$, podemos reutilizar el arreglo suma , lo que resulta en una implementación muy eficiente.

Part II

Graph algorithms

Chapter 11

Basics of graphs

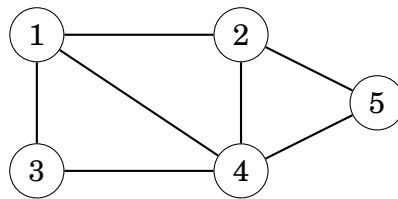
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

This part of the book discusses graph algorithms, especially focusing on topics that are important in competitive programming. In this chapter, we go through concepts related to graphs, and study different ways to represent graphs in algorithms.

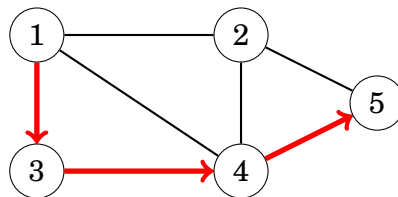
11.1 Graph terminology

A **graph** consists of **nodes** and **edges**. In this book, the variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges. The nodes are numbered using integers $1, 2, \dots, n$.

For example, the following graph consists of 5 nodes and 7 edges:



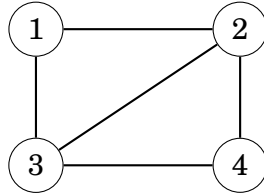
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ of length 3 from node 1 to node 5:



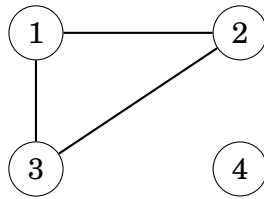
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

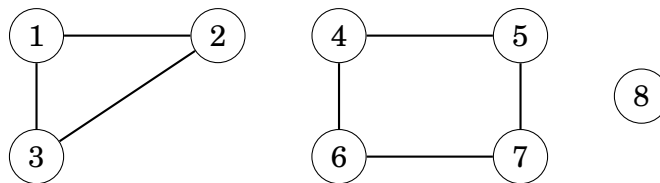
A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected:



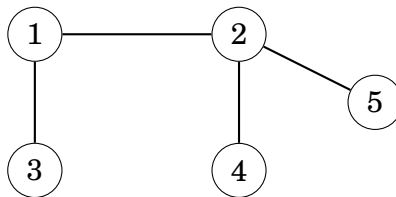
The following graph is not connected, because it is not possible to get from node 4 to any other node:



The connected parts of a graph are called its **components**. For example, the following graph contains three components: {1, 2, 3}, {4, 5, 6, 7} and {8}.

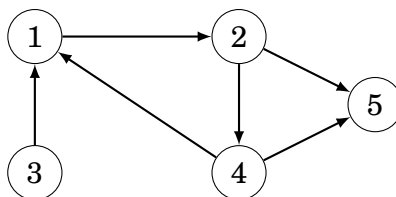


A **tree** is a connected graph that consists of n nodes and $n - 1$ edges. There is a unique path between any two nodes of a tree. For example, the following graph is a tree:



Edge directions

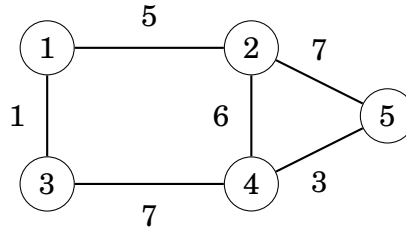
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



The above graph contains a path $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ from node 3 to node 5, but there is no path from node 5 to node 3.

Edge weights

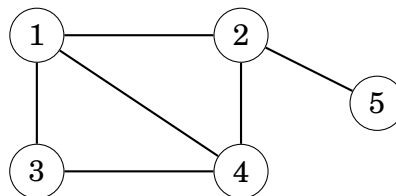
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



The length of a path in a weighted graph is the sum of the edge weights on the path. For example, in the above graph, the length of the path $1 \rightarrow 2 \rightarrow 5$ is 12, and the length of the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ is 11. The latter path is the **shortest** path from node 1 to node 5.

Neighbors and degrees

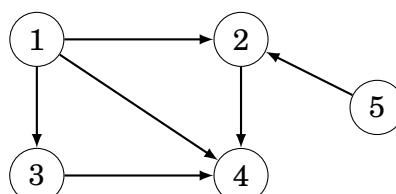
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



The sum of degrees in a graph is always $2m$, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even.

A graph is **regular** if the degree of every node is a constant d . A graph is **complete** if the degree of every node is $n - 1$, i.e., the graph contains all possible edges between the nodes.

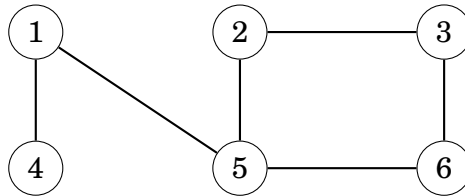
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2, and the outdegree of node 2 is 1.



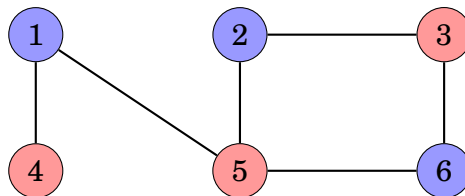
Colorings

In a **coloring** of a graph, each node is assigned a color so that no adjacent nodes have the same color.

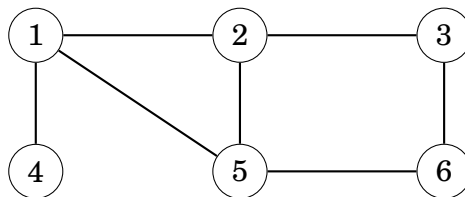
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



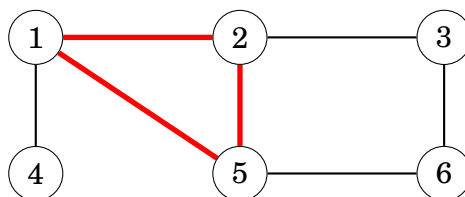
is bipartite, because it can be colored as follows:



However, the graph

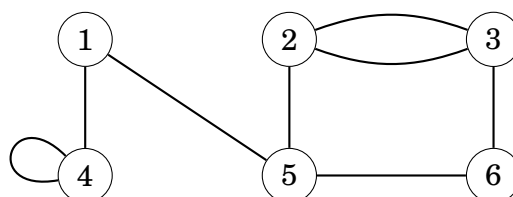


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



Simplicity

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is *not* simple:



11.2 Graph representation

There are several ways to represent graphs in algorithms. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. Next we will go through three common representations.

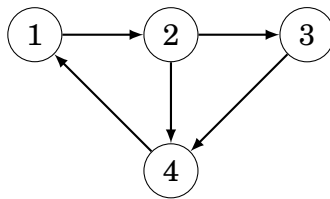
Adjacency list representation

In the adjacency list representation, each node x in the graph is assigned an **adjacency list** that consists of nodes to which there is an edge from x . Adjacency lists are the most popular way to represent graphs, and most algorithms can be efficiently implemented using them.

A convenient way to store the adjacency lists is to declare an array of vectors as follows:

```
vector<int> adj[N];
```

The constant N is chosen so that all adjacency lists can be stored. For example, the graph



can be stored as follows:

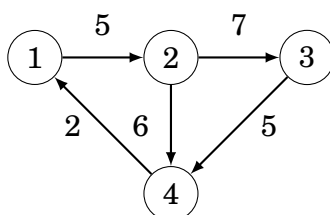
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

If the graph is undirected, it can be stored in a similar way, but each edge is added in both directions.

For a weighted graph, the structure can be extended as follows:

```
vector<pair<int,int>> adj[N];
```

In this case, the adjacency list of node a contains the pair (b, w) always when there is an edge from node a to node b with weight w . For example, the graph



can be stored as follows:

```
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});
```

The benefit of using adjacency lists is that we can efficiently find the nodes to which we can move from a given node through an edge. For example, the following loop goes through all nodes to which we can move from node s :

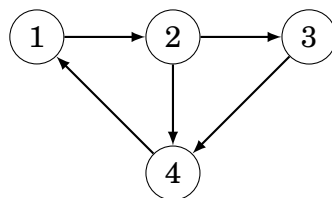
```
for (auto u : adj[s]) {  
    // process node u  
}
```

Adjacency matrix representation

An **adjacency matrix** is a two-dimensional array that indicates which edges the graph contains. We can efficiently check from an adjacency matrix if there is an edge between two nodes. The matrix can be stored as an array

```
int adj[N][N];
```

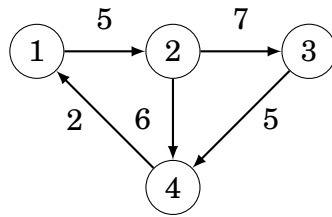
where each value $\text{adj}[a][b]$ indicates whether the graph contains an edge from node a to node b . If the edge is included in the graph, then $\text{adj}[a][b] = 1$, and otherwise $\text{adj}[a][b] = 0$. For example, the graph



can be represented as follows:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

If the graph is weighted, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph



corresponds to the following matrix:

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

The drawback of the adjacency matrix representation is that the matrix contains n^2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.

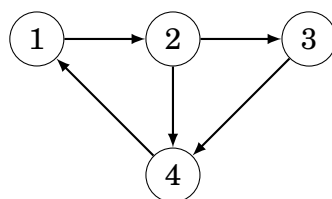
Edge list representation

An **edge list** contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node.

The edge list can be stored in a vector

```
vector<pair<int,int>> edges;
```

where each pair (a, b) denotes that there is an edge from node a to node b . Thus, the graph



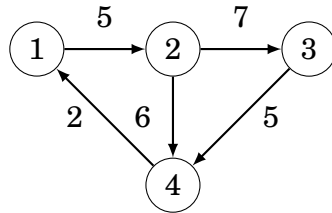
can be represented as follows:

```
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});
```

If the graph is weighted, the structure can be extended as follows:

```
vector<tuple<int,int,int>> edges;
```

Each element in this list is of the form (a,b,w) , which means that there is an edge from node a to node b with weight w . For example, the graph



can be represented as follows¹:

```
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});
```

¹In some older compilers, the function `make_tuple` must be used instead of the braces (for example, `make_tuple(1,2,5)` instead of `{1,2,5}`).

Chapter 12

Graph traversal

This chapter discusses two fundamental graph algorithms: depth-first search and breadth-first search. Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from the starting node. The difference in the algorithms is the order in which they visit the nodes.

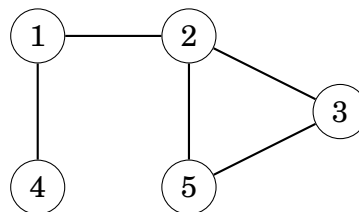
12.1 Depth-first search

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

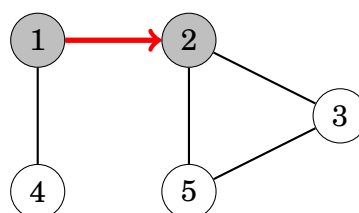
Example

Let us consider how depth-first search processes the following graph:

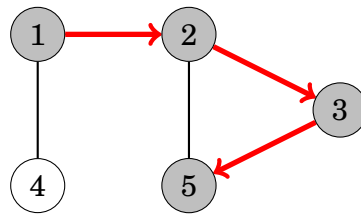


We may begin the search at any node of the graph; now we will begin the search at node 1.

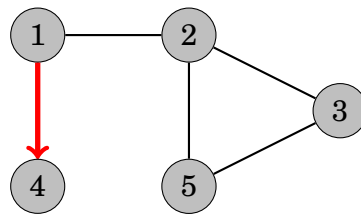
The search first proceeds to node 2:



After this, nodes 3 and 5 will be visited:



The neighbors of node 5 are 2 and 3, but the search has already visited both of them, so it is time to return to the previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:



After this, the search terminates because it has visited all nodes.

The time complexity of depth-first search is $O(n + m)$ where n is the number of nodes and m is the number of edges, because the algorithm processes each node and edge once.

Implementation

Depth-first search can be conveniently implemented using recursion. The following function `dfs` begins a depth-first search at a given node. The function assumes that the graph is stored as adjacency lists in an array

```
vector<int> adj[N];
```

and also maintains an array

```
bool visited[N];
```

that keeps track of the visited nodes. Initially, each array value is false, and when the search arrives at node s , the value of `visited[s]` becomes true. The function can be implemented as follows:

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

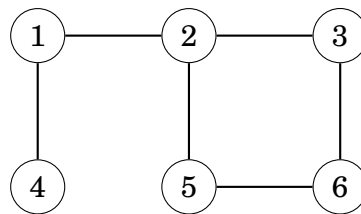

12.2 Breadth-first search

Breadth-first search (BFS) visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search.

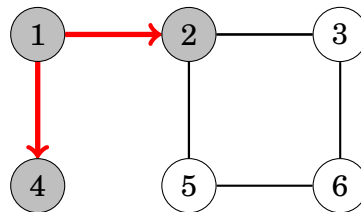
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

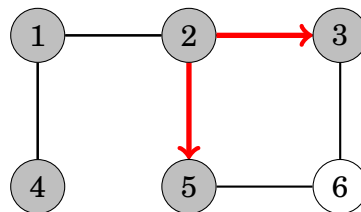
Let us consider how breadth-first search processes the following graph:



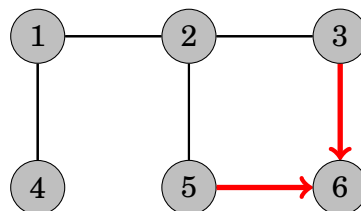
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we proceed to nodes 3 and 5:



Finally, we visit node 6:



Now we have calculated the distances from the starting node to all nodes of the graph. The distances are as follows:

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

Like in depth-first search, the time complexity of breadth-first search is $O(n + m)$, where n is the number of nodes and m is the number of edges.

Implementation

Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

The following code assumes that the graph is stored as adjacency lists and maintains the following data structures:

```
queue<int> q;
bool visited[N];
int distance[N];
```

The queue q contains nodes to be processed in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed. The array `visited` indicates which nodes the search has already visited, and the array `distance` will contain the distances from the starting node to all nodes of the graph.

The search can be implemented as follows, starting at node 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);
    }
}
```

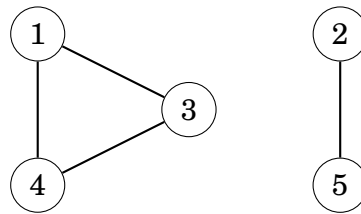
12.3 Applications

Using the graph traversal algorithms, we can check many properties of graphs. Usually, both depth-first search and breadth-first search may be used, but in practice, depth-first search is a better choice, because it is easier to implement. In the following applications we will assume that the graph is undirected.

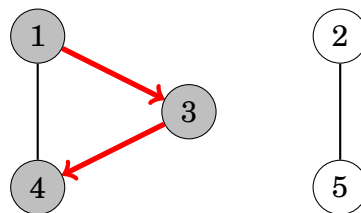
Connectivity check

A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

For example, in the graph



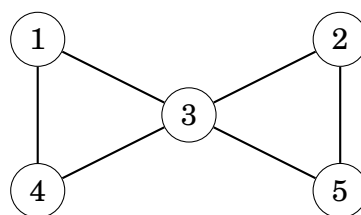
a depth-first search from node 1 visits the following nodes:



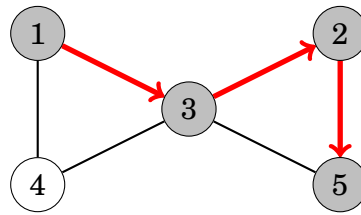
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

Finding cycles

A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, the graph



contains two cycles and we can find one of them as follows:



After moving from node 2 to node 5 we notice that the neighbor 3 of node 5 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

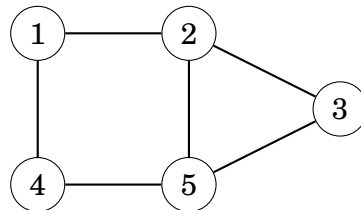
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c - 1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

Bipartiteness check

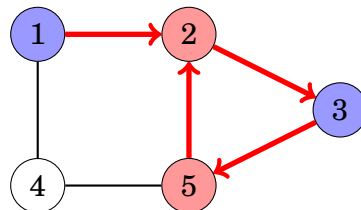
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found.

For example, the graph



is not bipartite, because a search from node 1 proceeds as follows:



We notice that the color of both nodes 2 and 5 is red, while they are adjacent nodes in the graph. Thus, the graph is not bipartite.

This algorithm always works, because when there are only two colors available, the color of the starting node in a component determines the colors of all other nodes in the component. It does not make any difference whether the starting node is red or blue.

Note that in the general case, it is difficult to find out if the nodes in a graph can be colored using k colors so that no adjacent nodes have the same color. Even when $k = 3$, no efficient algorithm is known but the problem is NP-hard.

Chapter 13

Shortest paths

Finding a shortest path between two nodes of a graph is an important problem that has many practical applications. For example, a natural problem related to a road network is to calculate the shortest possible length of a route between two cities, given the lengths of the roads.

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path. However, in this chapter we focus on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

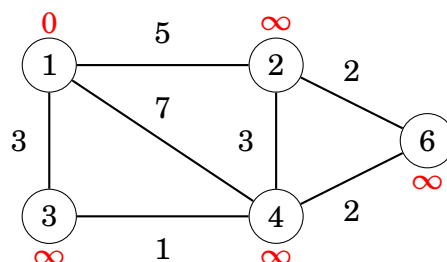
13.1 Bellman–Ford algorithm

The **Bellman–Ford algorithm**¹ finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes is infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Example

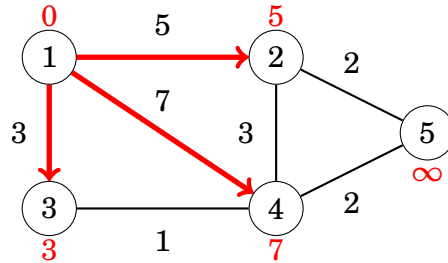
Let us consider how the Bellman–Ford algorithm works in the following graph:



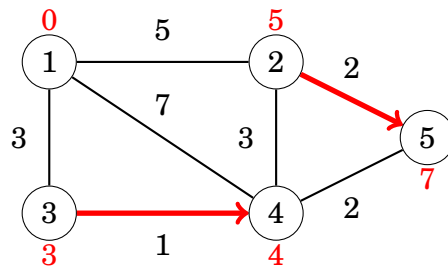
¹The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [5, 24].

Each node of the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

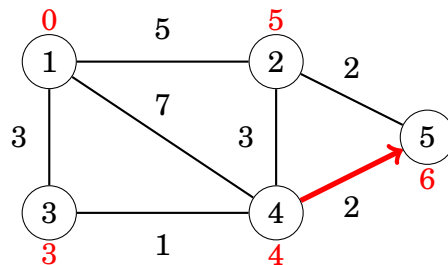
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



After this, edges $2 \rightarrow 5$ and $3 \rightarrow 4$ reduce distances:

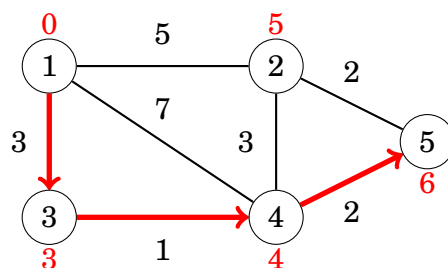


Finally, there is one more change:



After this, no edge can reduce any distance. This means that the distances are final, and we have successfully calculated the shortest distances from the starting node to all nodes of the graph.

For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



Implementation

The following implementation of the Bellman–Ford algorithm determines the shortest distances from a node x to all nodes of the graph. The code assumes that the graph is stored as an edge list edges that consists of tuples of the form (a, b, w) , meaning that there is an edge from node a to node b with weight w .

The algorithm consists of $n - 1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an array distance that will contain the distances from x to all nodes of the graph. The constant INF denotes an infinite distance.

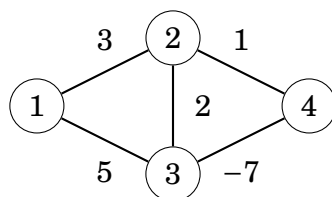
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n - 1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n - 1$ rounds, because each shortest path can contain at most $n - 1$ edges.

In practice, the final distances can usually be found faster than in $n - 1$ rounds. Thus, a possible way to make the algorithm more efficient is to stop the algorithm if no distance can be reduced during a round.

Negative cycles

The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ with length -4 .

If the graph contains a negative cycle, we can shorten infinitely many times any path that contains the cycle by repeating the cycle again and again. Thus, the concept of a shortest path is not meaningful in this situation.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

SPFA algorithm

The **SPFA algorithm** ("Shortest Path Faster Algorithm") [20] is a variant of the Bellman–Ford algorithm, that is often more efficient than the original algorithm. The SPFA algorithm does not go through all the edges on each round, but instead, it chooses the edges to be examined in a more intelligent way.

The algorithm maintains a queue of nodes that might be used for reducing the distances. First, the algorithm adds the starting node x to the queue. Then, the algorithm always processes the first node in the queue, and when an edge $a \rightarrow b$ reduces a distance, node b is added to the queue.

The efficiency of the SPFA algorithm depends on the structure of the graph: the algorithm is often efficient, but its worst case time complexity is still $O(nm)$ and it is possible to create inputs that make the algorithm as slow as the original Bellman–Ford algorithm.

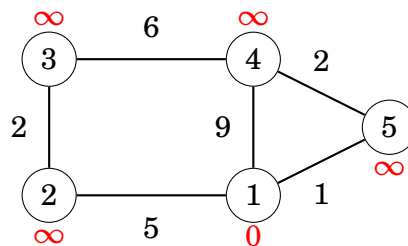
13.2 Dijkstra's algorithm

Dijkstra's algorithm² finds shortest paths from the starting node to all nodes of the graph, like the Bellman–Ford algorithm. The benefit of Dijkstra's algorithm is that it is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Like the Bellman–Ford algorithm, Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

Example

Let us consider how Dijkstra's algorithm works in the following graph when the starting node is node 1:

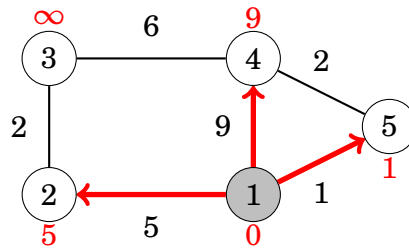


Like in the Bellman–Ford algorithm, initially the distance to the starting node is 0 and the distance to all other nodes is infinite.

At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. The first such node is node 1 with distance 0.

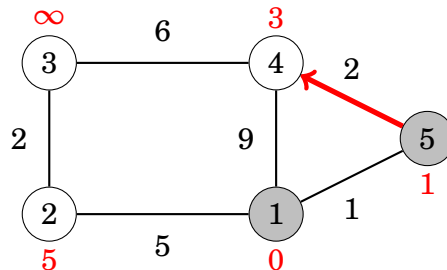
²E. W. Dijkstra published the algorithm in 1959 [14]; however, his original paper does not mention how to implement the algorithm efficiently.

When a node is selected, the algorithm goes through all edges that start at the node and reduces the distances using them:

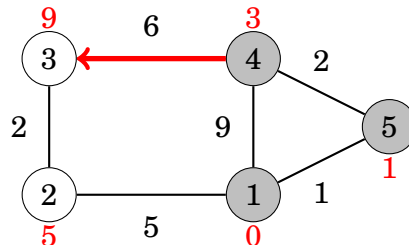


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

The next node to be processed is node 5 with distance 1. This reduces the distance to node 4 from 9 to 3:

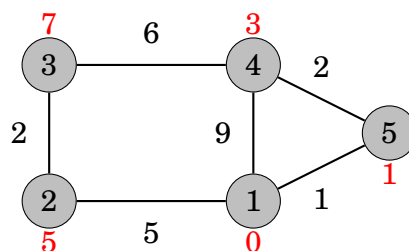


After this, the next node is node 4, which reduces the distance to node 3 to 9:



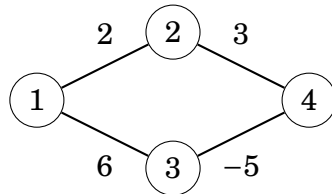
A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

After this, the algorithm processes the two remaining nodes, and the final distances are as follows:



Negative edges

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:



The shortest path from node 1 to node 4 is $1 \rightarrow 3 \rightarrow 4$ and its length is 1. However, Dijkstra's algorithm finds the path $1 \rightarrow 2 \rightarrow 4$ by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distances from a node x to other nodes of the graph. The graph is stored as adjacency lists so that $\text{adj}[a]$ contains a pair (b, w) always when there is an edge from node a to node b with weight w .

An efficient implementation of Dijkstra's algorithm requires that it is possible to efficiently find the minimum distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

In the following code, the priority queue q contains pairs of the form $(-d, x)$, meaning that the current distance to node x is d . The array distance contains the distance to each node, and the array processed indicates whether a node has been processed. Initially the distance is 0 to x and ∞ to all other nodes.

```
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});
        }
    }
}
```

Note that the priority queue contains *negative* distances to nodes. The reason for this is that the default version of the C++ priority queue finds maximum elements, while we want to find minimum elements. By using negative distances, we can directly use the default priority queue³. Also note that there may be several instances of the same node in the priority queue; however, only the instance with the minimum distance will be processed.

The time complexity of the above implementation is $O(n + m \log m)$, because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

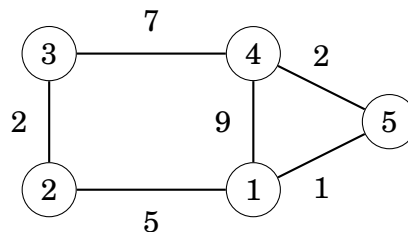
13.3 Floyd–Warshall algorithm

The **Floyd–Warshall algorithm**⁴ provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms of this chapter, it finds all shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Example

Let us consider how the Floyd–Warshall algorithm works in the following graph:



Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x . All other distances are infinite.

In this graph, the initial array is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

³Of course, we could also declare the priority queue as in Chapter 4.5 and use positive distances, but the implementation would be a bit longer.

⁴The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [23, 70].

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

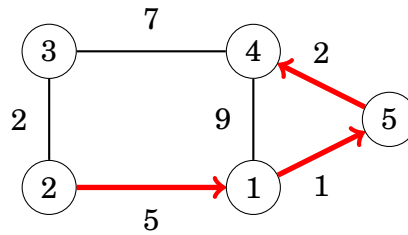
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	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

The algorithm continues like this, until all nodes have been appointed intermediate nodes. After the algorithm has finished, the array contains the minimum distances between any two nodes:

	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

For example, the array tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:



Implementation

The advantage of the Floyd–Warshall algorithm is that it is easy to implement. The following code constructs a distance matrix where $\text{distance}[a][b]$ is the shortest distance between nodes a and b . First, the algorithm initializes distance using the adjacency matrix adj of the graph:

```

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;
    }
}

```

After this, the shortest distances can be found as follows:

```

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]);
        }
    }
}

```

The time complexity of the algorithm is $O(n^3)$, because it contains three nested loops that go through the nodes of the graph.

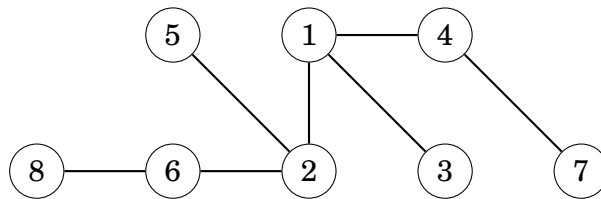
Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Chapter 14

Tree algorithms

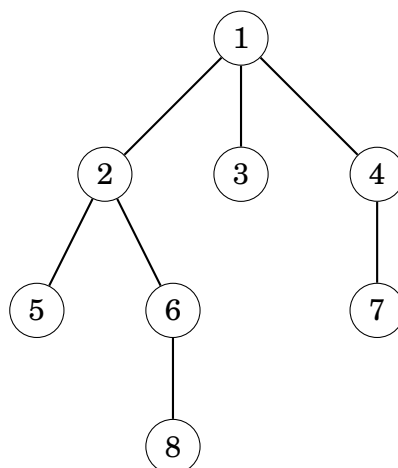
A **tree** is a connected, acyclic graph that consists of n nodes and $n - 1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 8 nodes and 7 edges:



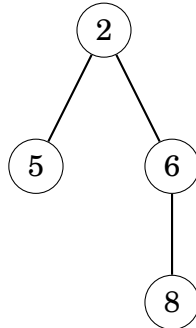
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.

In a **rooted** tree, one of the nodes is appointed the **root** of the tree, and all other nodes are placed underneath the root. For example, in the following tree, node 1 is the root node.



In a rooted tree, the **children** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except for the root that does not have a parent. For example, in the above tree, the children of node 2 are nodes 5 and 6, and its parent is node 1.

The structure of a rooted tree is *recursive*: each node of the tree acts as the root of a **subtree** that contains the node itself and all nodes that are in the subtrees of its children. For example, in the above tree, the subtree of node 2 consists of nodes 2, 5, 6 and 8:



14.1 Tree traversal

General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
void dfs(int s, int e) {  
    // process node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

The function is given two parameters: the current node s and the previous node e . The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x :

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic programming

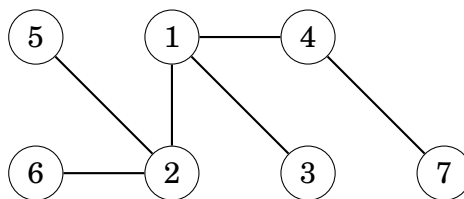
Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node of a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.

As an example, let us calculate for each node s a value $\text{count}[s]$: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children, so we can calculate the number of nodes recursively using the following code:

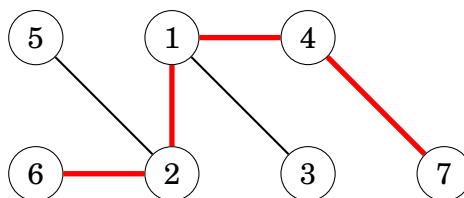
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Diameter

The **diameter** of a tree is the maximum length of a path between two nodes. For example, consider the following tree:



The diameter of this tree is 4, which corresponds to the following path:



Note that there may be several maximum-length paths. In the above path, we could replace node 6 with node 5 to obtain another path with length 4.

Next we will discuss two $O(n)$ time algorithms for calculating the diameter of a tree. The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches.

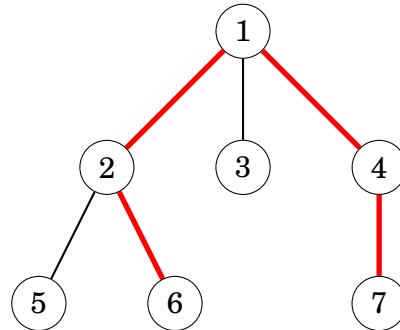
Algorithm 1

A general way to approach many tree problems is to first root the tree arbitrarily. After this, we can try to solve the problem separately for each subtree. Our first algorithm for calculating the diameter is based on this idea.

An important observation is that every path in a rooted tree has a *highest point*: the highest node that belongs to the path. Thus, we can calculate for each

node the length of the longest path whose highest point is the node. One of those paths corresponds to the diameter of the tree.

For example, in the following tree, node 1 is the highest point on the path that corresponds to the diameter:



We calculate for each node x two values:

- $\text{toLeaf}(x)$: the maximum length of a path from x to any leaf
- $\text{maxLength}(x)$: the maximum length of a path whose highest point is x

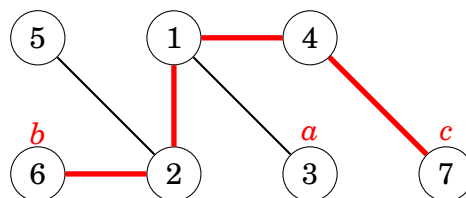
For example, in the above tree, $\text{toLeaf}(1) = 2$, because there is a path $1 \rightarrow 2 \rightarrow 6$, and $\text{maxLength}(1) = 4$, because there is a path $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$. In this case, $\text{maxLength}(1)$ equals the diameter.

Dynamic programming can be used to calculate the above values for all nodes in $O(n)$ time. First, to calculate $\text{toLeaf}(x)$, we go through the children of x , choose a child c with maximum $\text{toLeaf}(c)$ and add one to this value. Then, to calculate $\text{maxLength}(x)$, we choose two distinct children a and b such that the sum $\text{toLeaf}(a) + \text{toLeaf}(b)$ is maximum and add two to this sum.

Algorithm 2

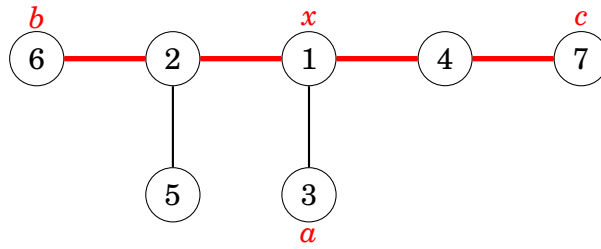
Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a . Then, we find the farthest node c from b . The diameter of the tree is the distance between b and c .

In the following graph, a , b and c could be:



This is an elegant method, but why does it work?

It helps to draw the tree differently so that the path that corresponds to the diameter is horizontal, and all other nodes hang from it:

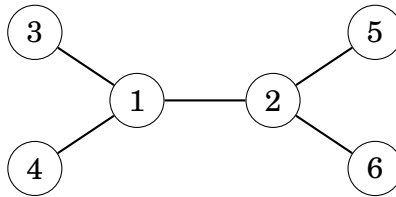


Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b , node c or some other node that is at least as far from node x . Thus, this node is always a valid choice for an endpoint of a path that corresponds to the diameter.

14.3 All longest paths

Our next problem is to calculate for every node in the tree the maximum length of a path that begins at the node. This can be seen as a generalization of the tree diameter problem, because the largest of those lengths equals the diameter of the tree. Also this problem can be solved in $O(n)$ time.

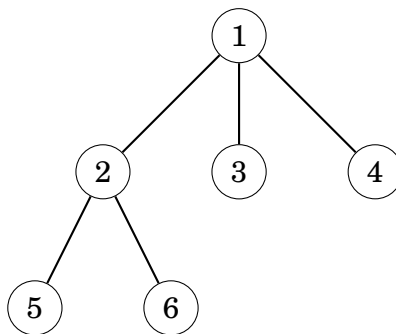
As an example, consider the following tree:



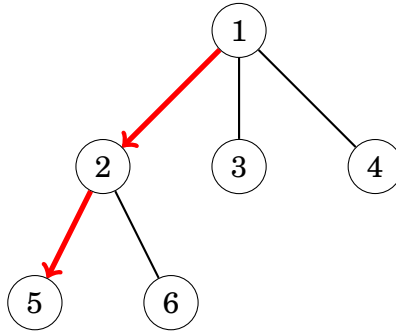
Let $\text{maxLength}(x)$ denote the maximum length of a path that begins at node x . For example, in the above tree, $\text{maxLength}(4) = 3$, because there is a path $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Here is a complete table of the values:

node x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Also in this problem, a good starting point for solving the problem is to root the tree arbitrarily:

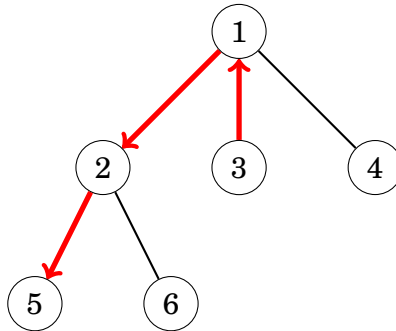


The first part of the problem is to calculate for every node x the maximum length of a path that goes through a child of x . For example, the longest path from node 1 goes through its child 2:

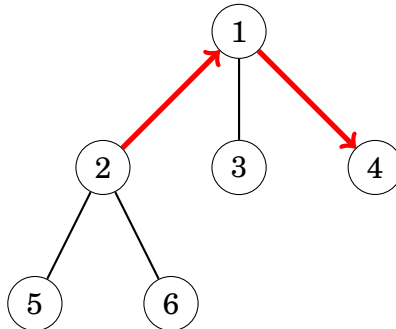


This part is easy to solve in $O(n)$ time, because we can use dynamic programming as we have done previously.

Then, the second part of the problem is to calculate for every node x the maximum length of a path through its parent p . For example, the longest path from node 3 goes through its parent 1:



At first glance, it seems that we should choose the longest path from p . However, this *does not* always work, because the longest path from p may go through x . Here is an example of this situation:



Still, we can solve the second part in $O(n)$ time by storing *two* maximum lengths for each node x :

- $\text{maxLength}_1(x)$: the maximum length of a path from x
- $\text{maxLength}_2(x)$ the maximum length of a path from x in another direction than the first path

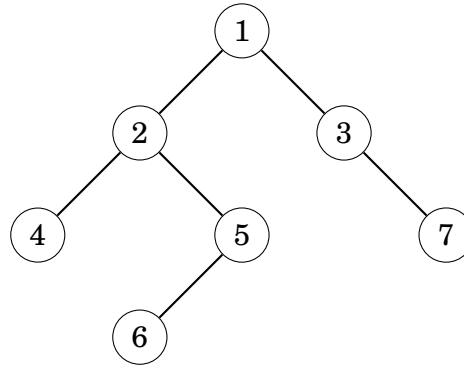
For example, in the above graph, $\text{maxLength}_1(1) = 2$ using the path $1 \rightarrow 2 \rightarrow 5$, and $\text{maxLength}_2(1) = 1$ using the path $1 \rightarrow 3$.

Finally, if the path that corresponds to $\text{maxLength}_1(p)$ goes through x , we conclude that the maximum length is $\text{maxLength}_2(p) + 1$, and otherwise the maximum length is $\text{maxLength}_1(p) + 1$.

14.4 Binary trees

A **binary tree** is a rooted tree where each node has a left and right subtree. It is possible that a subtree of a node is empty. Thus, every node in a binary tree has zero, one or two children.

For example, the following tree is a binary tree:



The nodes of a binary tree have three natural orderings that correspond to different ways to recursively traverse the tree:

- **pre-order**: first process the root, then traverse the left subtree, then traverse the right subtree
- **in-order**: first traverse the left subtree, then process the root, then traverse the right subtree
- **post-order**: first traverse the left subtree, then traverse the right subtree, then process the root

For the above tree, the nodes in pre-order are [1,2,4,5,6,3,7], in in-order [4,2,6,5,1,3,7] and in post-order [4,6,5,2,7,3,1].

If we know the pre-order and in-order of a tree, we can reconstruct the exact structure of the tree. For example, the above tree is the only possible tree with pre-order [1,2,4,5,6,3,7] and in-order [4,2,6,5,1,3,7]. In a similar way, the post-order and in-order also determine the structure of a tree.

However, the situation is different if we only know the pre-order and post-order of a tree. In this case, there may be more than one tree that match the orderings. For example, in both of the trees



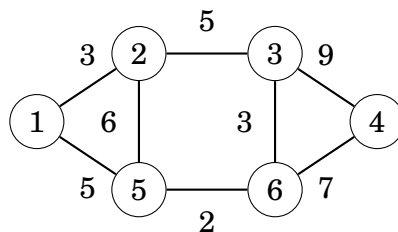
the pre-order is [1,2] and the post-order is [2,1], but the structures of the trees are different.

Chapter 15

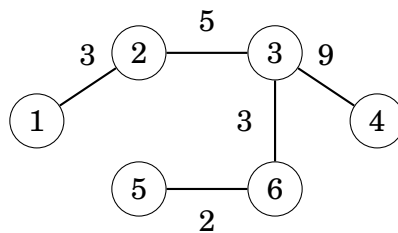
Spanning trees

A **spanning tree** of a graph consists of all nodes of the graph and some of the edges of the graph so that there is a path between any two nodes. Like trees in general, spanning trees are connected and acyclic. Usually there are several ways to construct a spanning tree.

For example, consider the following graph:

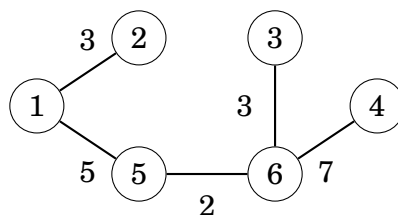


One spanning tree for the graph is as follows:

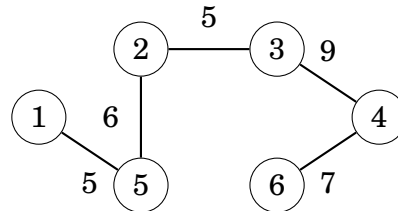


The weight of a spanning tree is the sum of its edge weights. For example, the weight of the above spanning tree is $3 + 5 + 9 + 3 + 2 = 22$.

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the example graph is 20, and such a tree can be constructed as follows:



In a similar way, a **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the example graph is 32:



Note that a graph may have several minimum and maximum spanning trees, so the trees are not unique.

It turns out that several greedy methods can be used to construct minimum and maximum spanning trees. In this chapter, we discuss two algorithms that process the edges of the graph ordered by their weights. We focus on finding minimum spanning trees, but the same algorithms can find maximum spanning trees by processing the edges in reverse order.

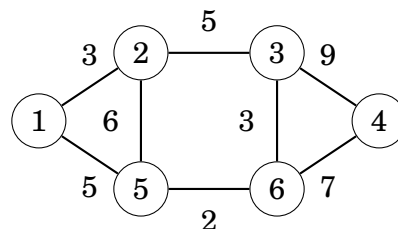
15.1 Kruskal's algorithm

In **Kruskal's algorithm**¹, the initial spanning tree only contains the nodes of the graph and does not contain any edges. Then the algorithm goes through the edges ordered by their weights, and always adds an edge to the tree if it does not create a cycle.

The algorithm maintains the components of the tree. Initially, each node of the graph belongs to a separate component. Always when an edge is added to the tree, two components are joined. Finally, all nodes belong to the same component, and a minimum spanning tree has been found.

Example

Let us consider how Kruskal's algorithm processes the following graph:



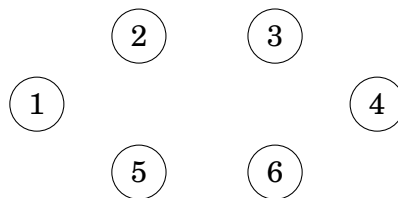
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

¹The algorithm was published in 1956 by J. B. Kruskal [48].

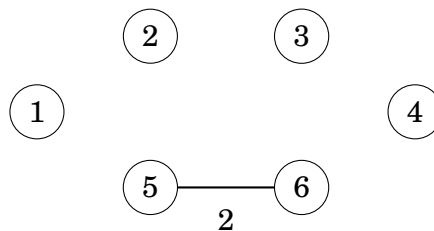
edge	weight
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components.

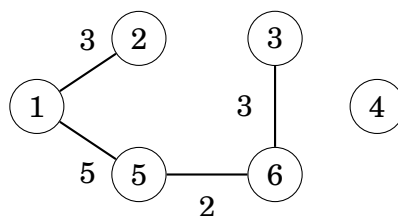
Initially, each node is in its own component:



The first edge to be added to the tree is the edge 5-6 that creates a component {5,6} by joining the components {5} and {6}:



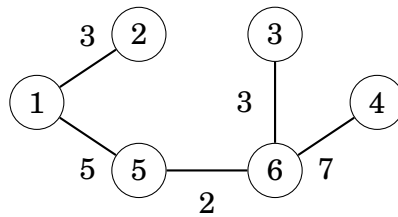
After this, the edges 1-2, 3-6 and 1-5 are added in a similar way:



After those steps, most components have been joined and there are two components in the tree: {1,2,3,5,6} and {4}.

The next edge in the list is the edge 2-3, but it will not be included in the tree, because nodes 2 and 3 are already in the same component. For the same reason, the edge 2-5 will not be included in the tree.

Finally, the edge 4–6 will be included in the tree:

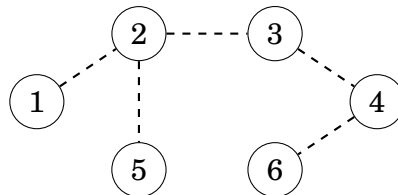


After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight $2 + 3 + 3 + 5 + 7 = 20$.

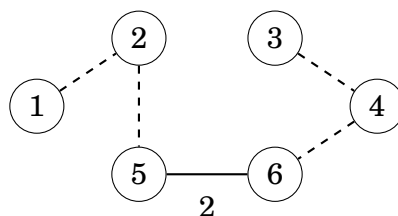
Why does this work?

It is a good question why Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let us see what happens if the minimum weight edge of the graph is *not* included in the spanning tree. For example, suppose that a spanning tree for the previous graph would not contain the minimum weight edge 5–6. We do not know the exact structure of such a spanning tree, but in any case it has to contain some edges. Assume that the tree would be as follows:



However, it is not possible that the above tree would be a minimum spanning tree for the graph. The reason for this is that we can remove an edge from the tree and replace it with the minimum weight edge 5–6. This produces a spanning tree whose weight is *smaller*:



For this reason, it is always optimal to include the minimum weight edge in the tree to produce a minimum spanning tree. Using a similar argument, we can show that it is also optimal to add the next edge in weight order to the tree, and so on. Hence, Kruskal's algorithm works correctly and always produces a minimum spanning tree.

Implementation

When implementing Kruskal's algorithm, it is convenient to use the edge list representation of the graph. The first phase of the algorithm sorts the edges in the list in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

The loop goes through the edges in the list and always processes an edge $a-b$ where a and b are two nodes. Two functions are needed: the function `same` determines if a and b are in the same component, and the function `unite` joins the components that contain a and b .

The problem is how to efficiently implement the functions `same` and `unite`. One possibility is to implement the function `same` as a graph traversal and check if we can get from node a to node b . However, the time complexity of such a function would be $O(n + m)$ and the resulting algorithm would be slow, because the function `same` will be called for each edge in the graph.

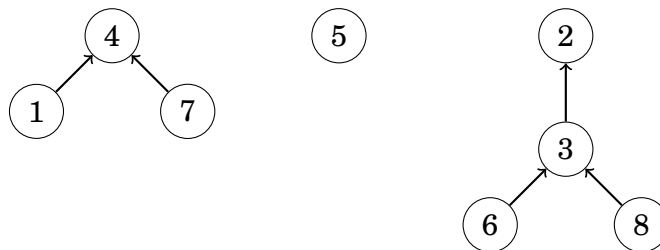
We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

15.2 Union-find structure

A **union-find structure** maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two $O(\log n)$ time operations are supported: the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element².

Structure

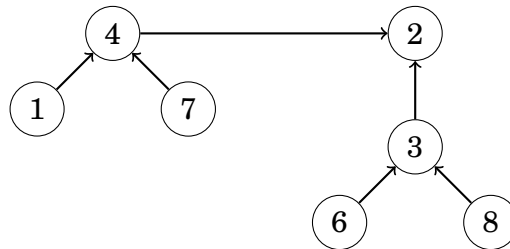
In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1, 4, 7\}$, $\{5\}$ and $\{2, 3, 6, 8\}$:



²The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [38]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [64] that is discussed in many algorithm textbooks nowadays.

In this case the representatives of the sets are 4, 5 and 2. We can find the representative of any element by following the chain that begins at the element. For example, the element 2 is the representative for the element 6, because we follow the chain $6 \rightarrow 3 \rightarrow 2$. Two elements belong to the same set exactly when their representatives are the same.

Two sets can be joined by connecting the representative of one set to the representative of the other set. For example, the sets $\{1, 4, 7\}$ and $\{2, 3, 6, 8\}$ can be joined as follows:



The resulting set contains the elements $\{1, 2, 3, 4, 6, 7, 8\}$. From this on, the element 2 is the representative for the entire set and the old representative 4 points to the element 2.

The efficiency of the union-find structure depends on how the sets are joined. It turns out that we can follow a simple strategy: always connect the representative of the *smaller* set to the representative of the *larger* set (or if the sets are of equal size, we can make an arbitrary choice). Using this strategy, the length of any chain will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding chain.

Implementation

The union-find structure can be implemented using arrays. In the following implementation, the array `link` contains for each element the next element in the chain or the element itself if it is a representative, and the array `size` indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

The function `find` returns the representative for an element x . The representative can be found by following the chain that begins at x .

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

The function `same` checks whether elements a and b belong to the same set. This can easily be done by using the function `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

The function `unite` joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
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;
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain is $O(\log n)$. In this case, the functions `same` and `unite` also work in $O(\log n)$ time. The function `unite` makes sure that the length of each chain is $O(\log n)$ by connecting the smaller set to the larger set.

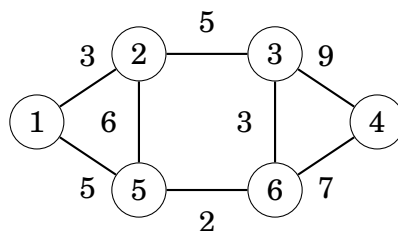
15.3 Prim's algorithm

Prim's algorithm³ is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning tree has been found.

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge whose distance from the starting node is minimum, but Prim's algorithm simply selects the minimum weight edge that adds a new node to the tree.

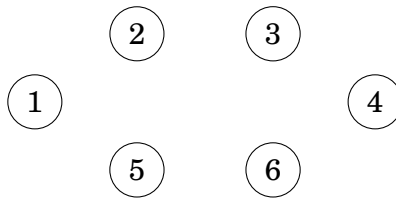
Example

Let us consider how Prim's algorithm works in the following graph:

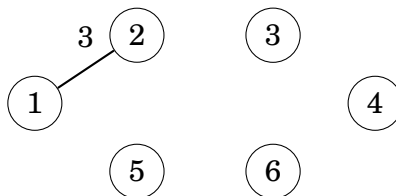


³The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.

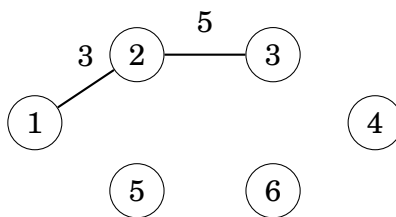
Initially, there are no edges between the nodes:



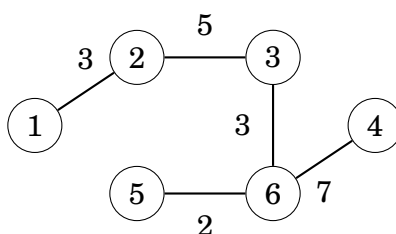
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let us add node 3 first:



The process continues until all nodes have been included in the tree:



Implementation

Like Dijkstra's algorithm, Prim's algorithm can be efficiently implemented using a priority queue. The priority queue should contain all nodes that can be connected to the current component using a single edge, in increasing order of the weights of the corresponding edges.

The time complexity of Prim's algorithm is $O(n + m \log m)$ that equals the time complexity of Dijkstra's algorithm. In practice, Prim's and Kruskal's algorithms are both efficient, and the choice of the algorithm is a matter of taste. Still, most competitive programmers use Kruskal's algorithm.

Chapter 16

Directed graphs

In this chapter, we focus on two classes of directed graphs:

- **Acyclic graphs:** There are no cycles in the graph, so there is no path from any node to itself¹.
- **Successor graphs:** The outdegree of each node is 1, so each node has a unique successor.

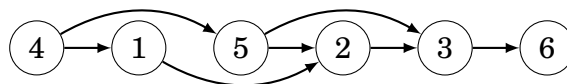
It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

16.1 Topological sorting

A **topological sort** is an ordering of the nodes of a directed graph such that if there is a path from node a to node b , then node a appears before node b in the ordering. For example, for the graph



one topological sort is [4, 1, 5, 2, 3, 6]:



An acyclic graph always has a topological sort. However, if the graph contains a cycle, it is not possible to form a topological sort, because no node of the cycle can appear before the other nodes of the cycle in the ordering. It turns out that depth-first search can be used to both check if a directed graph contains a cycle and, if it does not contain a cycle, to construct a topological sort.

¹Directed acyclic graphs are sometimes called DAGs.

Algorithm

The idea is to go through the nodes of the graph and always begin a depth-first search at the current node if it has not been processed yet. During the searches, the nodes have three possible states:

- state 0: the node has not been processed (white)
- state 1: the node is under processing (light gray)
- state 2: the node has been processed (dark gray)

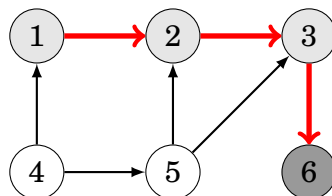
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

If the graph contains a cycle, we will find this out during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort.

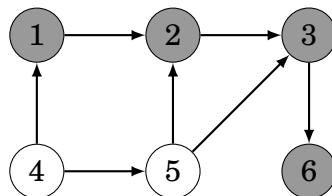
If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

Example 1

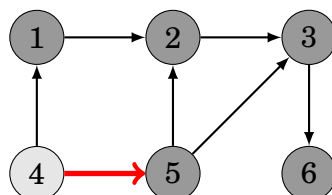
In the example graph, the search first proceeds from node 1 to node 6:



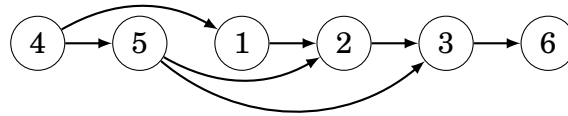
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:



At this point, the list is [6,3,2,1]. The next search begins at node 4:



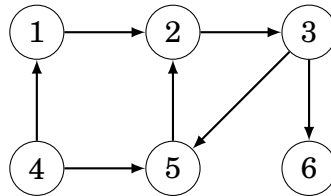
Thus, the final list is [6, 3, 2, 1, 5, 4]. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list [4, 5, 1, 2, 3, 6]:



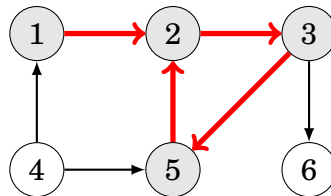
Note that a topological sort is not unique, and there can be several topological sorts for a graph.

Example 2

Let us now consider a graph for which we cannot construct a topological sort, because the graph contains a cycle:



The search proceeds as follows:



The search reaches node 2 whose state is 1, which means that the graph contains a cycle. In this example, there is a cycle $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

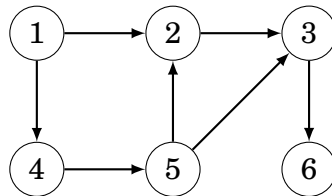
16.2 Dynamic programming

If a directed graph is acyclic, dynamic programming can be applied to it. For example, we can efficiently solve the following problems concerning paths from a starting node to an ending node:

- how many different paths are there?
- what is the shortest/longest path?
- what is the minimum/maximum number of edges in a path?
- which nodes certainly appear in any path?

Counting the number of paths

As an example, let us calculate the number of paths from node 1 to node 6 in the following graph:



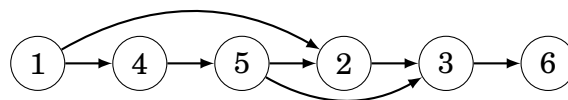
There are a total of three such paths:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

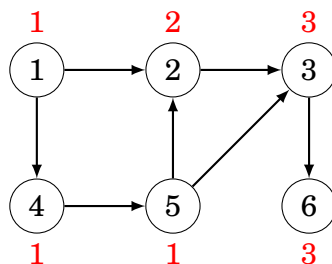
Let $\text{paths}(x)$ denote the number of paths from node 1 to node x . As a base case, $\text{paths}(1) = 1$. Then, to calculate other values of $\text{paths}(x)$, we may use the recursion

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \cdots + \text{paths}(a_k)$$

where a_1, a_2, \dots, a_k are the nodes from which there is an edge to x . Since the graph is acyclic, the values of $\text{paths}(x)$ can be calculated in the order of a topological sort. A topological sort for the above graph is as follows:



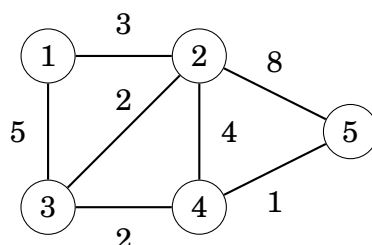
Hence, the numbers of paths are as follows:



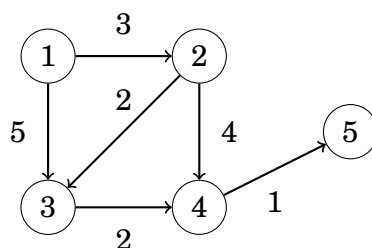
For example, to calculate the value of $\text{paths}(3)$, we can use the formula $\text{paths}(2) + \text{paths}(5)$, because there are edges from nodes 2 and 5 to node 3. Since $\text{paths}(2) = 2$ and $\text{paths}(5) = 1$, we conclude that $\text{paths}(3) = 3$.

Extending Dijkstra's algorithm

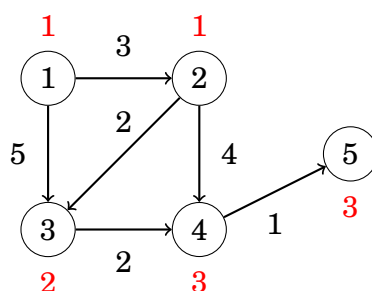
A by-product of Dijkstra's algorithm is a directed, acyclic graph that indicates for each node of the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied to that graph. For example, in the graph



the shortest paths from node 1 may use the following edges:



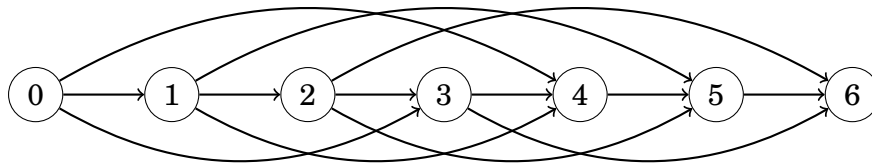
Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:



Representing problems as graphs

Actually, any dynamic programming problem can be represented as a directed, acyclic graph. In such a graph, each node corresponds to a dynamic programming state and the edges indicate how the states depend on each other.

As an example, consider the problem of forming a sum of money n using coins $\{c_1, c_2, \dots, c_k\}$. In this problem, we can construct a graph where each node corresponds to a sum of money, and the edges show how the coins can be chosen. For example, for coins $\{1, 3, 4\}$ and $n = 6$, the graph is as follows:



Using this representation, the shortest path from node 0 to node n corresponds to a solution with the minimum number of coins, and the total number of paths from node 0 to node n equals the total number of solutions.

16.3 Successor paths

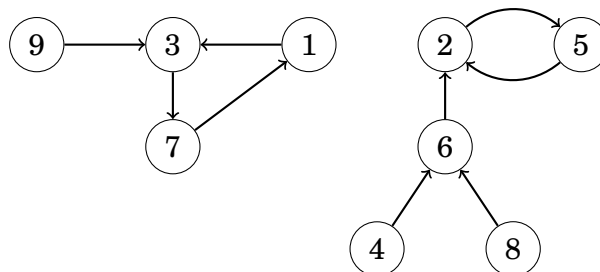
For the rest of the chapter, we will focus on **successor graphs**. In those graphs, the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called **functional graphs**. The reason for this is that any successor graph corresponds to a function that defines the edges of the graph. The parameter for the function is a node of the graph, and the function gives the successor of that node.

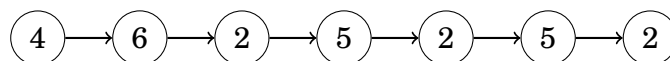
For example, the function

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

defines the following graph:



Since each node of a successor graph has a unique successor, we can also define a function $\text{succ}(x, k)$ that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph $\text{succ}(4, 6) = 2$, because we will reach node 2 by walking 6 steps from node 4:



A straightforward way to calculate a value of $\text{succ}(x, k)$ is to start at node x and walk k steps forward, which takes $O(k)$ time. However, using preprocessing, any value of $\text{succ}(x, k)$ can be calculated in only $O(\log k)$ time.

The idea is to precalculate all values of $\text{succ}(x, k)$ where k is a power of two and at most u , where u is the maximum number of steps we will ever walk. This can be efficiently done, because we can use the following recursion:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Precalculating the values takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above graph, the first values are as follows:

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
...									

After this, any value of $\text{succ}(x, k)$ can be calculated by presenting the number of steps k as a sum of powers of two. For example, if we want to calculate the value of $\text{succ}(x, 11)$, we first form the representation $11 = 8 + 2 + 1$. Using that,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

For example, in the previous graph

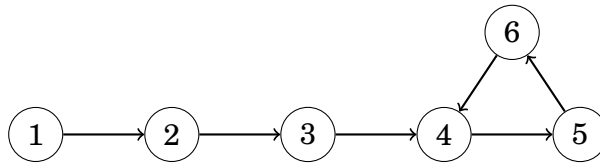
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Such a representation always consists of $O(\log k)$ parts, so calculating a value of $\text{succ}(x, k)$ takes $O(\log k)$ time.

16.4 Cycle detection

Consider a successor graph that only contains a path that ends in a cycle. We may ask the following questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain?

For example, in the graph



we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5 and 6).

A simple way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory.

However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if n is large. Next we will discuss Floyd's algorithm that achieves these properties.

Floyd's algorithm

Floyd's algorithm² walks forward in the graph using two pointers a and b . Both pointers begin at a node x that is the starting node of the graph. Then, on each turn, the pointer a walks one step forward and the pointer b walks two steps forward. The process continues until the pointers meet each other:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

At this point, the pointer a has walked k steps and the pointer b has walked $2k$ steps, so the length of the cycle divides k . Thus, the first node that belongs to the cycle can be found by moving the pointer a to node x and advancing the pointers step by step until they meet again.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

After this, the length of the cycle can be calculated as follows:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

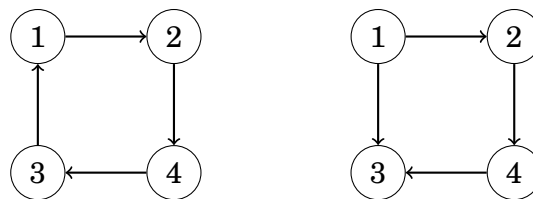
²The idea of the algorithm is mentioned in [46] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.

Chapter 17

Strong connectivity

In a directed graph, the edges can be traversed in one direction only, so even if the graph is connected, this does not guarantee that there would be a path from a node to another node. For this reason, it is meaningful to define a new concept that requires more than connectivity.

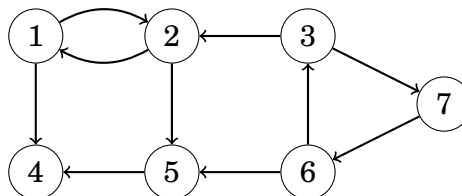
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.



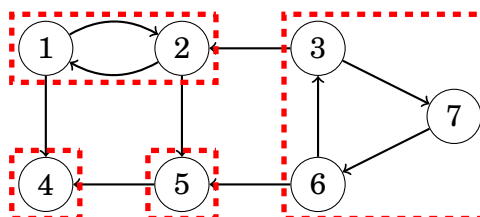
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The **strongly connected components** of a graph divide the graph into strongly connected parts that are as large as possible. The strongly connected components form an acyclic **component graph** that represents the deep structure of the original graph.

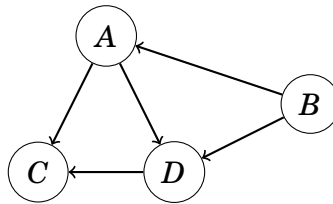
For example, for the graph



the strongly connected components are as follows:



The corresponding component graph is as follows:



The components are $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ and $D = \{5\}$.

A component graph is an acyclic, directed graph, so it is easier to process than the original graph. Since the graph does not contain cycles, we can always construct a topological sort and use dynamic programming techniques like those presented in Chapter 16.

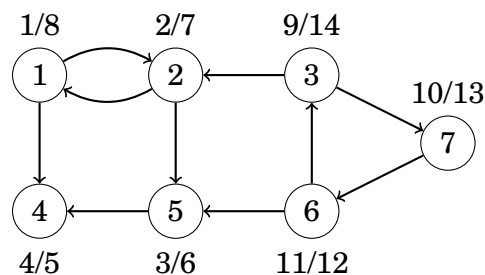
17.1 Kosaraju's algorithm

Kosaraju's algorithm¹ is an efficient method for finding the strongly connected components of a directed graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

Search 1

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which a depth-first search processes them. The algorithm goes through the nodes, and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed.

In the example graph, the nodes are processed in the following order:



The notation x/y means that processing the node started at time x and finished at time y . Thus, the corresponding list is as follows:

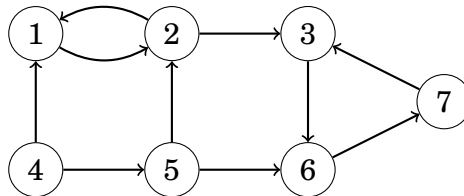
¹According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [57].

node	processing time
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Search 2

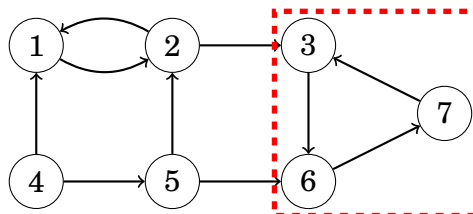
The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes.

After reversing the edges, the example graph is as follows:



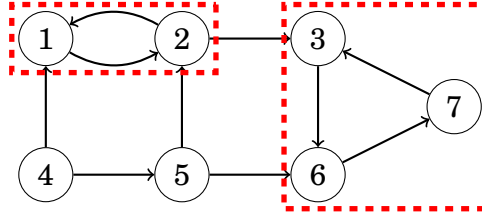
After this, the algorithm goes through the list of nodes created by the first search, in *reverse* order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

In the example graph, the first component begins at node 3:

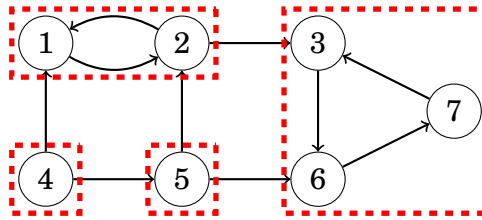


Note that since all edges are reversed, the component does not "leak" to other parts in the graph.

The next nodes in the list are nodes 7 and 6, but they already belong to a component, so the next new component begins at node 1:



Finally, the algorithm processes nodes 5 and 4 that create the remaining strongly connected components:



The time complexity of the algorithm is $O(n + m)$, because the algorithm performs two depth-first searches.

17.2 2SAT problem

Strong connectivity is also linked with the **2SAT problem**². In this problem, we are given a logical formula

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

where each a_i and b_i is either a logical variable (x_1, x_2, \dots, x_n) or a negation of a logical variable ($\neg x_1, \neg x_2, \dots, \neg x_n$). The symbols " \wedge " and " \vee " denote logical operators "and" and "or". Our task is to assign each variable a value so that the formula is true, or state that this is not possible.

For example, the formula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

is true when the variables are assigned as follows:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

²The algorithm presented here was introduced in [4]. There is also another well-known linear-time algorithm [19] that is based on backtracking.

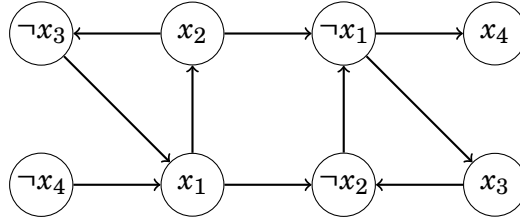
However, the formula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

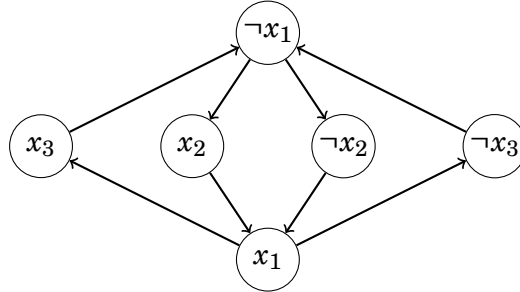
is always false, regardless of how we assign the values. The reason for this is that we cannot choose a value for x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should be true which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should be true which is also impossible.

The 2SAT problem can be represented as a graph whose nodes correspond to variables x_i and negations $\neg x_i$, and edges determine the connections between the variables. Each pair $(a_i \vee b_i)$ generates two edges: $\neg a_i \rightarrow b_i$ and $\neg b_i \rightarrow a_i$. This means that if a_i does not hold, b_i must hold, and vice versa.

The graph for the formula L_1 is:



And the graph for the formula L_2 is:



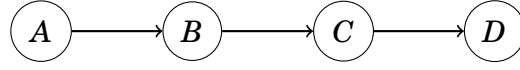
The structure of the graph tells us whether it is possible to assign the values of the variables so that the formula is true. It turns out that this can be done exactly when there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component. If there are such nodes, the graph contains a path from x_i to $\neg x_i$ and also a path from $\neg x_i$ to x_i , so both x_i and $\neg x_i$ should be true which is not possible.

In the graph of the formula L_1 there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component, so a solution exists. In the graph of the formula L_2 all nodes belong to the same strongly connected component, so a solution does not exist.

If a solution exists, the values for the variables can be found by going through the nodes of the component graph in a reverse topological sort order. At each step, we process a component that does not contain edges that lead to an unprocessed component. If the variables in the component have not been assigned values, their values will be determined according to the values in the component, and if

they already have values, they remain unchanged. The process continues until each variable has been assigned a value.

The component graph for the formula L_1 is as follows:



The components are $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ and $D = \{x_4\}$. When constructing the solution, we first process the component D where x_4 becomes true. After this, we process the component C where x_1 and x_2 become false and x_3 becomes true. All variables have been assigned values, so the remaining components A and B do not change the variables.

Note that this method works, because the graph has a special structure: if there are paths from node x_i to node x_j and from node x_j to node $\neg x_j$, then node x_i never becomes true. The reason for this is that there is also a path from node $\neg x_j$ to node $\neg x_i$, and both x_i and x_j become false.

A more difficult problem is the **3SAT problem**, where each part of the formula is of the form $(a_i \vee b_i \vee c_i)$. This problem is NP-hard, so no efficient algorithm for solving the problem is known.

Chapter 18

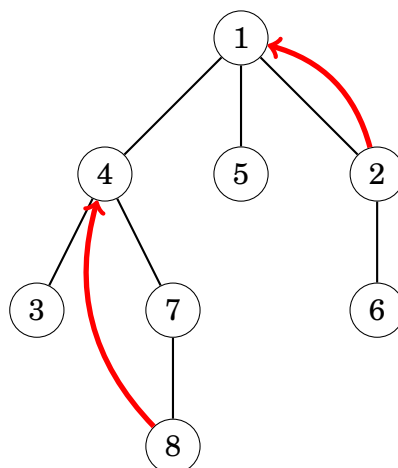
Tree queries

This chapter discusses techniques for processing queries on subtrees and paths of a rooted tree. For example, such queries are:

- what is the k th ancestor of a node?
- what is the sum of values in the subtree of a node?
- what is the sum of values on a path between two nodes?
- what is the lowest common ancestor of two nodes?

18.1 Finding ancestors

The k th **ancestor** of a node x in a rooted tree is the node that we will reach if we move k levels up from x . Let $\text{ancestor}(x, k)$ denote the k th ancestor of a node x (or 0 if there is no such an ancestor). For example, in the following tree, $\text{ancestor}(2, 1) = 1$ and $\text{ancestor}(8, 2) = 4$.



An easy way to calculate any value of $\text{ancestor}(x, k)$ is to perform a sequence of k moves in the tree. However, the time complexity of this method is $O(k)$, which may be slow, because a tree of n nodes may have a chain of n nodes.

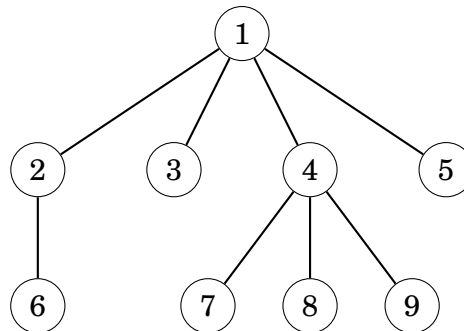
Fortunately, using a technique similar to that used in Chapter 16.3, any value of $\text{ancestor}(x, k)$ can be efficiently calculated in $O(\log k)$ time after preprocessing. The idea is to precalculate all values $\text{ancestor}(x, k)$ where $k \leq n$ is a power of two. For example, the values for the above tree are as follows:

x	1	2	3	4	5	6	7	8
$\text{ancestor}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestor}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestor}(x, 4)$	0	0	0	0	0	0	0	0
...								

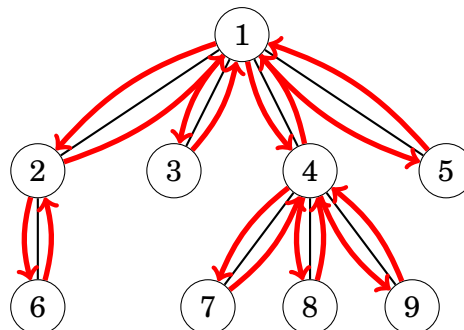
The preprocessing takes $O(n \log n)$ time, because $O(\log n)$ values are calculated for each node. After this, any value of $\text{ancestor}(x, k)$ can be calculated in $O(\log k)$ time by representing k as a sum where each term is a power of two.

18.2 Subtrees and paths

A **tree traversal array** contains the nodes of a rooted tree in the order in which a depth-first search from the root node visits them. For example, in the tree



a depth-first search proceeds as follows:



Hence, the corresponding tree traversal array is as follows:

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

Subtree queries

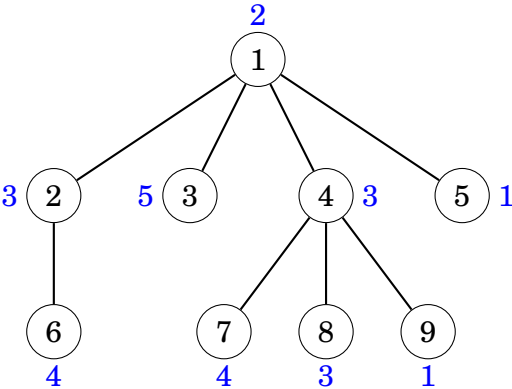
Each subtree of a tree corresponds to a subarray of the tree traversal array such that the first element of the subarray is the root node. For example, the following subarray contains the nodes of the subtree of node 4:

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

Using this fact, we can efficiently process queries that are related to subtrees of a tree. As an example, consider a problem where each node is assigned a value, and our task is to support the following queries:

- update the value of a node
- calculate the sum of values in the subtree of a node

Consider the following tree where the blue numbers are the values of the nodes. For example, the sum of the subtree of node 4 is $3 + 4 + 3 + 1 = 11$.



The idea is to construct a tree traversal array that contains three values for each node: the identifier of the node, the size of the subtree, and the value of the node. For example, the array for the above tree is as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

Using this array, we can calculate the sum of values in any subtree by first finding out the size of the subtree and then the values of the corresponding nodes. For example, the values in the subtree of node 4 can be found as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

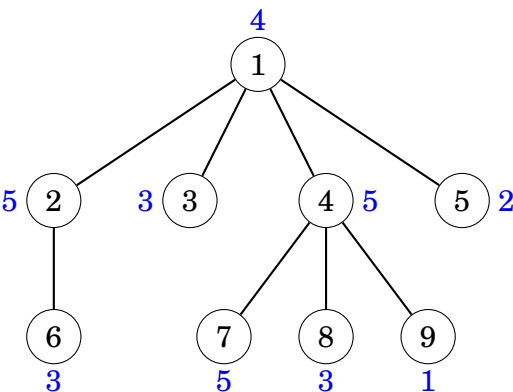
To answer the queries efficiently, it suffices to store the values of the nodes in a binary indexed or segment tree. After this, we can both update a value and calculate the sum of values in $O(\log n)$ time.

Path queries

Using a tree traversal array, we can also efficiently calculate sums of values on paths from the root node to any node of the tree. Consider a problem where our task is to support the following queries:

- change the value of a node
- calculate the sum of values on a path from the root to a node

For example, in the following tree, the sum of values from the root node to node 7 is $4 + 5 + 5 = 14$:



We can solve this problem like before, but now each value in the last row of the array is the sum of values on a path from the root to the node. For example, the following array corresponds to the above tree:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	9	14	12	10	6

When the value of a node increases by x , the sums of all nodes in its subtree increase by x . For example, if the value of node 4 increases by 1, the array changes as follows:

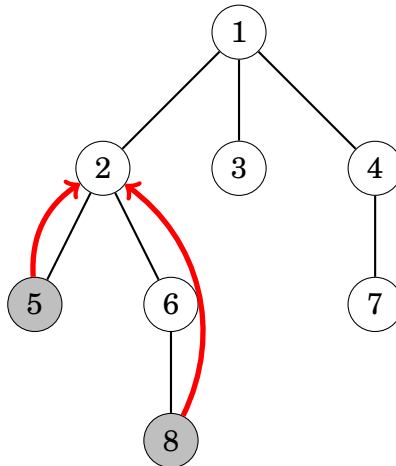
node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	10	15	13	11	6

Thus, to support both the operations, we should be able to increase all values in a range and retrieve a single value. This can be done in $O(\log n)$ time using a binary indexed or segment tree (see Chapter 9.4).

18.3 Lowest common ancestor

The **lowest common ancestor** of two nodes of a rooted tree is the lowest node whose subtree contains both the nodes. A typical problem is to efficiently process queries that ask to find the lowest common ancestor of two nodes.

For example, in the following tree, the lowest common ancestor of nodes 5 and 8 is node 2:



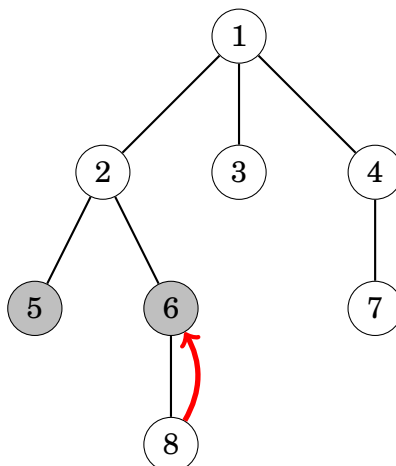
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

Method 1

One way to solve the problem is to use the fact that we can efficiently find the k th ancestor of any node in the tree. Using this, we can divide the problem of finding the lowest common ancestor into two parts.

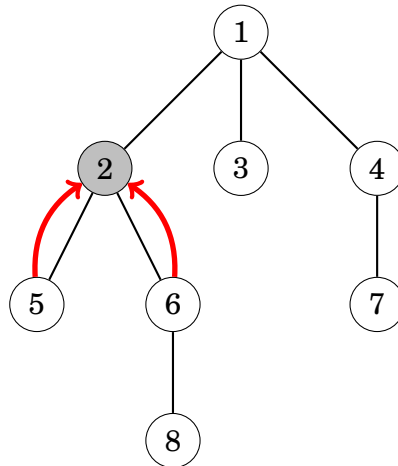
We use two pointers that initially point to the two nodes whose lowest common ancestor we should find. First, we move one of the pointers upwards so that both pointers point to nodes at the same level.

In the example scenario, we move the second pointer one level up so that it points to node 6 which is at the same level with node 5:



After this, we determine the minimum number of steps needed to move both pointers upwards so that they will point to the same node. The node to which the pointers point after this is the lowest common ancestor.

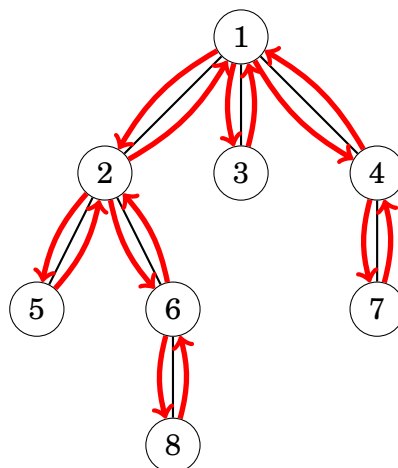
In the example scenario, it suffices to move both pointers one step upwards to node 2, which is the lowest common ancestor:



Since both parts of the algorithm can be performed in $O(\log n)$ time using precomputed information, we can find the lowest common ancestor of any two nodes in $O(\log n)$ time.

Method 2

Another way to solve the problem is based on a tree traversal array¹. Once again, the idea is to traverse the nodes using a depth-first search:



However, we use a different tree traversal array than before: we add each node to the array *always* when the depth-first search walks through the node, and not only at the first visit. Hence, a node that has k children appears $k + 1$ times in the array and there are a total of $2n - 1$ nodes in the array.

¹This lowest common ancestor algorithm was presented in [7]. This technique is sometimes called the **Euler tour technique** [66].

We store two values in the array: the identifier of the node and the depth of the node in the tree. The following array corresponds to the above tree:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Now we can find the lowest common ancestor of nodes a and b by finding the node with the *minimum* depth between nodes a and b in the array. For example, the lowest common ancestor of nodes 5 and 8 can be found as follows:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

Node 5 is at position 2, node 8 is at position 5, and the node with minimum depth between positions 2...5 is node 2 at position 3 whose depth is 2. Thus, the lowest common ancestor of nodes 5 and 8 is node 2.

Thus, to find the lowest common ancestor of two nodes it suffices to process a range minimum query. Since the array is static, we can process such queries in $O(1)$ time after an $O(n \log n)$ time preprocessing.

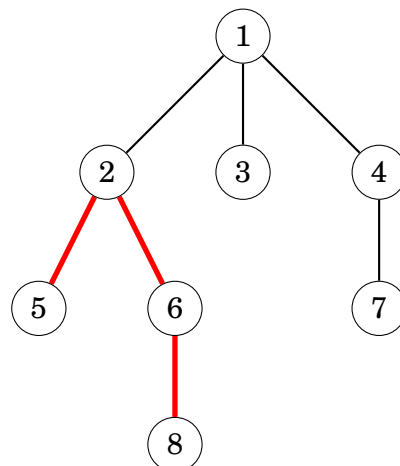
Distances of nodes

The distance between nodes a and b equals the length of the path from a to b . It turns out that the problem of calculating the distance between nodes reduces to finding their lowest common ancestor.

First, we root the tree arbitrarily. After this, the distance of nodes a and b can be calculated using the formula

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c),$$

where c is the lowest common ancestor of a and b and $\text{depth}(s)$ denotes the depth of node s . For example, consider the distance of nodes 5 and 8:



The lowest common ancestor of nodes 5 and 8 is node 2. The depths of the nodes are $\text{depth}(5) = 3$, $\text{depth}(8) = 4$ and $\text{depth}(2) = 2$, so the distance between nodes 5 and 8 is $3 + 4 - 2 \cdot 2 = 3$.

18.4 Offline algorithms

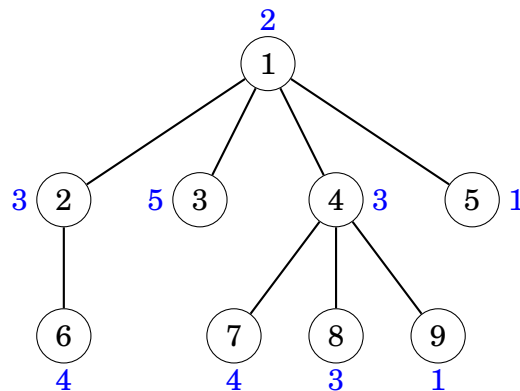
So far, we have discussed *online* algorithms for tree queries. Those algorithms are able to process queries one after another so that each query is answered before receiving the next query.

However, in many problems, the online property is not necessary. In this section, we focus on *offline* algorithms. Those algorithms are given a set of queries which can be answered in any order. It is often easier to design an offline algorithm compared to an online algorithm.

Merging data structures

One method to construct an offline algorithm is to perform a depth-first tree traversal and maintain data structures in nodes. At each node s , we create a data structure $d[s]$ that is based on the data structures of the children of s . Then, using this data structure, all queries related to s are processed.

As an example, consider the following problem: We are given a tree where each node has some value. Our task is to process queries of the form "calculate the number of nodes with value x in the subtree of node s ". For example, in the following tree, the subtree of node 4 contains two nodes whose value is 3.



In this problem, we can use map structures to answer the queries. For example, the maps for node 4 and its children are as follows:



If we create such a data structure for each node, we can easily process all given queries, because we can handle all queries related to a node immediately after creating its data structure. For example, the above map structure for node 4 tells us that its subtree contains two nodes whose value is 3.

However, it would be too slow to create all data structures from scratch. Instead, at each node s , we create an initial data structure $d[s]$ that only contains the value of s . After this, we go through the children of s and *merge* $d[s]$ and all data structures $d[u]$ where u is a child of s .

For example, in the above tree, the map for node 4 is created by merging the following maps:

$\frac{3}{1}$	$\frac{4}{1}$	$\frac{3}{1}$	$\frac{1}{1}$
---------------	---------------	---------------	---------------

Here the first map is the initial data structure for node 4, and the other three maps correspond to nodes 7, 8 and 9.

The merging at node s can be done as follows: We go through the children of s and at each child u merge $d[s]$ and $d[u]$. We always copy the contents from $d[u]$ to $d[s]$. However, before this, we *swap* the contents of $d[s]$ and $d[u]$ if $d[s]$ is smaller than $d[u]$. By doing this, each value is copied only $O(\log n)$ times during the tree traversal, which ensures that the algorithm is efficient.

To swap the contents of two data structures a and b efficiently, we can just use the following code:

```
swap(a,b);
```

It is guaranteed that the above code works in constant time when a and b are C++ standard library data structures.

Lowest common ancestors

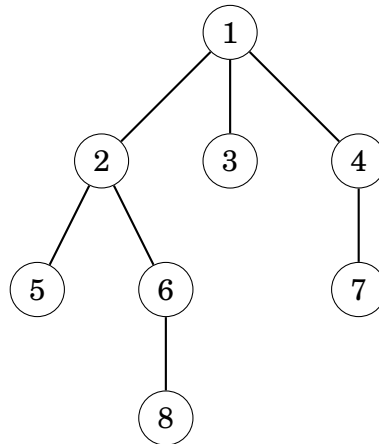
There is also an offline algorithm for processing a set of lowest common ancestor queries². The algorithm is based on the union-find data structure (see Chapter 15.2), and the benefit of the algorithm is that it is easier to implement than the algorithms discussed earlier in this chapter.

The algorithm is given as input a set of pairs of nodes, and it determines for each such pair the lowest common ancestor of the nodes. The algorithm performs a depth-first tree traversal and maintains disjoint sets of nodes. Initially, each node belongs to a separate set. For each set, we also store the highest node in the tree that belongs to the set.

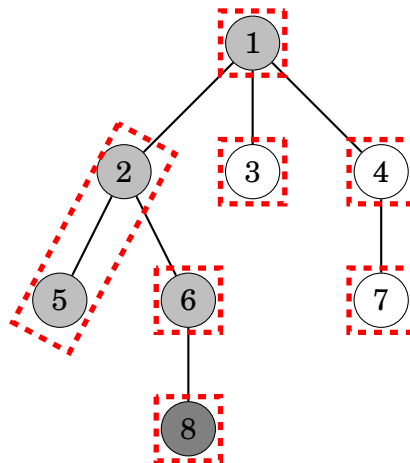
When the algorithm visits a node x , it goes through all nodes y such that the lowest common ancestor of x and y has to be found. If y has already been visited, the algorithm reports that the lowest common ancestor of x and y is the highest node in the set of y . Then, after processing node x , the algorithm joins the sets of x and its parent.

²This algorithm was published by R. E. Tarjan in 1979 [65].

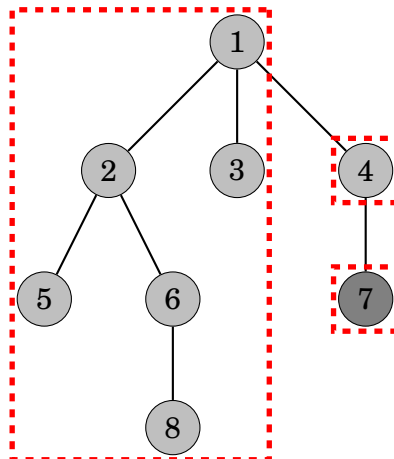
For example, suppose that we want to find the lowest common ancestors of node pairs (5,8) and (2,7) in the following tree:



In the following trees, gray nodes denote visited nodes and dashed groups of nodes belong to the same set. When the algorithm visits node 8, it notices that node 5 has been visited and the highest node in its set is 2. Thus, the lowest common ancestor of nodes 5 and 8 is 2:



Later, when visiting node 7, the algorithm determines that the lowest common ancestor of nodes 2 and 7 is 1:



Chapter 19

Paths and circuits

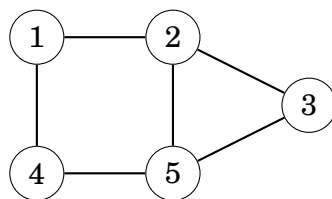
This chapter focuses on two types of paths in graphs:

- An **Eulerian path** is a path that goes through each edge exactly once.
- A **Hamiltonian path** is a path that visits each node exactly once.

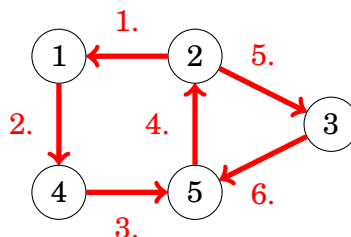
While Eulerian and Hamiltonian paths look like similar concepts at first glance, the computational problems related to them are very different. It turns out that there is a simple rule that determines whether a graph contains an Eulerian path, and there is also an efficient algorithm to find such a path if it exists. On the contrary, checking the existence of a Hamiltonian path is a NP-hard problem, and no efficient algorithm is known for solving the problem.

19.1 Eulerian paths

An **Eulerian path**¹ is a path that goes exactly once through each edge of the graph. For example, the graph

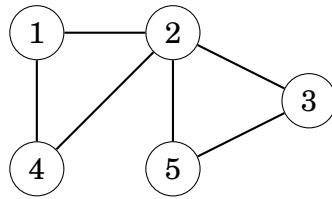


has an Eulerian path from node 2 to node 5:

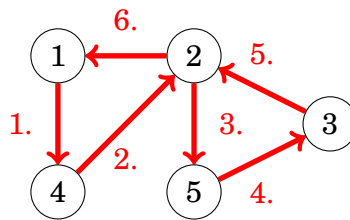


¹L. Euler studied such paths in 1736 when he solved the famous Königsberg bridge problem. This was the birth of graph theory.

An **Eulerian circuit** is an Eulerian path that starts and ends at the same node. For example, the graph



has an Eulerian circuit that starts and ends at node 1:



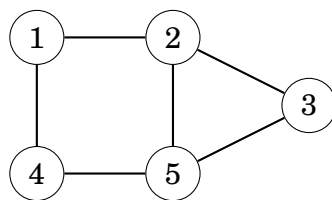
Existence

The existence of Eulerian paths and circuits depends on the degrees of the nodes. First, an undirected graph has an Eulerian path exactly when all the edges belong to the same connected component and

- the degree of each node is even *or*
- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path which is not an Eulerian circuit.

For example, in the graph



nodes 1, 3 and 4 have a degree of 2, and nodes 2 and 5 have a degree of 3. Exactly two nodes have an odd degree, so there is an Eulerian path between nodes 2 and 5, but the graph does not contain an Eulerian circuit.

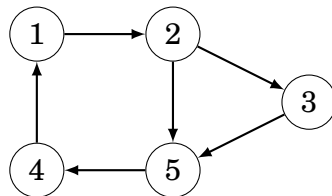
In a directed graph, we focus on indegrees and outdegrees of the nodes. A directed graph contains an Eulerian path exactly when all the edges belong to the same connected component and

- in each node, the indegree equals the outdegree, *or*

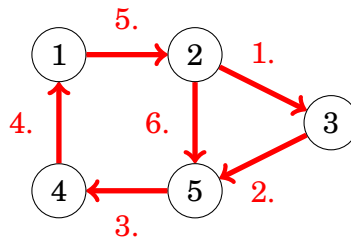
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph contains an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger.

For example, in the graph



nodes 1, 3 and 4 have both indegree 1 and outdegree 1, node 2 has indegree 1 and outdegree 2, and node 5 has indegree 2 and outdegree 1. Hence, the graph contains an Eulerian path from node 2 to node 5:



Hierholzer's algorithm

Hierholzer's algorithm² is an efficient method for constructing an Eulerian circuit. The algorithm consists of several rounds, each of which adds new edges to the circuit. Of course, we assume that the graph contains an Eulerian circuit; otherwise Hierholzer's algorithm cannot find it.

First, the algorithm constructs a circuit that contains some (not necessarily all) of the edges of the graph. After this, the algorithm extends the circuit step by step by adding subcircuits to it. The process continues until all edges have been added to the circuit.

The algorithm extends the circuit by always finding a node x that belongs to the circuit but has an outgoing edge that is not included in the circuit. The algorithm constructs a new path from node x that only contains edges that are not yet in the circuit. Sooner or later, the path will return to node x , which creates a subcircuit.

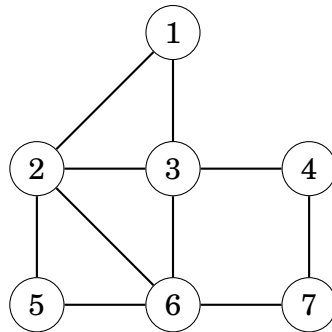
If the graph only contains an Eulerian path, we can still use Hierholzer's algorithm to find it by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

Next we will see how Hierholzer's algorithm constructs an Eulerian circuit for an undirected graph.

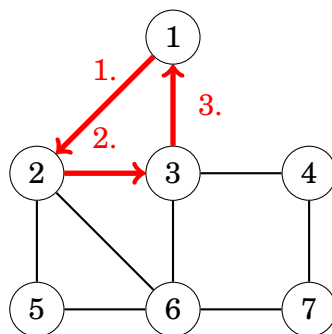
²The algorithm was published in 1873 after Hierholzer's death [35].

Example

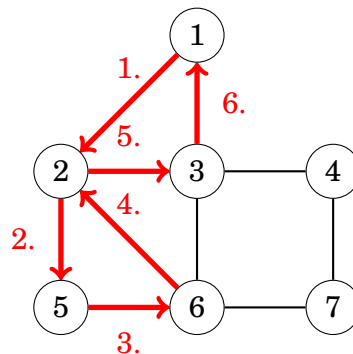
Let us consider the following graph:



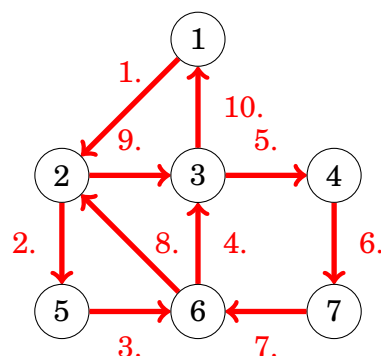
Suppose that the algorithm first creates a circuit that begins at node 1. A possible circuit is $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



After this, the algorithm adds the subcircuit $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ to the circuit:



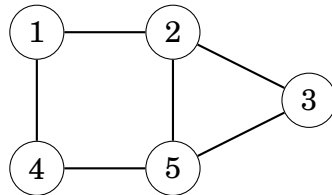
Finally, the algorithm adds the subcircuit $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ to the circuit:



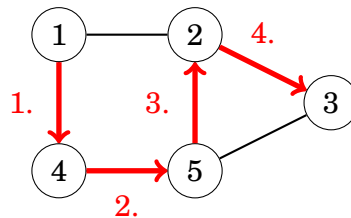
Now all edges are included in the circuit, so we have successfully constructed an Eulerian circuit.

19.2 Hamiltonian paths

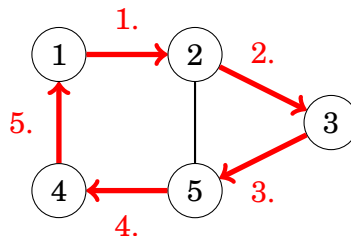
A **Hamiltonian path** is a path that visits each node of the graph exactly once. For example, the graph



contains a Hamiltonian path from node 1 to node 3:



If a Hamiltonian path begins and ends at the same node, it is called a **Hamiltonian circuit**. The graph above also has an Hamiltonian circuit that begins and ends at node 1:



Existence

No efficient method is known for testing if a graph contains a Hamiltonian path, and the problem is NP-hard. Still, in some special cases, we can be certain that a graph contains a Hamiltonian path.

A simple observation is that if the graph is complete, i.e., there is an edge between all pairs of nodes, it also contains a Hamiltonian path. Also stronger results have been achieved:

- **Dirac's theorem:** If the degree of each node is at least $n/2$, the graph contains a Hamiltonian path.
- **Ore's theorem:** If the sum of degrees of each non-adjacent pair of nodes is at least n , the graph contains a Hamiltonian path.

A common property in these theorems and other results is that they guarantee the existence of a Hamiltonian path if the graph has *a large number* of edges. This makes sense, because the more edges the graph contains, the more possibilities there is to construct a Hamiltonian path.

Construction

Since there is no efficient way to check if a Hamiltonian path exists, it is clear that there is also no method to efficiently construct the path, because otherwise we could just try to construct the path and see whether it exists.

A simple way to search for a Hamiltonian path is to use a backtracking algorithm that goes through all possible ways to construct the path. The time complexity of such an algorithm is at least $O(n!)$, because there are $n!$ different ways to choose the order of n nodes.

A more efficient solution is based on dynamic programming (see Chapter 10.5). The idea is to calculate values of a function $\text{possible}(S, x)$, where S is a subset of nodes and x is one of the nodes. The function indicates whether there is a Hamiltonian path that visits the nodes of S and ends at node x . It is possible to implement this solution in $O(2^n n^2)$ time.

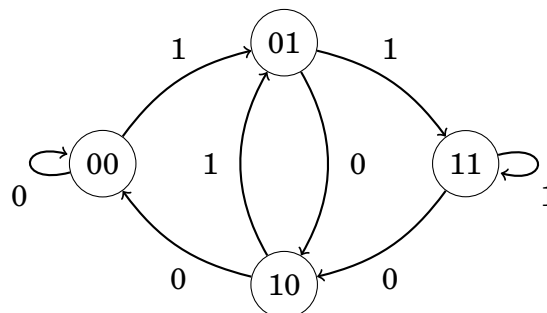
19.3 De Bruijn sequences

A **De Bruijn sequence** is a string that contains every string of length n exactly once as a substring, for a fixed alphabet of k characters. The length of such a string is $k^n + n - 1$ characters. For example, when $n = 3$ and $k = 2$, an example of a De Bruijn sequence is

0001011100.

The substrings of this string are all combinations of three bits: 000, 001, 010, 011, 100, 101, 110 and 111.

It turns out that each De Bruijn sequence corresponds to an Eulerian path in a graph. The idea is to construct a graph where each node contains a string of $n - 1$ characters and each edge adds one character to the string. The following graph corresponds to the above scenario:



An Eulerian path in this graph corresponds to a string that contains all strings of length n . The string contains the characters of the starting node and all characters of the edges. The starting node has $n - 1$ characters and there are k^n characters in the edges, so the length of the string is $k^n + n - 1$.

19.4 Knight's tours

A **knight's tour** is a sequence of moves of a knight on an $n \times n$ chessboard following the rules of chess such that the knight visits each square exactly once. A knight's tour is called a *closed* tour if the knight finally returns to the starting square and otherwise it is called an *open* tour.

For example, here is an open knight's tour on a 5×5 board:

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

A knight's tour corresponds to a Hamiltonian path in a graph whose nodes represent the squares of the board, and two nodes are connected with an edge if a knight can move between the squares according to the rules of chess.

A natural way to construct a knight's tour is to use backtracking. The search can be made more efficient by using *heuristics* that attempt to guide the knight so that a complete tour will be found quickly.

Warnsdorf's rule

Warnsdorf's rule is a simple and effective heuristic for finding a knight's tour³. Using the rule, it is possible to efficiently construct a tour even on a large board. The idea is to always move the knight so that it ends up in a square where the number of possible moves is as *small* as possible.

For example, in the following situation, there are five possible squares to which the knight can move (squares $a \dots e$):

1				a
		2		
b				e
	c		d	

In this situation, Warnsdorf's rule moves the knight to square a , because after this choice, there is only a single possible move. The other choices would move the knight to squares where there would be three moves available.

³This heuristic was proposed in Warnsdorf's book [69] in 1823. There are also polynomial algorithms for finding knight's tours [52], but they are more complicated.

Chapter 20

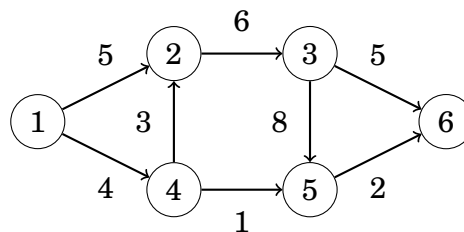
Flows and cuts

In this chapter, we focus on the following two problems:

- **Finding a maximum flow:** What is the maximum amount of flow we can send from a node to another node?
- **Finding a minimum cut:** What is a minimum-weight set of edges that separates two nodes of the graph?

The input for both these problems is a directed, weighted graph that contains two special nodes: the *source* is a node with no incoming edges, and the *sink* is a node with no outgoing edges.

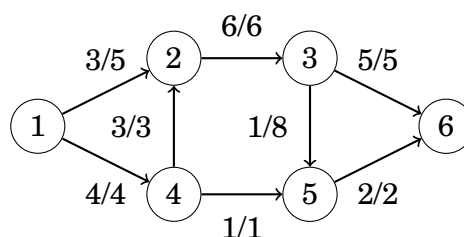
As an example, we will use the following graph where node 1 is the source and node 6 is the sink:



Maximum flow

In the **maximum flow** problem, our task is to send as much flow as possible from the source to the sink. The weight of each edge is a capacity that restricts the flow that can go through the edge. In each intermediate node, the incoming and outgoing flow has to be equal.

For example, the maximum size of a flow in the example graph is 7. The following picture shows how we can route the flow:

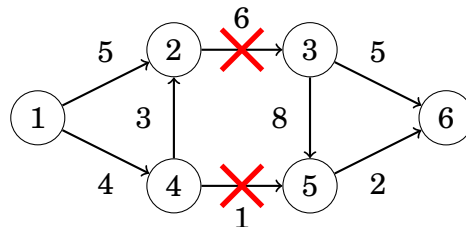


The notation v/k means that a flow of v units is routed through an edge whose capacity is k units. The size of the flow is 7, because the source sends $3 + 4$ units of flow and the sink receives $5 + 2$ units of flow. It is easy to see that this flow is maximum, because the total capacity of the edges leading to the sink is 7.

Minimum cut

In the **minimum cut** problem, our task is to remove a set of edges from the graph such that there will be no path from the source to the sink after the removal and the total weight of the removed edges is minimum.

The minimum size of a cut in the example graph is 7. It suffices to remove the edges $2 \rightarrow 3$ and $4 \rightarrow 5$:



After removing the edges, there will be no path from the source to the sink. The size of the cut is 7, because the weights of the removed edges are 6 and 1. The cut is minimum, because there is no valid way to remove edges from the graph such that their total weight would be less than 7.

It is not a coincidence that the maximum size of a flow and the minimum size of a cut are the same in the above example. It turns out that a maximum flow and a minimum cut are *always* equally large, so the concepts are two sides of the same coin.

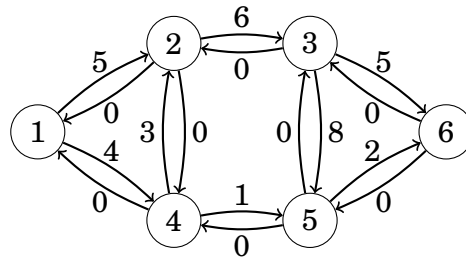
Next we will discuss the Ford–Fulkerson algorithm that can be used to find the maximum flow and minimum cut of a graph. The algorithm also helps us to understand *why* they are equally large.

20.1 Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** [25] finds the maximum flow in a graph. The algorithm begins with an empty flow, and at each step finds a path from the source to the sink that generates more flow. Finally, when the algorithm cannot increase the flow anymore, the maximum flow has been found.

The algorithm uses a special representation of the graph where each original edge has a reverse edge in another direction. The weight of each edge indicates how much more flow we could route through it. At the beginning of the algorithm, the weight of each original edge equals the capacity of the edge and the weight of each reverse edge is zero.

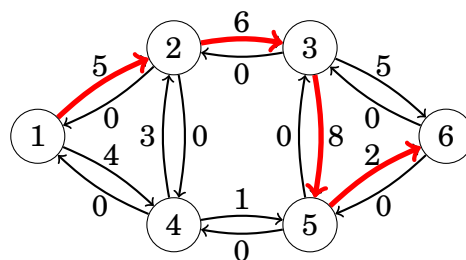
The new representation for the example graph is as follows:



Algorithm description

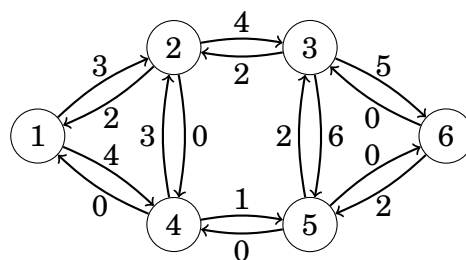
The Ford–Fulkerson algorithm consists of several rounds. On each round, the algorithm finds a path from the source to the sink such that each edge on the path has a positive weight. If there is more than one possible path available, we can choose any of them.

For example, suppose we choose the following path:



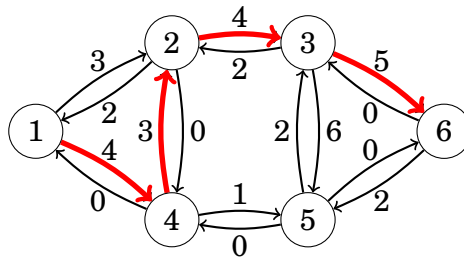
After choosing the path, the flow increases by x units, where x is the smallest edge weight on the path. In addition, the weight of each edge on the path decreases by x and the weight of each reverse edge increases by x .

In the above path, the weights of the edges are 5, 6, 8 and 2. The smallest weight is 2, so the flow increases by 2 and the new graph is as follows:



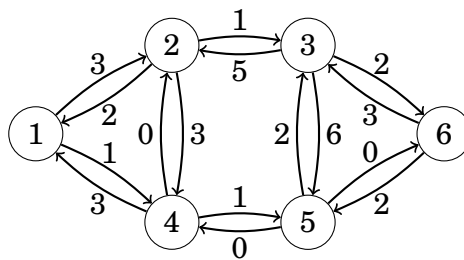
The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to cancel flow later using the reverse edges of the graph if it turns out that it would be beneficial to route the flow in another way.

The algorithm increases the flow as long as there is a path from the source to the sink through positive-weight edges. In the present example, our next path can be as follows:

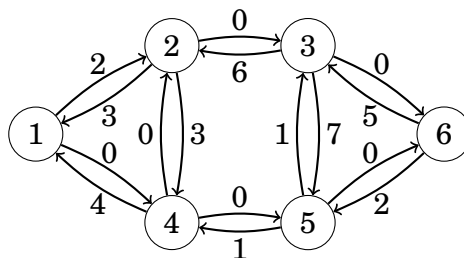


The minimum edge weight on this path is 3, so the path increases the flow by 3, and the total flow after processing the path is 5.

The new graph will be as follows:



We still need two more rounds before reaching the maximum flow. For example, we can choose the paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ and $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$. Both paths increase the flow by 1, and the final graph is as follows:



It is not possible to increase the flow anymore, because there is no path from the source to the sink with positive edge weights. Hence, the algorithm terminates and the maximum flow is 7.

Finding paths

The Ford–Fulkerson algorithm does not specify how we should choose the paths that increase the flow. In any case, the algorithm will terminate sooner or later and correctly find the maximum flow. However, the efficiency of the algorithm depends on the way the paths are chosen.

A simple way to find paths is to use depth-first search. Usually, this works well, but in the worst case, each path only increases the flow by 1 and the algorithm is slow. Fortunately, we can avoid this situation by using one of the following techniques:

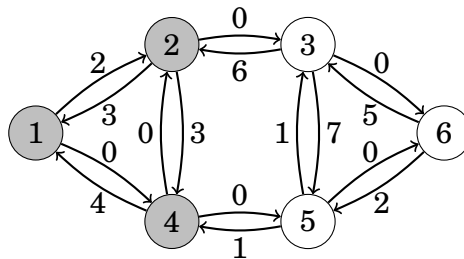
The **Edmonds–Karp algorithm** [18] chooses each path so that the number of edges on the path is as small as possible. This can be done by using breadth-first search instead of depth-first search for finding paths. It can be proven that this guarantees that the flow increases quickly, and the time complexity of the algorithm is $O(m^2n)$.

The **scaling algorithm** [2] uses depth-first search to find paths where each edge weight is at least a threshold value. Initially, the threshold value is some large number, for example the sum of all edge weights of the graph. Always when a path cannot be found, the threshold value is divided by 2. The time complexity of the algorithm is $O(m^2 \log c)$, where c is the initial threshold value.

In practice, the scaling algorithm is easier to implement, because depth-first search can be used for finding paths. Both algorithms are efficient enough for problems that typically appear in programming contests.

Minimum cuts

It turns out that once the Ford–Fulkerson algorithm has found a maximum flow, it has also determined a minimum cut. Let A be the set of nodes that can be reached from the source using positive-weight edges. In the example graph, A contains nodes 1, 2 and 4:



Now the minimum cut consists of the edges of the original graph that start at some node in A , end at some node outside A , and whose capacity is fully used in the maximum flow. In the above graph, such edges are $2 \rightarrow 3$ and $4 \rightarrow 5$, that correspond to the minimum cut $6 + 1 = 7$.

Why is the flow produced by the algorithm maximum and why is the cut minimum? The reason is that a graph cannot contain a flow whose size is larger than the weight of any cut of the graph. Hence, always when a flow and a cut are equally large, they are a maximum flow and a minimum cut.

Let us consider any cut of the graph such that the source belongs to A , the sink belongs to B and there are some edges between the sets:



The size of the cut is the sum of the edges that go from A to B . This is an upper bound for the flow in the graph, because the flow has to proceed from A to B . Thus, the size of a maximum flow is smaller than or equal to the size of any cut in the graph.

On the other hand, the Ford–Fulkerson algorithm produces a flow whose size is *exactly* as large as the size of a cut in the graph. Thus, the flow has to be a maximum flow and the cut has to be a minimum cut.

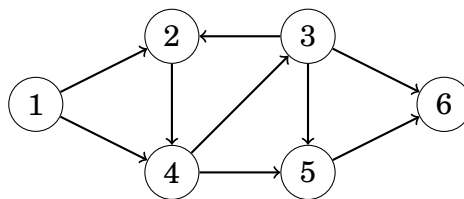
20.2 Disjoint paths

Many graph problems can be solved by reducing them to the maximum flow problem. Our first example of such a problem is as follows: we are given a directed graph with a source and a sink, and our task is to find the maximum number of disjoint paths from the source to the sink.

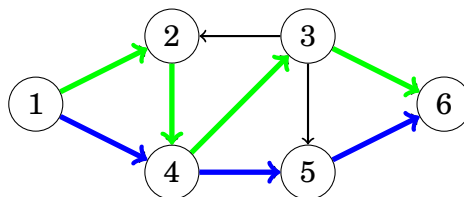
Edge-disjoint paths

We will first focus on the problem of finding the maximum number of **edge-disjoint paths** from the source to the sink. This means that we should construct a set of paths such that each edge appears in at most one path.

For example, consider the following graph:



In this graph, the maximum number of edge-disjoint paths is 2. We can choose the paths $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ and $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ as follows:



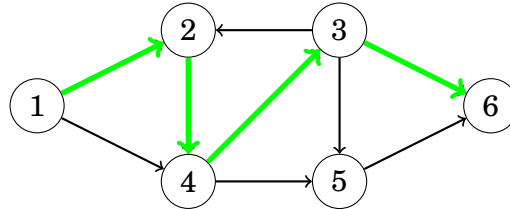
It turns out that the maximum number of edge-disjoint paths equals the maximum flow of the graph, assuming that the capacity of each edge is one. After the maximum flow has been constructed, the edge-disjoint paths can be found greedily by following paths from the source to the sink.

Node-disjoint paths

Let us now consider another problem: finding the maximum number of **node-disjoint paths** from the source to the sink. In this problem, every node, except

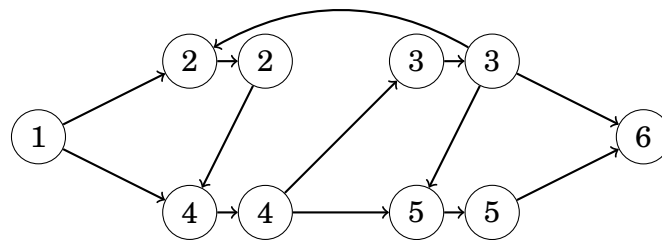
for the source and sink, may appear in at most one path. The number of node-disjoint paths may be smaller than the number of edge-disjoint paths.

For example, in the previous graph, the maximum number of node-disjoint paths is 1:

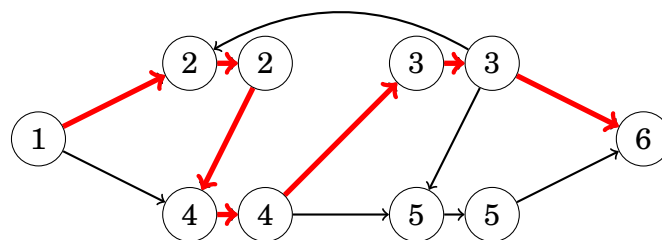


We can reduce also this problem to the maximum flow problem. Since each node can appear in at most one path, we have to limit the flow that goes through the nodes. A standard method for this is to divide each node into two nodes such that the first node has the incoming edges of the original node, the second node has the outgoing edges of the original node, and there is a new edge from the first node to the second node.

In our example, the graph becomes as follows:



The maximum flow for the graph is as follows:



Thus, the maximum number of node-disjoint paths from the source to the sink is 1.

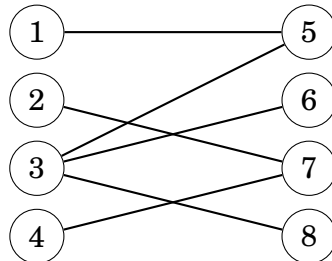
20.3 Maximum matchings

The **maximum matching** problem asks to find a maximum-size set of node pairs in an undirected graph such that each pair is connected with an edge and each node belongs to at most one pair.

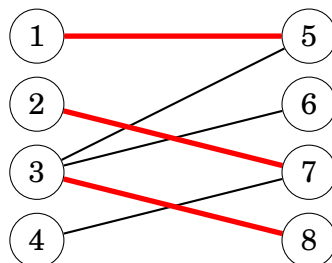
There are polynomial algorithms for finding maximum matchings in general graphs [17], but such algorithms are complex and rarely seen in programming contests. However, in bipartite graphs, the maximum matching problem is much easier to solve, because we can reduce it to the maximum flow problem.

Finding maximum matchings

The nodes of a bipartite graph can be always divided into two groups such that all edges of the graph go from the left group to the right group. For example, in the following bipartite graph, the groups are $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8\}$.

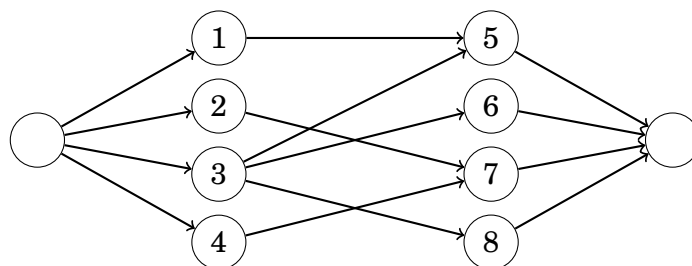


The size of a maximum matching of this graph is 3:

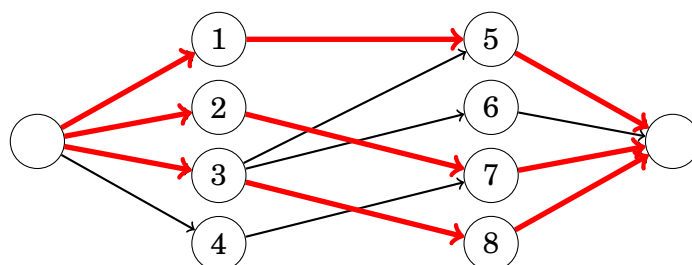


We can reduce the bipartite maximum matching problem to the maximum flow problem by adding two new nodes to the graph: a source and a sink. We also add edges from the source to each left node and from each right node to the sink. After this, the size of a maximum flow in the graph equals the size of a maximum matching in the original graph.

For example, the reduction for the above graph is as follows:



The maximum flow of this graph is as follows:

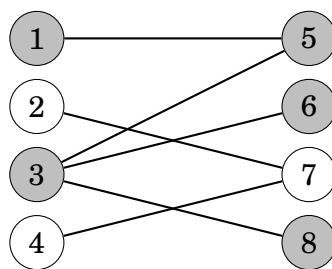


Hall's theorem

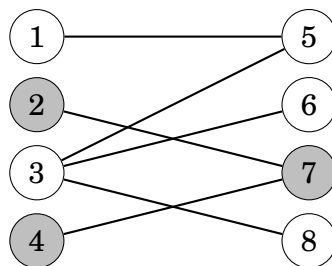
Hall's theorem can be used to find out whether a bipartite graph has a matching that contains all left or right nodes. If the number of left and right nodes is the same, Hall's theorem tells us if it is possible to construct a **perfect matching** that contains all nodes of the graph.

Assume that we want to find a matching that contains all left nodes. Let X be any set of left nodes and let $f(X)$ be the set of their neighbors. According to Hall's theorem, a matching that contains all left nodes exists exactly when for each X , the condition $|X| \leq |f(X)|$ holds.

Let us study Hall's theorem in the example graph. First, let $X = \{1, 3\}$ which yields $f(X) = \{5, 6, 8\}$:



The condition of Hall's theorem holds, because $|X| = 2$ and $|f(X)| = 3$. Next, let $X = \{2, 4\}$ which yields $f(X) = \{7\}$:



In this case, $|X| = 2$ and $|f(X)| = 1$, so the condition of Hall's theorem does not hold. This means that it is not possible to form a perfect matching for the graph. This result is not surprising, because we already know that the maximum matching of the graph is 3 and not 4.

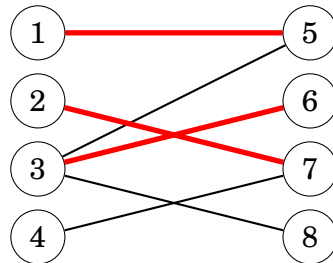
If the condition of Hall's theorem does not hold, the set X provides an explanation *why* we cannot form such a matching. Since X contains more nodes than $f(X)$, there are no pairs for all nodes in X . For example, in the above graph, both nodes 2 and 4 should be connected with node 7 which is not possible.

Kőnig's theorem

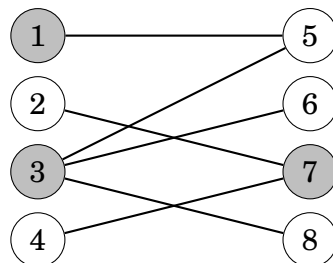
A **minimum node cover** of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set. In a general graph, finding a minimum node cover is a NP-hard problem. However, if the graph is bipartite, **Kőnig's theorem** tells us that the size of a minimum node cover and the size

of a maximum matching are always equal. Thus, we can calculate the size of a minimum node cover using a maximum flow algorithm.

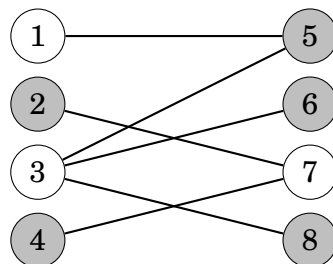
Let us consider the following graph with a maximum matching of size 3:



Now König's theorem tells us that the size of a minimum node cover is also 3. Such a cover can be constructed as follows:



The nodes that do *not* belong to a minimum node cover form a **maximum independent set**. This is the largest possible set of nodes such that no two nodes in the set are connected with an edge. Once again, finding a maximum independent set in a general graph is a NP-hard problem, but in a bipartite graph we can use König's theorem to solve the problem efficiently. In the example graph, the maximum independent set is as follows:

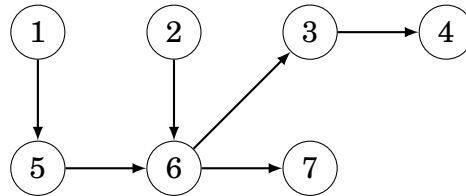


20.4 Path covers

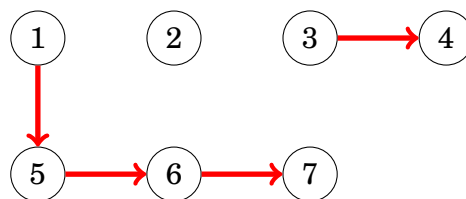
A **path cover** is a set of paths in a graph such that each node of the graph belongs to at least one path. It turns out that in directed, acyclic graphs, we can reduce the problem of finding a minimum path cover to the problem of finding a maximum flow in another graph.

Node-disjoint path cover

In a **node-disjoint path cover**, each node belongs to exactly one path. As an example, consider the following graph:



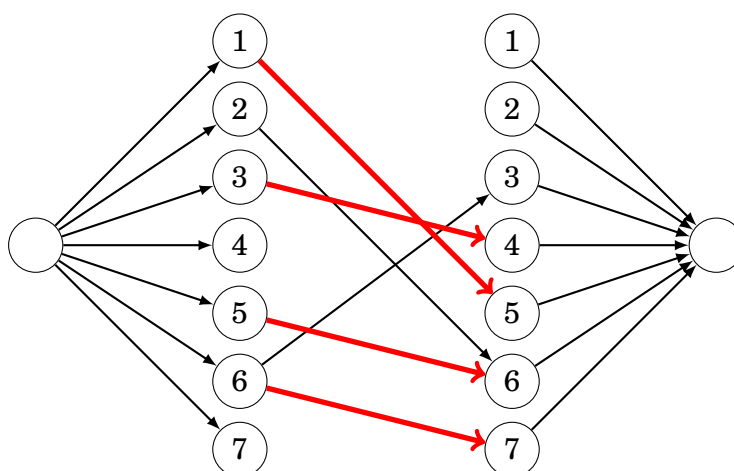
A minimum node-disjoint path cover of this graph consists of three paths. For example, we can choose the following paths:



Note that one of the paths only contains node 2, so it is possible that a path does not contain any edges.

We can find a minimum node-disjoint path cover by constructing a *matching graph* where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is such an edge in the original graph. In addition, the matching graph contains a source and a sink, and there are edges from the source to all left nodes and from all right nodes to the sink.

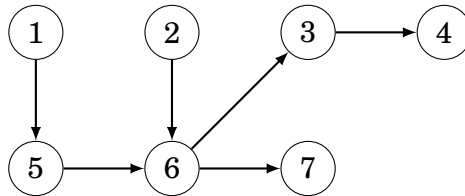
A maximum matching in the resulting graph corresponds to a minimum node-disjoint path cover in the original graph. For example, the following matching graph for the above graph contains a maximum matching of size 4:



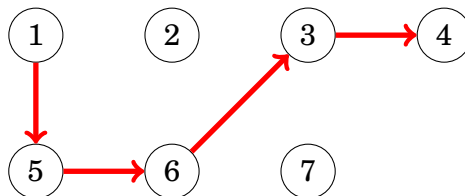
Each edge in the maximum matching of the matching graph corresponds to an edge in the minimum node-disjoint path cover of the original graph. Thus, the size of the minimum node-disjoint path cover is $n - c$, where n is the number of nodes in the original graph and c is the size of the maximum matching.

General path cover

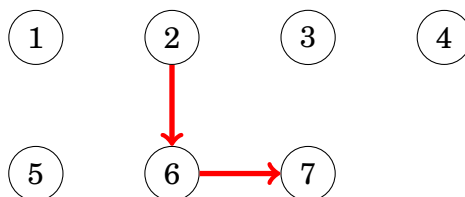
A **general path cover** is a path cover where a node can belong to more than one path. A minimum general path cover may be smaller than a minimum node-disjoint path cover, because a node can be used multiple times in paths. Consider again the following graph:



The minimum general path cover of this graph consists of two paths. For example, the first path may be as follows:

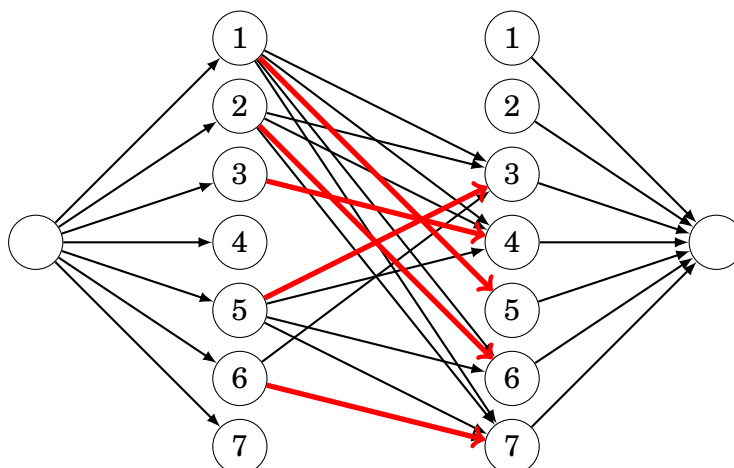


And the second path may be as follows:



A minimum general path cover can be found almost like a minimum node-disjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge $a \rightarrow b$ always when there is a path from a to b in the original graph (possibly through several edges).

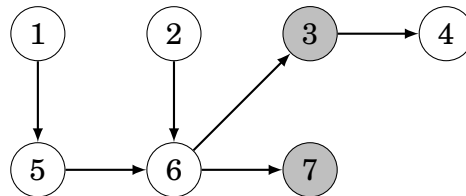
The matching graph for the above graph is as follows:



Dilworth's theorem

An **antichain** is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. **Dilworth's theorem** states that in a directed acyclic graph, the size of a minimum general path cover equals the size of a maximum antichain.

For example, nodes 3 and 7 form an antichain in the following graph:



This is a maximum antichain, because it is not possible to construct any antichain that would contain three nodes. We have seen before that the size of a minimum general path cover of this graph consists of two paths.

Part III

Advanced topics

Chapter 21

Number theory

Number theory is a branch of mathematics that studies integers. Number theory is a fascinating field, because many questions involving integers are very difficult to solve even if they seem simple at first glance.

As an example, consider the following equation:

$$x^3 + y^3 + z^3 = 33$$

It is easy to find three real numbers x , y and z that satisfy the equation. For example, we can choose

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

However, it is an open problem in number theory if there are any three *integers* x , y and z that would satisfy the equation [6].

In this chapter, we will focus on basic concepts and algorithms in number theory. Throughout the chapter, we will assume that all numbers are integers, if not otherwise stated.

21.1 Primes and factors

A number a is called a **factor** or a **divisor** of a number b if a divides b . If a is a factor of b , we write $a \mid b$, and otherwise we write $a \nmid b$. For example, the factors of 24 are 1, 2, 3, 4, 6, 8, 12 and 24.

A number $n > 1$ is a **prime** if its only positive factors are 1 and n . For example, 7, 19 and 41 are primes, but 35 is not a prime, because $5 \cdot 7 = 35$. For every number $n > 1$, there is a unique **prime factorization**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

where p_1, p_2, \dots, p_k are distinct primes and $\alpha_1, \alpha_2, \dots, \alpha_k$ are positive numbers. For example, the prime factorization for 84 is

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

The **number of factors** of a number n is

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

because for each prime p_i , there are $\alpha_i + 1$ ways to choose how many times it appears in the factor. For example, the number of factors of 84 is $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. The factors are 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 and 84.

The **sum of factors** of n is

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

where the latter formula is based on the geometric progression formula. For example, the sum of factors of 84 is

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

The **product of factors** of n is

$$\mu(n) = n^{\tau(n)/2},$$

because we can form $\tau(n)/2$ pairs from the factors, each with product n . For example, the factors of 84 produce the pairs $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$, etc., and the product of the factors is $\mu(84) = 84^6 = 351298031616$.

A number n is called a **perfect number** if $n = \sigma(n) - n$, i.e., n equals the sum of its factors between 1 and $n - 1$. For example, 28 is a perfect number, because $28 = 1 + 2 + 4 + 7 + 14$.

Number of primes

It is easy to show that there is an infinite number of primes. If the number of primes would be finite, we could construct a set $P = \{p_1, p_2, \dots, p_n\}$ that would contain all the primes. For example, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and so on. However, using P , we could form a new prime

$$p_1 p_2 \cdots p_n + 1$$

that is larger than all elements in P . This is a contradiction, and the number of primes has to be infinite.

Density of primes

The density of primes means how often there are primes among the numbers. Let $\pi(n)$ denote the number of primes between 1 and n . For example, $\pi(10) = 4$, because there are 4 primes between 1 and 10: 2, 3, 5 and 7.

It is possible to show that

$$\pi(n) \approx \frac{n}{\ln n},$$

which means that primes are quite frequent. For example, the number of primes between 1 and 10^6 is $\pi(10^6) = 78498$, and $10^6 / \ln 10^6 \approx 72382$.

Conjectures

There are many *conjectures* involving primes. Most people think that the conjectures are true, but nobody has been able to prove them. For example, the following conjectures are famous:

- **Goldbach's conjecture:** Each even integer $n > 2$ can be represented as a sum $n = a + b$ so that both a and b are primes.
- **Twin prime conjecture:** There is an infinite number of pairs of the form $\{p, p + 2\}$, where both p and $p + 2$ are primes.
- **Legendre's conjecture:** There is always a prime between numbers n^2 and $(n + 1)^2$, where n is any positive integer.

Basic algorithms

If a number n is not prime, it can be represented as a product $a \cdot b$, where $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$, so it certainly has a factor between 2 and $\lfloor \sqrt{n} \rfloor$. Using this observation, we can both test if a number is prime and find the prime factorization of a number in $O(\sqrt{n})$ time.

The following function `prime` checks if the given number n is prime. The function attempts to divide n by all numbers between 2 and $\lfloor \sqrt{n} \rfloor$, and if none of them divides n , then n is prime.

```
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;
}
```

The following function `factors` constructs a vector that contains the prime factorization of n . The function divides n by its prime factors, and adds them to the vector. The process ends when the remaining number n has no factors between 2 and $\lfloor \sqrt{n} \rfloor$. If $n > 1$, it is prime and the last 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;
}
```

Note that each prime factor appears in the vector as many times as it divides the number. For example, $24 = 2^3 \cdot 3$, so the result of the function is $[2, 2, 2, 3]$.

Sieve of Eratosthenes

The **sieve of Eratosthenes** is a preprocessing algorithm that builds an array using which we can efficiently check if a given number between $2 \dots n$ is prime and, if it is not, find one prime factor of the number.

The algorithm builds an array `sieve` whose positions $2, 3, \dots, n$ are used. The value `sieve[k] = 0` means that k is prime, and the value `sieve[k] ≠ 0` means that k is not a prime and one of its prime factors is `sieve[k]`.

The algorithm iterates through the numbers $2 \dots n$ one by one. Always when a new prime x is found, the algorithm records that the multiples of x ($2x, 3x, 4x, \dots$) are not primes, because the number x divides them.

For example, if $n = 20$, the array is as follows:

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

The following code implements the sieve of Eratosthenes. The code assumes that each element of `sieve` is initially zero.

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}
```

The inner loop of the algorithm is executed n/x times for each value of x . Thus, an upper bound for the running time of the algorithm is the harmonic sum

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

In fact, the algorithm is more efficient, because the inner loop will be executed only if the number x is prime. It can be shown that the running time of the algorithm is only $O(n \log \log n)$, a complexity very near to $O(n)$.

Euclid's algorithm

The **greatest common divisor** of numbers a and b , $\gcd(a, b)$, is the greatest number that divides both a and b , and the **least common multiple** of a and b , $\text{lcm}(a, b)$, is the smallest number that is divisible by both a and b . For example, $\gcd(24, 36) = 12$ and $\text{lcm}(24, 36) = 72$.

The greatest common divisor and the least common multiple are connected as follows:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

Euclid's algorithm¹ provides an efficient way to find the greatest common divisor of two numbers. The algorithm is based on the following formula:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

For example,

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

The algorithm can be implemented as follows:

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

It can be shown that Euclid's algorithm works in $O(\log n)$ time, where $n = \min(a, b)$. The worst case for the algorithm is the case when a and b are consecutive Fibonacci numbers. For example,

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

Euler's totient function

Numbers a and b are **coprime** if $\gcd(a, b) = 1$. **Euler's totient function** $\varphi(n)$ gives the number of coprime numbers to n between 1 and n . For example, $\varphi(12) = 4$, because 1, 5, 7 and 11 are coprime to 12.

The value of $\varphi(n)$ can be calculated from the prime factorization of n using the formula

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1).$$

For example, $\varphi(12) = 2^1 \cdot (2-1) \cdot 3^0 \cdot (3-1) = 4$. Note that $\varphi(n) = n-1$ if n is prime.

21.2 Modular arithmetic

In **modular arithmetic**, the set of numbers is limited so that only numbers $0, 1, 2, \dots, m-1$ are used, where m is a constant. Each number x is represented by the number $x \bmod m$: the remainder after dividing x by m . For example, if $m = 17$, then 75 is represented by $75 \bmod 17 = 7$.

Often we can take remainders before doing calculations. In particular, the following formulas hold:

$$\begin{aligned} (x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\ (x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\ (x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\ x^n \bmod m &= (x \bmod m)^n \bmod m \end{aligned}$$

¹Euclid was a Greek mathematician who lived in about 300 BC. This is perhaps the first known algorithm in history.

Modular exponentiation

There is often need to efficiently calculate the value of $x^n \bmod m$. This can be done in $O(\log n)$ time using the following recursion:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

It is important that in the case of an even n , the value of $x^{n/2}$ is calculated only once. This guarantees that the time complexity of the algorithm is $O(\log n)$, because n is always halved when it is even.

The following function calculates the value of $x^n \bmod m$:

```
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;
}
```

Fermat's theorem and Euler's theorem

Fermat's theorem states that

$$x^{m-1} \bmod m = 1$$

when m is prime and x and m are coprime. This also yields

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

More generally, **Euler's theorem** states that

$$x^{\varphi(m)} \bmod m = 1$$

when x and m are coprime. Fermat's theorem follows from Euler's theorem, because if m is a prime, then $\varphi(m) = m - 1$.

Modular inverse

The inverse of x modulo m is a number x^{-1} such that

$$xx^{-1} \bmod m = 1.$$

For example, if $x = 6$ and $m = 17$, then $x^{-1} = 3$, because $6 \cdot 3 \bmod 17 = 1$.

Using modular inverses, we can divide numbers modulo m , because division by x corresponds to multiplication by x^{-1} . For example, to evaluate the value

of $36/6 \bmod 17$, we can use the formula $2 \cdot 3 \bmod 17$, because $36 \bmod 17 = 2$ and $6^{-1} \bmod 17 = 3$.

However, a modular inverse does not always exist. For example, if $x = 2$ and $m = 4$, the equation

$$xx^{-1} \bmod m = 1$$

cannot be solved, because all multiples of 2 are even and the remainder can never be 1 when $m = 4$. It turns out that the value of $x^{-1} \bmod m$ can be calculated exactly when x and m are coprime.

If a modular inverse exists, it can be calculated using the formula

$$x^{-1} = x^{\varphi(m)-1}.$$

If m is prime, the formula becomes

$$x^{-1} = x^{m-2}.$$

For example,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

This formula allows us to efficiently calculate modular inverses using the modular exponentiation algorithm. The formula can be derived using Euler's theorem. First, the modular inverse should satisfy the following equation:

$$xx^{-1} \bmod m = 1.$$

On the other hand, according to Euler's theorem,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

so the numbers x^{-1} and $x^{\varphi(m)-1}$ are equal.

Computer arithmetic

In programming, unsigned integers are represented modulo 2^k , where k is the number of bits of the data type. A usual consequence of this is that a number wraps around if it becomes too large.

For example, in C++, numbers of type `unsigned int` are represented modulo 2^{32} . The following code declares an `unsigned int` variable whose value is 123456789. After this, the value will be multiplied by itself, and the result is $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3 Solving equations

Diophantine equations

A **Diophantine equation** is an equation of the form

$$ax + by = c,$$

where a , b and c are constants and the values of x and y should be found. Each number in the equation has to be an integer. For example, one solution for the equation $5x + 2y = 11$ is $x = 3$ and $y = -2$.

We can efficiently solve a Diophantine equation by using Euclid's algorithm. It turns out that we can extend Euclid's algorithm so that it will find numbers x and y that satisfy the following equation:

$$ax + by = \gcd(a, b)$$

A Diophantine equation can be solved if c is divisible by $\gcd(a, b)$, and otherwise it cannot be solved.

As an example, let us find numbers x and y that satisfy the following equation:

$$39x + 15y = 12$$

The equation can be solved, because $\gcd(39, 15) = 3$ and $3 \mid 12$. When Euclid's algorithm calculates the greatest common divisor of 39 and 15, it produces the following sequence of function calls:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

This corresponds to the following equations:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Using these equations, we can derive

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

and by multiplying this by 4, the result is

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

so a solution to the equation is $x = 8$ and $y = -20$.

A solution to a Diophantine equation is not unique, because we can form an infinite number of solutions if we know one solution. If a pair (x, y) is a solution, then also all pairs

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

are solutions, where k is any integer.

Chinese remainder theorem

The **Chinese remainder theorem** solves a group of equations of the form

$$\begin{aligned}x &= a_1 \bmod m_1 \\x &= a_2 \bmod m_2 \\&\dots \\x &= a_n \bmod m_n\end{aligned}$$

where all pairs of m_1, m_2, \dots, m_n are coprime.

Let x_m^{-1} be the inverse of x modulo m , and

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Using this notation, a solution to the equations is

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

In this solution, for each $k = 1, 2, \dots, n$,

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

because

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Since all other terms in the sum are divisible by m_k , they have no effect on the remainder, and $x \bmod m_k = a_k$.

For example, a solution for

$$\begin{aligned}x &= 3 \bmod 5 \\x &= 4 \bmod 7 \\x &= 2 \bmod 3\end{aligned}$$

is

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Once we have found a solution x , we can create an infinite number of other solutions, because all numbers of the form

$$x + m_1 m_2 \cdots m_n$$

are solutions.

21.4 Other results

Lagrange's theorem

Lagrange's theorem states that every positive integer can be represented as a sum of four squares, i.e., $a^2 + b^2 + c^2 + d^2$. For example, the number 123 can be represented as the sum $8^2 + 5^2 + 5^2 + 3^2$.

Zeckendorf's theorem

Zeckendorf's theorem states that every positive integer has a unique representation as a sum of Fibonacci numbers such that no two numbers are equal or consecutive Fibonacci numbers. For example, the number 74 can be represented as the sum $55 + 13 + 5 + 1$.

Pythagorean triples

A **Pythagorean triple** is a triple (a, b, c) that satisfies the Pythagorean theorem $a^2 + b^2 = c^2$, which means that there is a right triangle with side lengths a , b and c . For example, $(3, 4, 5)$ is a Pythagorean triple.

If (a, b, c) is a Pythagorean triple, all triples of the form (ka, kb, kc) are also Pythagorean triples where $k > 1$. A Pythagorean triple is *primitive* if a , b and c are coprime, and all Pythagorean triples can be constructed from primitive triples using a multiplier k .

Euclid's formula can be used to produce all primitive Pythagorean triples. Each such triple is of the form

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

where $0 < m < n$, n and m are coprime and at least one of n and m is even. For example, when $m = 1$ and $n = 2$, the formula produces the smallest Pythagorean triple

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

Wilson's theorem

Wilson's theorem states that a number n is prime exactly when

$$(n - 1)! \bmod n = n - 1.$$

For example, the number 11 is prime, because

$$10! \bmod 11 = 10,$$

and the number 12 is not prime, because

$$11! \bmod 12 = 0 \neq 11.$$

Hence, Wilson's theorem can be used to find out whether a number is prime. However, in practice, the theorem cannot be applied to large values of n , because it is difficult to calculate values of $(n - 1)!$ when n is large.

Chapter 22

Combinatorics

Combinatorics studies methods for counting combinations of objects. Usually, the goal is to find a way to count the combinations efficiently without generating each combination separately.

As an example, consider the problem of counting the number of ways to represent an integer n as a sum of positive integers. For example, there are 8 representations for 4:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- 4

A combinatorial problem can often be solved using a recursive function. In this problem, we can define a function $f(n)$ that gives the number of representations for n . For example, $f(4) = 8$ according to the above example. The values of the function can be recursively calculated as follows:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \cdots + f(n-1) & n > 0 \end{cases}$$

The base case is $f(0) = 1$, because the empty sum represents the number 0. Then, if $n > 0$, we consider all ways to choose the first number of the sum. If the first number is k , there are $f(n - k)$ representations for the remaining part of the sum. Thus, we calculate the sum of all values of the form $f(n - k)$ where $k < n$.

The first values for the function are:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

Sometimes, a recursive formula can be replaced with a closed-form formula. In this problem,

$$f(n) = 2^{n-1},$$

which is based on the fact that there are $n - 1$ possible positions for $+$ -signs in the sum and we can choose any subset of them.

22.1 Binomial coefficients

The **binomial coefficient** $\binom{n}{k}$ equals the number of ways we can choose a subset of k elements from a set of n elements. For example, $\binom{5}{3} = 10$, because the set $\{1, 2, 3, 4, 5\}$ has 10 subsets of 3 elements:

$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$

Formula 1

Binomial coefficients can be recursively calculated as follows:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

The idea is to fix an element x in the set. If x is included in the subset, we have to choose $k - 1$ elements from $n - 1$ elements, and if x is not included in the subset, we have to choose k elements from $n - 1$ elements.

The base cases for the recursion are

$$\binom{n}{0} = \binom{n}{n} = 1,$$

because there is always exactly one way to construct an empty subset and a subset that contains all the elements.

Formula 2

Another way to calculate binomial coefficients is as follows:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

There are $n!$ permutations of n elements. We go through all permutations and always include the first k elements of the permutation in the subset. Since the order of the elements in the subset and outside the subset does not matter, the result is divided by $k!$ and $(n - k)!$

Properties

For binomial coefficients,

$$\binom{n}{k} = \binom{n}{n-k},$$

because we actually divide a set of n elements into two subsets: the first contains k elements and the second contains $n - k$ elements.

The sum of binomial coefficients is

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

The reason for the name "binomial coefficient" can be seen when the binomial $(a + b)$ is raised to the n th power:

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

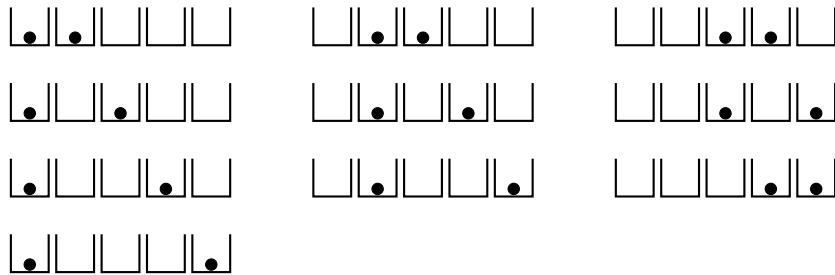
Binomial coefficients also appear in **Pascal's triangle** where each value equals the sum of two above values:

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 1 & & 1 & & \\ & & & 1 & & 2 & & 1 & \\ & & 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & & \dots & & \dots & & \dots & & \dots \end{array}$$

Boxes and balls

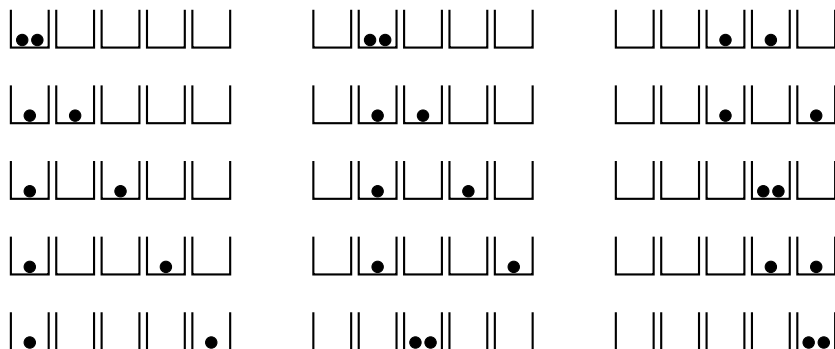
"Boxes and balls" is a useful model, where we count the ways to place k balls in n boxes. Let us consider three scenarios:

Scenario 1: Each box can contain at most one ball. For example, when $n = 5$ and $k = 2$, there are 10 solutions:



In this scenario, the answer is directly the binomial coefficient $\binom{n}{k}$.

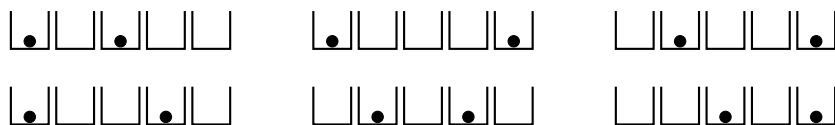
Scenario 2: A box can contain multiple balls. For example, when $n = 5$ and $k = 2$, there are 15 solutions:



The process of placing the balls in the boxes can be represented as a string that consists of symbols "o" and "→". Initially, assume that we are standing at the leftmost box. The symbol "o" means that we place a ball in the current box, and the symbol "→" means that we move to the next box to the right.

Using this notation, each solution is a string that contains k times the symbol "o" and $n - 1$ times the symbol "→". For example, the upper-right solution in the above picture corresponds to the string "→ → o → o →". Thus, the number of solutions is $\binom{k+n-1}{k}$.

Scenario 3: Each box may contain at most one ball, and in addition, no two adjacent boxes may both contain a ball. For example, when $n = 5$ and $k = 2$, there are 6 solutions:



In this scenario, we can assume that k balls are initially placed in boxes and there is an empty box between each two adjacent boxes. The remaining task is to choose the positions for the remaining empty boxes. There are $n - 2k + 1$ such boxes and $k + 1$ positions for them. Thus, using the formula of scenario 2, the number of solutions is $\binom{n-k+1}{n-2k+1}$.

Multinomial coefficients

The **multinomial coefficient**

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

equals the number of ways we can divide n elements into subsets of sizes k_1, k_2, \dots, k_m , where $k_1 + k_2 + \cdots + k_m = n$. Multinomial coefficients can be seen as a generalization of binomial coefficients; if $m = 2$, the above formula corresponds to the binomial coefficient formula.

22.2 Catalan numbers

The **Catalan number** C_n equals the number of valid parenthesis expressions that consist of n left parentheses and n right parentheses.

For example, $C_3 = 5$, because we can construct the following parenthesis expressions using three left and right parentheses:

- $()()()$
- $((()))$
- $()(())$
- $((()))$
- $((()))$

Parenthesis expressions

What is exactly a *valid parenthesis expression*? The following rules precisely define all valid parenthesis expressions:

- An empty parenthesis expression is valid.
- If an expression A is valid, then also the expression (A) is valid.
- If expressions A and B are valid, then also the expression AB is valid.

Another way to characterize valid parenthesis expressions is that if we choose any prefix of such an expression, it has to contain at least as many left parentheses as right parentheses. In addition, the complete expression has to contain an equal number of left and right parentheses.

Formula 1

Catalan numbers can be calculated using the formula

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

The sum goes through the ways to divide the expression into two parts such that both parts are valid expressions and the first part is as short as possible but not empty. For any i , the first part contains $i + 1$ pairs of parentheses and the number of expressions is the product of the following values:

- C_i : the number of ways to construct an expression using the parentheses of the first part, not counting the outermost parentheses
- C_{n-i-1} : the number of ways to construct an expression using the parentheses of the second part

The base case is $C_0 = 1$, because we can construct an empty parenthesis expression using zero pairs of parentheses.

Formula 2

Catalan numbers can also be calculated using binomial coefficients:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

The formula can be explained as follows:

There are a total of $\binom{2n}{n}$ ways to construct a (not necessarily valid) parenthesis expression that contains n left parentheses and n right parentheses. Let us calculate the number of such expressions that are *not* valid.

If a parenthesis expression is not valid, it has to contain a prefix where the number of right parentheses exceeds the number of left parentheses. The

idea is to reverse each parenthesis that belongs to such a prefix. For example, the expression $()()()$ contains a prefix $()()$, and after reversing the prefix, the expression becomes $)((()()$.

The resulting expression consists of $n + 1$ left parentheses and $n - 1$ right parentheses. The number of such expressions is $\binom{2n}{n+1}$, which equals the number of non-valid parenthesis expressions. Thus, the number of valid parenthesis expressions can be calculated using the formula

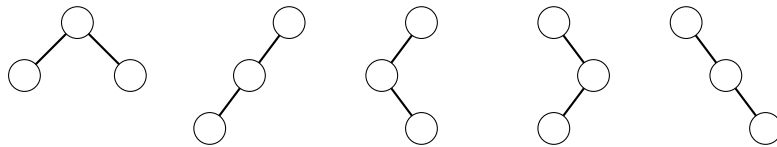
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Counting trees

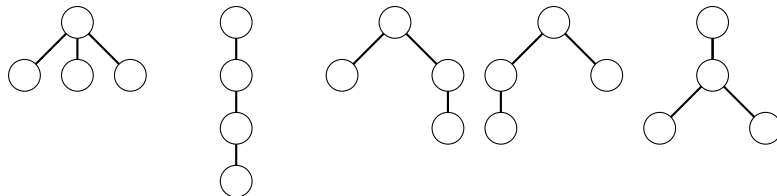
Catalan numbers are also related to trees:

- there are C_n binary trees of n nodes
- there are C_{n-1} rooted trees of n nodes

For example, for $C_3 = 5$, the binary trees are



and the rooted trees are



22.3 Inclusion-exclusion

Inclusion-exclusion is a technique that can be used for counting the size of a union of sets when the sizes of the intersections are known, and vice versa. A simple example of the technique is the formula

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

where A and B are sets and $|X|$ denotes the size of X . The formula can be illustrated as follows:

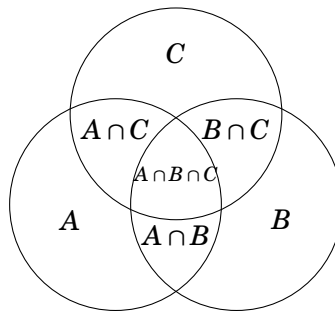


Our goal is to calculate the size of the union $A \cup B$ that corresponds to the area of the region that belongs to at least one circle. The picture shows that we can calculate the area of $A \cup B$ by first summing the areas of A and B and then subtracting the area of $A \cap B$.

The same idea can be applied when the number of sets is larger. When there are three sets, the inclusion-exclusion formula is

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

and the corresponding picture is



In the general case, the size of the union $X_1 \cup X_2 \cup \dots \cup X_n$ can be calculated by going through all possible intersections that contain some of the sets X_1, X_2, \dots, X_n . If the intersection contains an odd number of sets, its size is added to the answer, and otherwise its size is subtracted from the answer.

Note that there are similar formulas for calculating the size of an intersection from the sizes of unions. For example,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

and

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Derangements

As an example, let us count the number of **derangements** of elements $\{1, 2, \dots, n\}$, i.e., permutations where no element remains in its original place. For example, when $n = 3$, there are two derangements: $(2, 3, 1)$ and $(3, 1, 2)$.

One approach for solving the problem is to use inclusion-exclusion. Let X_k be the set of permutations that contain the element k at position k . For example, when $n = 3$, the sets are as follows:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Using these sets, the number of derangements equals

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

so it suffices to calculate the size of the union. Using inclusion-exclusion, this reduces to calculating sizes of intersections which can be done efficiently. For example, when $n = 3$, the size of $|X_1 \cup X_2 \cup X_3|$ is

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ = & 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ = & 4, \end{aligned}$$

so the number of solutions is $3! - 4 = 2$.

It turns out that the problem can also be solved without using inclusion-exclusion. Let $f(n)$ denote the number of derangements for $\{1, 2, \dots, n\}$. We can use the following recursive formula:

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

The formula can be derived by considering the possibilities how the element 1 changes in the derangement. There are $n - 1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

Option 1: We also replace the element x with the element 1. After this, the remaining task is to construct a derangement of $n - 2$ elements.

Option 2: We replace the element x with some other element than 1. Now we have to construct a derangement of $n - 1$ element, because we cannot replace the element x with the element 1, and all other elements must be changed.

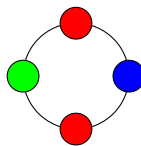
22.4 Burnside's lemma

Burnside's lemma can be used to count the number of combinations so that only one representative is counted for each group of symmetric combinations. Burnside's lemma states that the number of combinations is

$$\sum_{k=1}^n \frac{c(k)}{n},$$

where there are n ways to change the position of a combination, and there are $c(k)$ combinations that remain unchanged when the k th way is applied.

As an example, let us calculate the number of necklaces of n pearls, where each pearl has m possible colors. Two necklaces are symmetric if they are similar after rotating them. For example, the necklace



has the following symmetric necklaces:



There are n ways to change the position of a necklace, because we can rotate it $0, 1, \dots, n-1$ steps clockwise. If the number of steps is 0, all m^n necklaces remain the same, and if the number of steps is 1, only the m necklaces where each pearl has the same color remain the same.

More generally, when the number of steps is k , a total of

$$m^{\gcd(k,n)}$$

necklaces remain the same, where $\gcd(k, n)$ is the greatest common divisor of k and n . The reason for this is that blocks of pearls of size $\gcd(k, n)$ will replace each other. Thus, according to Burnside's lemma, the number of necklaces is

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

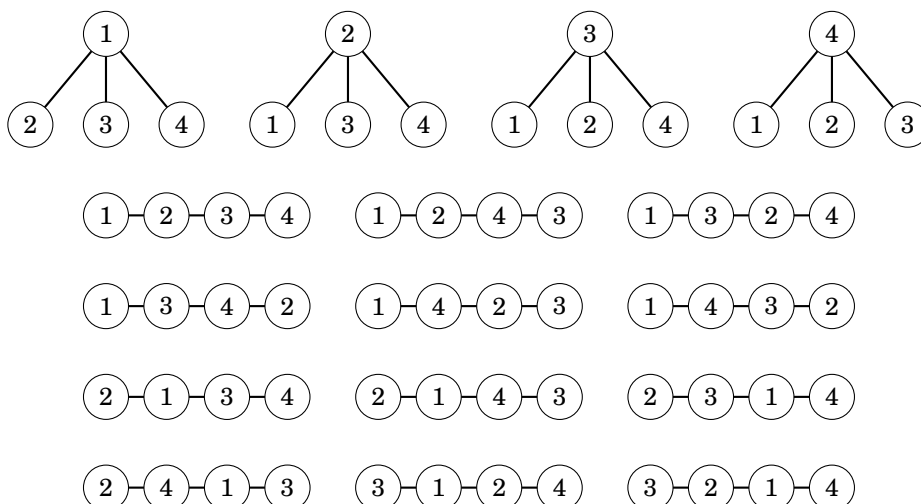
For example, the number of necklaces of length 4 with 3 colors is

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Cayley's formula

Cayley's formula states that there are n^{n-2} labeled trees that contain n nodes. The nodes are labeled $1, 2, \dots, n$, and two trees are different if either their structure or labeling is different.

For example, when $n = 4$, the number of labeled trees is $4^{4-2} = 16$:

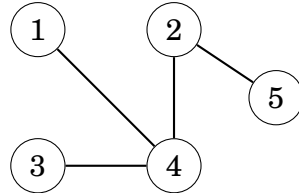


Next we will see how Cayley's formula can be derived using Prüfer codes.

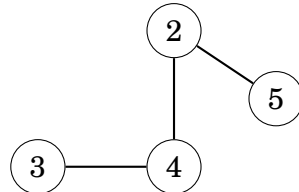
Prüfer code

A **Prüfer code** is a sequence of $n - 2$ numbers that describes a labeled tree. The code is constructed by following a process that removes $n - 2$ leaves from the tree. At each step, the leaf with the smallest label is removed, and the label of its only neighbor is added to the code.

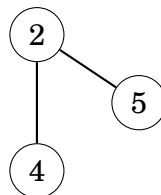
For example, let us calculate the Prüfer code of the following graph:



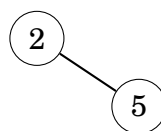
First we remove node 1 and add node 4 to the code:



Then we remove node 3 and add node 4 to the code:



Finally we remove node 4 and add node 2 to the code:



Thus, the Prüfer code of the graph is $[4, 4, 2]$.

We can construct a Prüfer code for any tree, and more importantly, the original tree can be reconstructed from a Prüfer code. Hence, the number of labeled trees of n nodes equals n^{n-2} , the number of Prüfer codes of size n .

Chapter 23

Matrices

A **matrix** is a mathematical concept that corresponds to a two-dimensional array in programming. For example,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

is a matrix of size 3×4 , i.e., it has 3 rows and 4 columns. The notation $[i, j]$ refers to the element in row i and column j in a matrix. For example, in the above matrix, $A[2, 3] = 8$ and $A[3, 1] = 9$.

A special case of a matrix is a **vector** that is a one-dimensional matrix of size $n \times 1$. For example,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

is a vector that contains three elements.

The **transpose** A^T of a matrix A is obtained when the rows and columns of A are swapped, i.e., $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

A matrix is a **square matrix** if it has the same number of rows and columns. For example, the following matrix is a square matrix:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Operations

The sum $A + B$ of matrices A and B is defined if the matrices are of the same size. The result is a matrix where each element is the sum of the corresponding elements in A and B .

For example,

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Multiplying a matrix A by a value x means that each element of A is multiplied by x . For example,

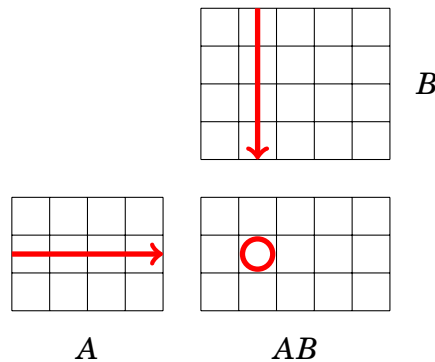
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Matrix multiplication

The product AB of matrices A and B is defined if A is of size $a \times n$ and B is of size $n \times b$, i.e., the width of A equals the height of B . The result is a matrix of size $a \times b$ whose elements are calculated using the formula

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j].$$

The idea is that each element of AB is a sum of products of elements of A and B according to the following picture:



For example,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Matrix multiplication is associative, so $A(BC) = (AB)C$ holds, but it is not commutative, so $AB = BA$ does not usually hold.

An **identity matrix** is a square matrix where each element on the diagonal is 1 and all other elements are 0. For example, the following matrix is the 3×3 identity matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying a matrix by an identity matrix does not change it. For example,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Using a straightforward algorithm, we can calculate the product of two $n \times n$ matrices in $O(n^3)$ time. There are also more efficient algorithms for matrix multiplication¹, but they are mostly of theoretical interest and such algorithms are not necessary in competitive programming.

Matrix power

The power A^k of a matrix A is defined if A is a square matrix. The definition is based on matrix multiplication:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

In addition, A^0 is an identity matrix. For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The matrix A^k can be efficiently calculated in $O(n^3 \log k)$ time using the algorithm in Chapter 21.2. For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

Determinant

The **determinant** $\det(A)$ of a matrix A is defined if A is a square matrix. If A is of size 1×1 , then $\det(A) = A[1, 1]$. The determinant of a larger matrix is calculated recursively using the formula

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

where $C[i, j]$ is the **cofactor** of A at $[i, j]$. The cofactor is calculated using the formula

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

¹The first such algorithm was Strassen's algorithm, published in 1969 [63], whose time complexity is $O(n^{2.80735})$; the best current algorithm [27] works in $O(n^{2.37286})$ time.

where $M[i, j]$ is obtained by removing row i and column j from A . Due to the coefficient $(-1)^{i+j}$ in the cofactor, every other determinant is positive and negative. For example,

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

and

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

The determinant of A tells us whether there is an **inverse matrix** A^{-1} such that $A \cdot A^{-1} = I$, where I is an identity matrix. It turns out that A^{-1} exists exactly when $\det(A) \neq 0$, and it can be calculated using the formula

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

For example,

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 Linear recurrences

A **linear recurrence** is a function $f(n)$ whose initial values are $f(0), f(1), \dots, f(k-1)$ and larger values are calculated recursively using the formula

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

where c_1, c_2, \dots, c_k are constant coefficients.

Dynamic programming can be used to calculate any value of $f(n)$ in $O(kn)$ time by calculating all values of $f(0), f(1), \dots, f(n)$ one after another. However, if k is small, it is possible to calculate $f(n)$ much more efficiently in $O(k^3 \log n)$ time using matrix operations.

Fibonacci numbers

A simple example of a linear recurrence is the following function that defines the Fibonacci numbers:

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

In this case, $k = 2$ and $c_1 = c_2 = 1$.

To efficiently calculate Fibonacci numbers, we represent the Fibonacci formula as a square matrix X of size 2×2 , for which the following holds:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Thus, values $f(i)$ and $f(i+1)$ are given as "input" for X , and X calculates values $f(i+1)$ and $f(i+2)$ from them. It turns out that such a matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Thus, we can calculate $f(n)$ using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The value of X^n can be calculated in $O(\log n)$ time, so the value of $f(n)$ can also be calculated in $O(\log n)$ time.

General case

Let us now consider the general case where $f(n)$ is any linear recurrence. Again, our goal is to construct a matrix X for which

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Such a matrix is

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

In the first $k-1$ rows, each element is 0 except that one element is 1. These rows replace $f(i)$ with $f(i+1)$, $f(i+1)$ with $f(i+2)$, and so on. The last row contains the coefficients of the recurrence to calculate the new value $f(i+k)$.

Now, $f(n)$ can be calculated in $O(k^3 \log n)$ time using the formula

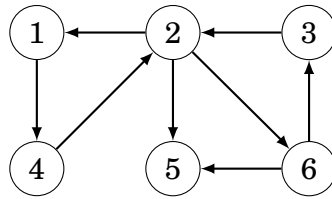
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3 Graphs and matrices

Counting paths

The powers of an adjacency matrix of a graph have an interesting property. When V is an adjacency matrix of an unweighted graph, the matrix V^n contains the numbers of paths of n edges between the nodes in the graph.

For example, for the graph



the adjacency matrix is

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Now, for example, the matrix

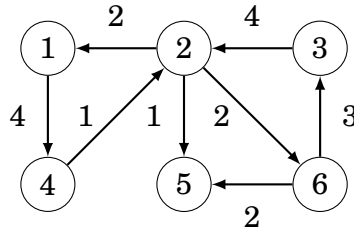
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

contains the numbers of paths of 4 edges between the nodes. For example, $V^4[2,5] = 2$, because there are two paths of 4 edges from node 2 to node 5: $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ and $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Shortest paths

Using a similar idea in a weighted graph, we can calculate for each pair of nodes the minimum length of a path between them that contains exactly n edges. To calculate this, we have to define matrix multiplication in a new way, so that we do not calculate the numbers of paths but minimize the lengths of paths.

As an example, consider the following graph:



Let us construct an adjacency matrix where ∞ means that an edge does not exist, and other values correspond to edge weights. The matrix is

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Instead of the formula

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

we now use the formula

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j]$$

for matrix multiplication, so we calculate a minimum instead of a sum, and a sum of elements instead of a product. After this modification, matrix powers correspond to shortest paths in the graph.

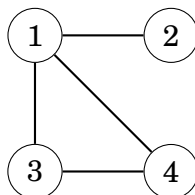
For example, as

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

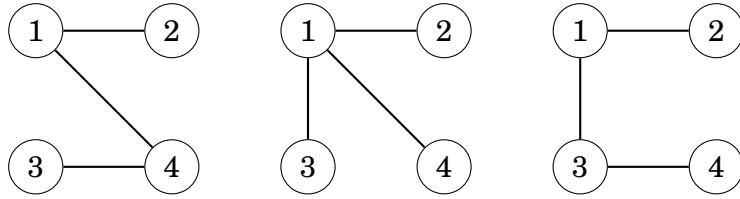
we can conclude that the minimum length of a path of 4 edges from node 2 to node 5 is 8. Such a path is $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

Kirchhoff's theorem

Kirchhoff's theorem provides a way to calculate the number of spanning trees of a graph as a determinant of a special matrix. For example, the graph



has three spanning trees:



To calculate the number of spanning trees, we construct a **Laplacian matrix** L , where $L[i, i]$ is the degree of node i and $L[i, j] = -1$ if there is an edge between nodes i and j , and otherwise $L[i, j] = 0$. The Laplacean matrix for the above graph is as follows:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

It can be shown that the number of spanning trees equals the determinant of a matrix that is obtained when we remove any row and any column from L . For example, if we remove the first row and column, the result is

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

The determinant is always the same, regardless of which row and column we remove from L .

Note that Cayley's formula in Chapter 22.5 is a special case of Kirchhoff's theorem, because in a complete graph of n nodes

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

Chapter 24

Probability

A **probability** is a real number between 0 and 1 that indicates how probable an event is. If an event is certain to happen, its probability is 1, and if an event is impossible, its probability is 0. The probability of an event is denoted $P(\dots)$ where the three dots describe the event.

For example, when throwing a dice, the outcome is an integer between 1 and 6, and the probability of each outcome is $1/6$. For example, we can calculate the following probabilities:

- $P(\text{"the outcome is 4"}) = 1/6$
- $P(\text{"the outcome is not 6"}) = 5/6$
- $P(\text{"the outcome is even"}) = 1/2$

24.1 Calculation

To calculate the probability of an event, we can either use combinatorics or simulate the process that generates the event. As an example, let us calculate the probability of drawing three cards with the same value from a shuffled deck of cards (for example, $\spadesuit 8$, $\clubsuit 8$ and $\diamondsuit 8$).

Method 1

We can calculate the probability using the formula

$$\frac{\text{number of desired outcomes}}{\text{total number of outcomes}}.$$

In this problem, the desired outcomes are those in which the value of each card is the same. There are $13 \binom{4}{3}$ such outcomes, because there are 13 possibilities for the value of the cards and $\binom{4}{3}$ ways to choose 3 suits from 4 possible suits.

There are a total of $\binom{52}{3}$ outcomes, because we choose 3 cards from 52 cards. Thus, the probability of the event is

$$\frac{13 \binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Method 2

Another way to calculate the probability is to simulate the process that generates the event. In this example, we draw three cards, so the process consists of three steps. We require that each step of the process is successful.

Drawing the first card certainly succeeds, because there are no restrictions. The second step succeeds with probability $3/51$, because there are 51 cards left and 3 of them have the same value as the first card. In a similar way, the third step succeeds with probability $2/50$.

The probability that the entire process succeeds is

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 Events

An event in probability theory can be represented as a set

$$A \subset X,$$

where X contains all possible outcomes and A is a subset of outcomes. For example, when drawing a dice, the outcomes are

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Now, for example, the event "the outcome is even" corresponds to the set

$$A = \{2, 4, 6\}.$$

Each outcome x is assigned a probability $p(x)$. Then, the probability $P(A)$ of an event A can be calculated as a sum of probabilities of outcomes using the formula

$$P(A) = \sum_{x \in A} p(x).$$

For example, when throwing a dice, $p(x) = 1/6$ for each outcome x , so the probability of the event "the outcome is even" is

$$p(2) + p(4) + p(6) = 1/2.$$

The total probability of the outcomes in X must be 1, i.e., $P(X) = 1$.

Since the events in probability theory are sets, we can manipulate them using standard set operations:

- The **complement** \bar{A} means "A does not happen". For example, when throwing a dice, the complement of $A = \{2, 4, 6\}$ is $\bar{A} = \{1, 3, 5\}$.
- The **union** $A \cup B$ means "A or B happen". For example, the union of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cup B = \{2, 4, 5, 6\}$.
- The **intersection** $A \cap B$ means "A and B happen". For example, the intersection of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cap B = \{5\}$.

Complement

The probability of the complement \bar{A} is calculated using the formula

$$P(\bar{A}) = 1 - P(A).$$

Sometimes, we can solve a problem easily using complements by solving the opposite problem. For example, the probability of getting at least one six when throwing a dice ten times is

$$1 - (5/6)^{10}.$$

Here $5/6$ is the probability that the outcome of a single throw is not six, and $(5/6)^{10}$ is the probability that none of the ten throws is a six. The complement of this is the answer to the problem.

Union

The probability of the union $A \cup B$ is calculated using the formula

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

For example, when throwing a dice, the union of the events

$$A = \text{"the outcome is even"}$$

and

$$B = \text{"the outcome is less than 4"}$$

is

$$A \cup B = \text{"the outcome is even or less than 4"},$$

and its probability is

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

If the events A and B are **disjoint**, i.e., $A \cap B$ is empty, the probability of the event $A \cup B$ is simply

$$P(A \cup B) = P(A) + P(B).$$

Conditional probability

The **conditional probability**

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

is the probability of A assuming that B happens. Hence, when calculating the probability of A , we only consider the outcomes that also belong to B .

Using the previous sets,

$$P(A|B) = 1/3,$$

because the outcomes of B are $\{1, 2, 3\}$, and one of them is even. This is the probability of an even outcome if we know that the outcome is between $1 \dots 3$.

Intersection

Using conditional probability, the probability of the intersection $A \cap B$ can be calculated using the formula

$$P(A \cap B) = P(A)P(B|A).$$

Events A and B are **independent** if

$$P(A|B) = P(A) \quad \text{and} \quad P(B|A) = P(B),$$

which means that the fact that B happens does not change the probability of A , and vice versa. In this case, the probability of the intersection is

$$P(A \cap B) = P(A)P(B).$$

For example, when drawing a card from a deck, the events

$$A = \text{"the suit is clubs"}$$

and

$$B = \text{"the value is four"}$$

are independent. Hence the event

$$A \cap B = \text{"the card is the four of clubs"}$$

happens with probability

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3 Random variables

A **random variable** is a value that is generated by a random process. For example, when throwing two dice, a possible random variable is

$$X = \text{"the sum of the outcomes"}.$$

For example, if the outcomes are $[4, 6]$ (meaning that we first throw a four and then a six), then the value of X is 10.

We denote $P(X = x)$ the probability that the value of a random variable X is x . For example, when throwing two dice, $P(X = 10) = 3/36$, because the total number of outcomes is 36 and there are three possible ways to obtain the sum 10: $[4, 6]$, $[5, 5]$ and $[6, 4]$.

Expected value

The **expected value** $E[X]$ indicates the average value of a random variable X . The expected value can be calculated as the sum

$$\sum_x P(X = x)x,$$

where x goes through all possible values of X .

For example, when throwing a dice, the expected outcome is

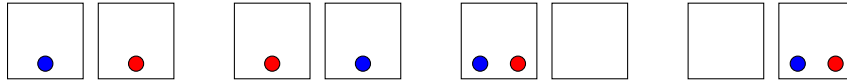
$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

A useful property of expected values is **linearity**. It means that the sum $E[X_1 + X_2 + \dots + X_n]$ always equals the sum $E[X_1] + E[X_2] + \dots + E[X_n]$. This formula holds even if random variables depend on each other.

For example, when throwing two dice, the expected sum is

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Let us now consider a problem where n balls are randomly placed in n boxes, and our task is to calculate the expected number of empty boxes. Each ball has an equal probability to be placed in any of the boxes. For example, if $n = 2$, the possibilities are as follows:



In this case, the expected number of empty boxes is

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

In the general case, the probability that a single box is empty is

$$\left(\frac{n-1}{n}\right)^n,$$

because no ball should be placed in it. Hence, using linearity, the expected number of empty boxes is

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Distributions

The **distribution** of a random variable X shows the probability of each value that X may have. The distribution consists of values $P(X = x)$. For example, when throwing two dice, the distribution for their sum is:

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

In a **uniform distribution**, the random variable X has n possible values $a, a+1, \dots, b$ and the probability of each value is $1/n$. For example, when throwing a dice, $a = 1$, $b = 6$ and $P(X = x) = 1/6$ for each value x .

The expected value of X in a uniform distribution is

$$E[X] = \frac{a+b}{2}.$$

In a **binomial distribution**, n attempts are made and the probability that a single attempt succeeds is p . The random variable X counts the number of successful attempts, and the probability of a value x is

$$P(X = x) = p^x(1-p)^{n-x} \binom{n}{x},$$

where p^x and $(1-p)^{n-x}$ correspond to successful and unsuccessful attempts, and $\binom{n}{x}$ is the number of ways we can choose the order of the attempts.

For example, when throwing a dice ten times, the probability of throwing a six exactly three times is $(1/6)^3(5/6)^7 \binom{10}{3}$.

The expected value of X in a binomial distribution is

$$E[X] = pn.$$

In a **geometric distribution**, the probability that an attempt succeeds is p , and we continue until the first success happens. The random variable X counts the number of attempts needed, and the probability of a value x is

$$P(X = x) = (1-p)^{x-1}p,$$

where $(1-p)^{x-1}$ corresponds to the unsuccessful attempts and p corresponds to the first successful attempt.

For example, if we throw a dice until we throw a six, the probability that the number of throws is exactly 4 is $(5/6)^3 1/6$.

The expected value of X in a geometric distribution is

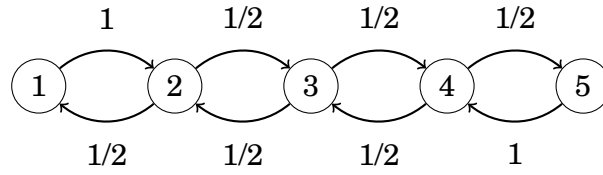
$$E[X] = \frac{1}{p}.$$

24.4 Markov chains

A **Markov chain** is a random process that consists of states and transitions between them. For each state, we know the probabilities for moving to other states. A Markov chain can be represented as a graph whose nodes are states and edges are transitions.

As an example, consider a problem where we are in floor 1 in an n floor building. At each step, we randomly walk either one floor up or one floor down, except that we always walk one floor up from floor 1 and one floor down from floor n . What is the probability of being in floor m after k steps?

In this problem, each floor of the building corresponds to a state in a Markov chain. For example, if $n = 5$, the graph is as follows:



The probability distribution of a Markov chain is a vector $[p_1, p_2, \dots, p_n]$, where p_k is the probability that the current state is k . The formula $p_1 + p_2 + \dots + p_n = 1$ always holds.

In the above scenario, the initial distribution is $[1, 0, 0, 0, 0]$, because we always begin in floor 1. The next distribution is $[0, 1, 0, 0, 0]$, because we can only move from floor 1 to floor 2. After this, we can either move one floor up or one floor down, so the next distribution is $[1/2, 0, 1/2, 0, 0]$, and so on.

An efficient way to simulate the walk in a Markov chain is to use dynamic programming. The idea is to maintain the probability distribution, and at each step go through all possibilities how we can move. Using this method, we can simulate a walk of m steps in $O(n^2m)$ time.

The transitions of a Markov chain can also be represented as a matrix that updates the probability distribution. In the above scenario, the matrix is

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

When we multiply a probability distribution by this matrix, we get the new distribution after moving one step. For example, we can move from the distribution $[1, 0, 0, 0, 0]$ to the distribution $[0, 1, 0, 0, 0]$ as follows:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

By calculating matrix powers efficiently, we can calculate the distribution after m steps in $O(n^3 \log m)$ time.

24.5 Randomized algorithms

Sometimes we can use randomness for solving a problem, even if the problem is not related to probabilities. A **randomized algorithm** is an algorithm that is based on randomness.

A **Monte Carlo algorithm** is a randomized algorithm that may sometimes give a wrong answer. For such an algorithm to be useful, the probability of a wrong answer should be small.

A **Las Vegas algorithm** is a randomized algorithm that always gives the correct answer, but its running time varies randomly. The goal is to design an algorithm that is efficient with high probability.

Next we will go through three example problems that can be solved using randomness.

Order statistics

The k th **order statistic** of an array is the element at position k after sorting the array in increasing order. It is easy to calculate any order statistic in $O(n \log n)$ time by first sorting the array, but is it really needed to sort the entire array just to find one element?

It turns out that we can find order statistics using a randomized algorithm without sorting the array. The algorithm, called **quickselect**¹, is a Las Vegas algorithm: its running time is usually $O(n)$ but $O(n^2)$ in the worst case.

The algorithm chooses a random element x of the array, and moves elements smaller than x to the left part of the array, and all other elements to the right part of the array. This takes $O(n)$ time when there are n elements. Assume that the left part contains a elements and the right part contains b elements. If $a = k$, element x is the k th order statistic. Otherwise, if $a > k$, we recursively find the k th order statistic for the left part, and if $a < k$, we recursively find the r th order statistic for the right part where $r = k - a$. The search continues in a similar way, until the element has been found.

When each element x is randomly chosen, the size of the array about halves at each step, so the time complexity for finding the k th order statistic is about

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

The worst case of the algorithm requires still $O(n^2)$ time, because it is possible that x is always chosen in such a way that it is one of the smallest or largest elements in the array and $O(n)$ steps are needed. However, the probability for this is so small that this never happens in practice.

Verifying matrix multiplication

Our next problem is to *verify* if $AB = C$ holds when A , B and C are matrices of size $n \times n$. Of course, we can solve the problem by calculating the product AB again (in $O(n^3)$ time using the basic algorithm), but one could hope that verifying the answer would be easier than to calculate it from scratch.

It turns out that we can solve the problem using a Monte Carlo algorithm² whose time complexity is only $O(n^2)$. The idea is simple: we choose a random vector X of n elements, and calculate the matrices ABX and CX . If $ABX = CX$, we report that $AB = C$, and otherwise we report that $AB \neq C$.

¹In 1961, C. A. R. Hoare published two algorithms that are efficient on average: **quicksort** [36] for sorting arrays and **quickselect** [37] for finding order statistics.

²R. M. Freivalds published this algorithm in 1977 [26], and it is sometimes called **Freivalds' algorithm**.

The time complexity of the algorithm is $O(n^2)$, because we can calculate the matrices ABX and CX in $O(n^2)$ time. We can calculate the matrix ABX efficiently by using the representation $A(BX)$, so only two multiplications of $n \times n$ and $n \times 1$ size matrices are needed.

The drawback of the algorithm is that there is a small chance that the algorithm makes a mistake when it reports that $AB = C$. For example,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

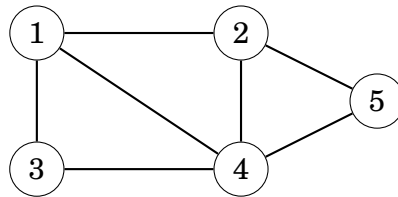
but

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

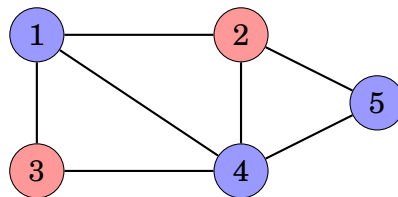
However, in practice, the probability that the algorithm makes a mistake is small, and we can decrease the probability by verifying the result using multiple random vectors X before reporting that $AB = C$.

Graph coloring

Given a graph that contains n nodes and m edges, our task is to find a way to color the nodes of the graph using two colors so that for at least $m/2$ edges, the endpoints have different colors. For example, in the graph



a valid coloring is as follows:



The above graph contains 7 edges, and for 5 of them, the endpoints have different colors, so the coloring is valid.

The problem can be solved using a Las Vegas algorithm that generates random colorings until a valid coloring has been found. In a random coloring, the color of each node is independently chosen so that the probability of both colors is $1/2$.

In a random coloring, the probability that the endpoints of a single edge have different colors is $1/2$. Hence, the expected number of edges whose endpoints have different colors is $m/2$. Since it is expected that a random coloring is valid, we will quickly find a valid coloring in practice.

Chapter 25

Game theory

In this chapter, we will focus on two-player games that do not contain random elements. Our goal is to find a strategy that we can follow to win the game no matter what the opponent does, if such a strategy exists.

It turns out that there is a general strategy for such games, and we can analyze the games using the **nim theory**. First, we will analyze simple games where players remove sticks from heaps, and after this, we will generalize the strategy used in those games to other games.

25.1 Game states

Let us consider a game where there is initially a heap of n sticks. Players A and B move alternately, and player A begins. On each move, the player has to remove 1, 2 or 3 sticks from the heap, and the player who removes the last stick wins the game.

For example, if $n = 10$, the game may proceed as follows:

- Player A removes 2 sticks (8 sticks left).
- Player B removes 3 sticks (5 sticks left).
- Player A removes 1 stick (4 sticks left).
- Player B removes 2 sticks (2 sticks left).
- Player A removes 2 sticks and wins.

This game consists of states $0, 1, 2, \dots, n$, where the number of the state corresponds to the number of sticks left.

Winning and losing states

A **winning state** is a state where the player will win the game if they play optimally, and a **losing state** is a state where the player will lose the game if the opponent plays optimally. It turns out that we can classify all states of a game so that each state is either a winning state or a losing state.

In the above game, state 0 is clearly a losing state, because the player cannot make any moves. States 1, 2 and 3 are winning states, because we can remove 1,

2 or 3 sticks and win the game. State 4, in turn, is a losing state, because any move leads to a state that is a winning state for the opponent.

More generally, if there is a move that leads from the current state to a losing state, the current state is a winning state, and otherwise the current state is a losing state. Using this observation, we can classify all states of a game starting with losing states where there are no possible moves.

The states 0...15 of the above game can be classified as follows (W denotes a winning state and L denotes a losing state):

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

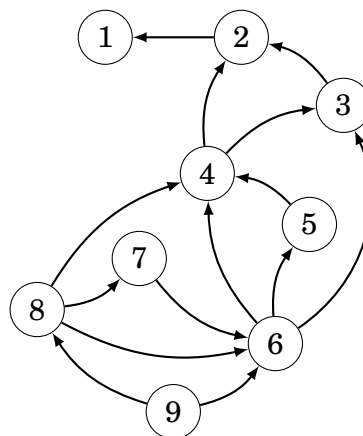
It is easy to analyze this game: a state k is a losing state if k is divisible by 4, and otherwise it is a winning state. An optimal way to play the game is to always choose a move after which the number of sticks in the heap is divisible by 4. Finally, there are no sticks left and the opponent has lost.

Of course, this strategy requires that the number of sticks is *not* divisible by 4 when it is our move. If it is, there is nothing we can do, and the opponent will win the game if they play optimally.

State graph

Let us now consider another stick game, where in each state k , it is allowed to remove any number x of sticks such that x is smaller than k and divides k . For example, in state 8 we may remove 1, 2 or 4 sticks, but in state 7 the only allowed move is to remove 1 stick.

The following picture shows the states 1...9 of the game as a **state graph**, whose nodes are the states and edges are the moves between them:



The final state in this game is always state 1, which is a losing state, because there are no valid moves. The classification of states 1...9 is as follows:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Surprisingly, in this game, all even-numbered states are winning states, and all odd-numbered states are losing states.

25.2 Nim game

The **nim game** is a simple game that has an important role in game theory, because many other games can be played using the same strategy. First, we focus on nim, and then we generalize the strategy to other games.

There are n heaps in nim, and each heap contains some number of sticks. The players move alternately, and on each turn, the player chooses a heap that still contains sticks and removes any number of sticks from it. The winner is the player who removes the last stick.

The states in nim are of the form $[x_1, x_2, \dots, x_n]$, where x_k denotes the number of sticks in heap k . For example, $[10, 12, 5]$ is a game where there are three heaps with 10, 12 and 5 sticks. The state $[0, 0, \dots, 0]$ is a losing state, because it is not possible to remove any sticks, and this is always the final state.

Analysis

It turns out that we can easily classify any nim state by calculating the **nim sum** $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$, where \oplus is the xor operation¹. The states whose nim sum is 0 are losing states, and all other states are winning states. For example, the nim sum of $[10, 12, 5]$ is $10 \oplus 12 \oplus 5 = 3$, so the state is a winning state.

But how is the nim sum related to the nim game? We can explain this by looking at how the nim sum changes when the nim state changes.

Losing states: The final state $[0, 0, \dots, 0]$ is a losing state, and its nim sum is 0, as expected. In other losing states, any move leads to a winning state, because when a single value x_k changes, the nim sum also changes, so the nim sum is different from 0 after the move.

Winning states: We can move to a losing state if there is any heap k for which $x_k \oplus s < x_k$. In this case, we can remove sticks from heap k so that it will contain $x_k \oplus s$ sticks, which will lead to a losing state. There is always such a heap, where x_k has a one bit at the position of the leftmost one bit of s .

As an example, consider the state $[10, 12, 5]$. This state is a winning state, because its nim sum is 3. Thus, there has to be a move which leads to a losing state. Next we will find out such a move.

The nim sum of the state is as follows:

10		1010
12		1100
5		0101
3		0011

In this case, the heap with 10 sticks is the only heap that has a one bit at the position of the leftmost one bit of the nim sum:

10		10 <u>1</u> 0
12		1100
5		0101
3		00 <u>1</u> 1

¹The optimal strategy for nim was published in 1901 by C. L. Bouton [10].

The new size of the heap has to be $10 \oplus 3 = 9$, so we will remove just one stick. After this, the state will be $[9, 12, 5]$, which is a losing state:

9	1001
12	1100
5	0101
0	0000

Misère game

In a **misère game**, the goal of the game is opposite, so the player who removes the last stick loses the game. It turns out that the misère nim game can be optimally played almost like the standard nim game.

The idea is to first play the misère game like the standard game, but change the strategy at the end of the game. The new strategy will be introduced in a situation where each heap would contain at most one stick after the next move.

In the standard game, we should choose a move after which there is an even number of heaps with one stick. However, in the misère game, we choose a move so that there is an odd number of heaps with one stick.

This strategy works because a state where the strategy changes always appears in the game, and this state is a winning state, because it contains exactly one heap that has more than one stick so the nim sum is not 0.

25.3 Sprague–Grundy theorem

The **Sprague–Grundy theorem**² generalizes the strategy used in nim to all games that fulfil the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

The idea is to calculate for each game state a Grundy number that corresponds to the number of sticks in a nim heap. When we know the Grundy numbers of all states, we can play the game like the nim game.

Grundy numbers

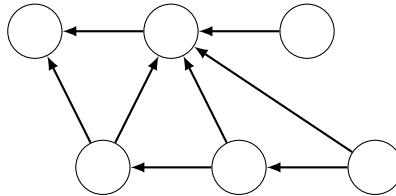
The **Grundy number** of a game state is

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

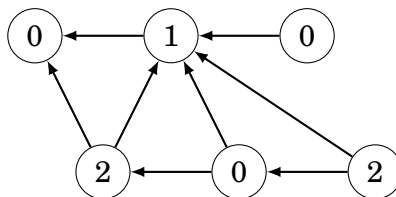
²The theorem was independently discovered by R. Sprague [61] and P. M. Grundy [31].

where g_1, g_2, \dots, g_n are the Grundy numbers of the states to which we can move, and the mex function gives the smallest nonnegative number that is not in the set. For example, $\text{mex}(\{0, 1, 3\}) = 2$. If there are no possible moves in a state, its Grundy number is 0, because $\text{mex}(\emptyset) = 0$.

For example, in the state graph



the Grundy numbers are as follows:



The Grundy number of a losing state is 0, and the Grundy number of a winning state is a positive number.

The Grundy number of a state corresponds to the number of sticks in a nim heap. If the Grundy number is 0, we can only move to states whose Grundy numbers are positive, and if the Grundy number is $x > 0$, we can move to states whose Grundy numbers include all numbers $0, 1, \dots, x - 1$.

As an example, consider a game where the players move a figure in a maze. Each square in the maze is either floor or wall. On each turn, the player has to move the figure some number of steps left or up. The winner of the game is the player who makes the last move.

The following picture shows a possible initial state of the game, where @ denotes the figure and * denotes a square where it can move.



The states of the game are all floor squares of the maze. In the above maze, the Grundy numbers are as follows:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Thus, each state of the maze game corresponds to a heap in the nim game. For example, the Grundy number for the lower-right square is 2, so it is a winning state. We can reach a losing state and win the game by moving either four steps left or two steps up.

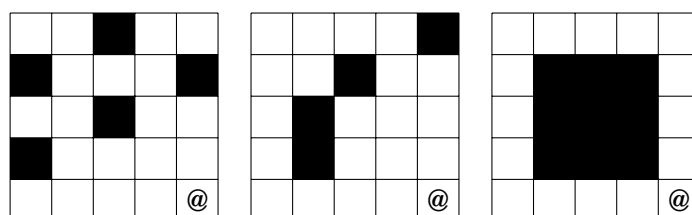
Note that unlike in the original nim game, it may be possible to move to a state whose Grundy number is larger than the Grundy number of the current state. However, the opponent can always choose a move that cancels such a move, so it is not possible to escape from a losing state.

Subgames

Next we will assume that our game consists of subgames, and on each turn, the player first chooses a subgame and then a move in the subgame. The game ends when it is not possible to make any move in any subgame.

In this case, the Grundy number of a game is the nim sum of the Grundy numbers of the subgames. The game can be played like a nim game by calculating all Grundy numbers for subgames and then their nim sum.

As an example, consider a game that consists of three mazes. In this game, on each turn, the player chooses one of the mazes and then moves the figure in the maze. Assume that the initial state of the game is as follows:



The Grundy numbers for the mazes are as follows:

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

In the initial state, the nim sum of the Grundy numbers is $2 \oplus 3 \oplus 3 = 2$, so the first player can win the game. One optimal move is to move two steps up in the first maze, which produces the nim sum $0 \oplus 3 \oplus 3 = 0$.

Grundy's game

Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of the game is

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

where n is the number of possible moves and

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

where move k generates subgames with Grundy numbers $a_{k,1}, a_{k,2}, \dots, a_{k,m}$.

An example of such a game is **Grundy's game**. Initially, there is a single heap that contains n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game.

Let $f(n)$ be the Grundy number of a heap that contains n sticks. The Grundy number can be calculated by going through all ways to divide the heap into two heaps. For example, when $n = 8$, the possibilities are $1 + 7$, $2 + 6$ and $3 + 5$, so

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

In this game, the value of $f(n)$ is based on the values of $f(1), \dots, f(n-1)$. The base cases are $f(1) = f(2) = 0$, because it is not possible to divide the heaps of 1 and 2 sticks. The first Grundy numbers are:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

The Grundy number for $n = 8$ is 2, so it is possible to win the game. The winning move is to create heaps $1 + 7$, because $f(1) \oplus f(7) = 0$.

Chapter 26

String algorithms

This chapter deals with efficient algorithms for string processing. Many string problems can be easily solved in $O(n^2)$ time, but the challenge is to find algorithms that work in $O(n)$ or $O(n \log n)$ time.

For example, a fundamental string processing problem is the **pattern matching** problem: given a string of length n and a pattern of length m , our task is to find the occurrences of the pattern in the string. For example, the pattern ABC occurs two times in the string ABABCBABC.

The pattern matching problem can be easily solved in $O(nm)$ time by a brute force algorithm that tests all positions where the pattern may occur in the string. However, in this chapter, we will see that there are more efficient algorithms that require only $O(n + m)$ time.

26.1 String terminology

Throughout the chapter, we assume that zero-based indexing is used in strings. Thus, a string s of length n consists of characters $s[0], s[1], \dots, s[n-1]$. The set of characters that may appear in strings is called an **alphabet**. For example, the alphabet $\{A, B, \dots, Z\}$ consists of the capital letters of English.

A **substring** is a sequence of consecutive characters in a string. We use the notation $s[a \dots b]$ to refer to a substring of s that begins at position a and ends at position b . A string of length n has $n(n+1)/2$ substrings. For example, the substrings of ABCD are A, B, C, D, AB, BC, CD, ABC, BCD and ABCD.

A **subsequence** is a sequence of (not necessarily consecutive) characters in a string in their original order. A string of length n has $2^n - 1$ subsequences. For example, the subsequences of ABCD are A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD and ABCD.

A **prefix** is a substring that starts at the beginning of a string, and a **suffix** is a substring that ends at the end of a string. For example, the prefixes of ABCD are A, AB, ABC and ABCD, and the suffixes of ABCD are D, CD, BCD and ABCD.

A **rotation** can be generated by moving the characters of a string one by one from the beginning to the end (or vice versa). For example, the rotations of ABCD are ABCD, BCDA, CDAB and DABC.

A **period** is a prefix of a string such that the string can be constructed by repeating the period. The last repetition may be partial and contain only a prefix of the period. For example, the shortest period of ABCABCA is ABC.

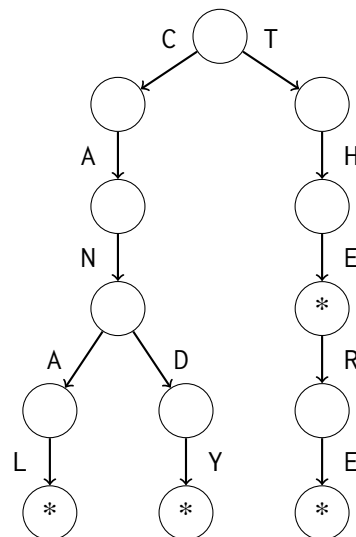
A **border** is a string that is both a prefix and a suffix of a string. For example, the borders of ABACABA are A, ABA and ABACABA.

Strings are compared using the **lexicographical order** (which corresponds to the alphabetical order). It means that $x < y$ if either $x \neq y$ and x is a prefix of y , or there is a position k such that $x[i] = y[i]$ when $i < k$ and $x[k] < y[k]$.

26.2 Trie structure

A **trie** is a rooted tree that maintains a set of strings. Each string in the set is stored as a chain of characters that starts at the root. If two strings have a common prefix, they also have a common chain in the tree.

For example, consider the following trie:



This trie corresponds to the set {CANAL, CANDY, THE, THERE}. The character * in a node means that a string in the set ends at the node. Such a character is needed, because a string may be a prefix of another string. For example, in the above trie, THE is a prefix of THERE.

We can check in $O(n)$ time whether a trie contains a string of length n , because we can follow the chain that starts at the root node. We can also add a string of length n to the trie in $O(n)$ time by first following the chain and then adding new nodes to the trie if necessary.

Using a trie, we can find the longest prefix of a given string such that the prefix belongs to the set. Moreover, by storing additional information in each node, we can calculate the number of strings that belong to the set and have a given string as a prefix.

A trie can be stored in an array

```
int trie[N][A];
```

where N is the maximum number of nodes (the maximum total length of the strings in the set) and A is the size of the alphabet. The nodes of a trie are numbered $0, 1, 2, \dots$ so that the number of the root is 0, and $\text{trie}[s][c]$ is the next node in the chain when we move from node s using character c .

26.3 String hashing

String hashing is a technique that allows us to efficiently check whether two strings are equal¹. The idea in string hashing is to compare hash values of strings instead of their individual characters.

Calculating hash values

A **hash value** of a string is a number that is calculated from the characters of the string. If two strings are the same, their hash values are also the same, which makes it possible to compare strings based on their hash values.

A usual way to implement string hashing is **polynomial hashing**, which means that the hash value of a string s of length n is

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

where $s[0], s[1], \dots, s[n-1]$ are interpreted as the codes of the characters of s , and A and B are pre-chosen constants.

For example, the codes of the characters of ALLEY are:

A	L	L	E	Y
65	76	76	69	89

Thus, if $A = 3$ and $B = 97$, the hash value of ALLEY is

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

Preprocessing

Using polynomial hashing, we can calculate the hash value of any substring of a string s in $O(1)$ time after an $O(n)$ time preprocessing. The idea is to construct an array h such that $h[k]$ contains the hash value of the prefix $s[0 \dots k]$. The array values can be recursively calculated as follows:

$$\begin{aligned} h[0] &= s[0] \\ h[k] &= (h[k-1]A + s[k]) \bmod B \end{aligned}$$

In addition, we construct an array p where $p[k] = A^k \bmod B$:

$$\begin{aligned} p[0] &= 1 \\ p[k] &= (p[k-1]A) \bmod B. \end{aligned}$$

¹The technique was popularized by the Karp–Rabin pattern matching algorithm [42].

Constructing these arrays takes $O(n)$ time. After this, the hash value of any substring $s[a \dots b]$ can be calculated in $O(1)$ time using the formula

$$(h[b] - h[a - 1]p[b - a + 1]) \bmod B$$

assuming that $a > 0$. If $a = 0$, the hash value is simply $h[b]$.

Using hash values

We can efficiently compare strings using hash values. Instead of comparing the individual characters of the strings, the idea is to compare their hash values. If the hash values are equal, the strings are *probably* equal, and if the hash values are different, the strings are *certainly* different.

Using hashing, we can often make a brute force algorithm efficient. As an example, consider the pattern matching problem: given a string s and a pattern p , find the positions where p occurs in s . A brute force algorithm goes through all positions where p may occur and compares the strings character by character. The time complexity of such an algorithm is $O(n^2)$.

We can make the brute force algorithm more efficient by using hashing, because the algorithm compares substrings of strings. Using hashing, each comparison only takes $O(1)$ time, because only hash values of substrings are compared. This results in an algorithm with time complexity $O(n)$, which is the best possible time complexity for this problem.

By combining hashing and *binary search*, it is also possible to find out the lexicographic order of two strings in logarithmic time. This can be done by calculating the length of the common prefix of the strings using binary search. Once we know the length of the common prefix, we can just check the next character after the prefix, because this determines the order of the strings.

Collisions and parameters

An evident risk when comparing hash values is a **collision**, which means that two strings have different contents but equal hash values. In this case, an algorithm that relies on the hash values concludes that the strings are equal, but in reality they are not, and the algorithm may give incorrect results.

Collisions are always possible, because the number of different strings is larger than the number of different hash values. However, the probability of a collision is small if the constants A and B are carefully chosen. A usual way is to choose random constants near 10^9 , for example as follows:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Using such constants, the `long long` type can be used when calculating hash values, because the products AB and BB will fit in `long long`. But is it enough to have about 10^9 different hash values?

Let us consider three scenarios where hashing can be used:

Scenario 1: Strings x and y are compared with each other. The probability of a collision is $1/B$ assuming that all hash values are equally probable.

Scenario 2: A string x is compared with strings y_1, y_2, \dots, y_n . The probability of one or more collisions is

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

Scenario 3: All pairs of strings x_1, x_2, \dots, x_n are compared with each other. The probability of one or more collisions is

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

The following table shows the collision probabilities when $n = 10^6$ and the value of B varies:

constant B	scenario 1	scenario 2	scenario 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

The table shows that in scenario 1, the probability of a collision is negligible when $B \approx 10^9$. In scenario 2, a collision is possible but the probability is still quite small. However, in scenario 3 the situation is very different: a collision will almost always happen when $B \approx 10^9$.

The phenomenon in scenario 3 is known as the **birthday paradox**: if there are n people in a room, the probability that *some* two people have the same birthday is large even if n is quite small. In hashing, correspondingly, when all hash values are compared with each other, the probability that some two hash values are equal is large.

We can make the probability of a collision smaller by calculating *multiple* hash values using different parameters. It is unlikely that a collision would occur in all hash values at the same time. For example, two hash values with parameter $B \approx 10^9$ correspond to one hash value with parameter $B \approx 10^{18}$, which makes the probability of a collision very small.

Some people use constants $B = 2^{32}$ and $B = 2^{64}$, which is convenient, because operations with 32 and 64 bit integers are calculated modulo 2^{32} and 2^{64} . However, this is *not* a good choice, because it is possible to construct inputs that always generate collisions when constants of the form 2^x are used [51].

26.4 Z-algorithm

The **Z-array** z of a string s of length n contains for each $k = 0, 1, \dots, n-1$ the length of the longest substring of s that begins at position k and is a prefix of

s. Thus, $z[k] = p$ tells us that $s[0 \dots p-1]$ equals $s[k \dots k+p-1]$. Many string processing problems can be efficiently solved using the Z-array.

For example, the Z-array of ACBACDACBACBACDA is as follows:

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

In this case, for example, $z[6] = 5$, because the substring ACBAC of length 5 is a prefix of s, but the substring ACBACB of length 6 is not a prefix of s.

Algorithm description

Next we describe an algorithm, called the **Z-algorithm**², that efficiently constructs the Z-array in $O(n)$ time. The algorithm calculates the Z-array values from left to right by both using information already stored in the Z-array and comparing substrings character by character.

To efficiently calculate the Z-array values, the algorithm maintains a range $[x, y]$ such that $s[x \dots y]$ is a prefix of s and y is as large as possible. Since we know that $s[0 \dots y-x]$ and $s[x \dots y]$ are equal, we can use this information when calculating Z-values for positions $x+1, x+2, \dots, y$.

At each position k , we first check the value of $z[k-x]$. If $k+z[k-x] < y$, we know that $z[k] = z[k-x]$. However, if $k+z[k-x] \geq y$, $s[0 \dots y-k]$ equals $s[k \dots y]$, and to determine the value of $z[k]$ we need to compare the substrings character by character. Still, the algorithm works in $O(n)$ time, because we start comparing at positions $y-k+1$ and $y+1$.

For example, let us construct the following Z-array:

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
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

After calculating the value $z[6] = 5$, the current $[x, y]$ range is $[6, 10]$:

						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	?	?	?	?	?	?	?	?	?

Now we can calculate subsequent Z-array values efficiently, because we know that $s[0 \dots 4]$ and $s[6 \dots 10]$ are equal. First, since $z[1] = z[2] = 0$, we immediately know that also $z[7] = z[8] = 0$:

²The Z-algorithm was presented in [32] as the simplest known method for linear-time pattern matching, and the original idea was attributed to [50].

						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	?	?	?	?	?	?	?

Then, since $z[3] = 2$, we know that $z[9] \geq 2$:

						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	?	?	?	?	?	?	?

However, we have no information about the string after position 10, so we need to compare the substrings character by character:

						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	?	?	?	?	?	?	?

It turns out that $z[9] = 7$, so the new $[x, y]$ range is $[9, 15]$:

									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	?	?	?	?	?	?

After this, all the remaining Z-array values can be determined by using the information already stored in the Z-array:

									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

Using the Z-array

It is often a matter of taste whether to use string hashing or the Z-algorithm. Unlike hashing, the Z-algorithm always works and there is no risk for collisions. On the other hand, the Z-algorithm is more difficult to implement and some problems can only be solved using hashing.

As an example, consider again the pattern matching problem, where our task is to find the occurrences of a pattern p in a string s . We already solved this problem efficiently using string hashing, but the Z-algorithm provides another way to solve the problem.

A usual idea in string processing is to construct a string that consists of multiple strings separated by special characters. In this problem, we can construct a string $p\#s$, where p and s are separated by a special character $\#$ that does not occur in the strings. The Z-array of $p\#s$ tells us the positions where p occurs in s , because such positions contain the length of p .

For example, if $s = \text{HATTIVATTI}$ and $p = \text{ATT}$, the Z-array is as follows:

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

The positions 5 and 10 contain the value 3, which means that the pattern ATT occurs in the corresponding positions of HATTIVATTI.

The time complexity of the resulting algorithm is linear, because it suffices to construct the Z-array and go through its values.

Implementation

Here is a short implementation of the Z-algorithm that returns a vector that corresponds to the Z-array.

```
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

Square root algorithms

A **square root algorithm** is an algorithm that has a square root in its time complexity. A square root can be seen as a "poor man's logarithm": the complexity $O(\sqrt{n})$ is better than $O(n)$ but worse than $O(\log n)$. In any case, many square root algorithms are fast and usable in practice.

As an example, consider the problem of creating a data structure that supports two operations on an array: modifying an element at a given position and calculating the sum of elements in the given range. We have previously solved the problem using binary indexed and segment trees, that support both operations in $O(\log n)$ time. However, now we will solve the problem in another way using a square root structure that allows us to modify elements in $O(1)$ time and calculate sums in $O(\sqrt{n})$ time.

The idea is to divide the array into *blocks* of size \sqrt{n} so that each block contains the sum of elements inside the block. For example, an array of 16 elements will be divided into blocks of 4 elements as follows:


21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

In this structure, it is easy to modify array elements, because it is only needed to update the sum of a single block after each modification, which can be done in $O(1)$ time. For example, the following picture shows how the value of an element and the sum of the corresponding block change:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Then, to calculate the sum of elements in a range, we divide the range into three parts such that the sum consists of values of single elements and sums of blocks between them:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2



Since the number of single elements is $O(\sqrt{n})$ and the number of blocks is also $O(\sqrt{n})$, the sum query takes $O(\sqrt{n})$ time. The purpose of the block size \sqrt{n} is that it *balances* two things: the array is divided into \sqrt{n} blocks, each of which contains \sqrt{n} elements.

In practice, it is not necessary to use the exact value of \sqrt{n} as a parameter, and instead we may use parameters k and n/k where k is different from \sqrt{n} . The optimal parameter depends on the problem and input. For example, if an algorithm often goes through the blocks but rarely inspects single elements inside the blocks, it may be a good idea to divide the array into $k < \sqrt{n}$ blocks, each of which contains $n/k > \sqrt{n}$ elements.

27.1 Combining algorithms

In this section we discuss two square root algorithms that are based on combining two algorithms into one algorithm. In both cases, we could use either of the algorithms without the other and solve the problem in $O(n^2)$ time. However, by combining the algorithms, the running time is only $O(n\sqrt{n})$.

Case processing

Suppose that we are given a two-dimensional grid that contains n cells. Each cell is assigned a letter, and our task is to find two cells with the same letter whose distance is minimum, where the distance between cells (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$. For example, consider the following grid:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

In this case, the minimum distance is 2 between the two 'E' letters.

We can solve the problem by considering each letter separately. Using this approach, the new problem is to calculate the minimum distance between two cells with a *fixed* letter c . We focus on two algorithms for this:

Algorithm 1: Go through all pairs of cells with letter c , and calculate the minimum distance between such cells. This will take $O(k^2)$ time where k is the number of cells with letter c .

Algorithm 2: Perform a breadth-first search that simultaneously starts at each cell with letter c . The minimum distance between two cells with letter c will be calculated in $O(n)$ time.

One way to solve the problem is to choose either of the algorithms and use it for all letters. If we use Algorithm 1, the running time is $O(n^2)$, because all cells may contain the same letter, and in this case $k = n$. Also if we use Algorithm 2, the running time is $O(n^2)$, because all cells may have different letters, and in this case n searches are needed.

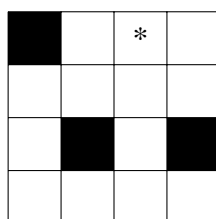
However, we can *combine* the two algorithms and use different algorithms for different letters depending on how many times each letter appears in the grid. Assume that a letter c appears k times. If $k \leq \sqrt{n}$, we use Algorithm 1, and if $k > \sqrt{n}$, we use Algorithm 2. It turns out that by doing this, the total running time of the algorithm is only $O(n\sqrt{n})$.

First, suppose that we use Algorithm 1 for a letter c . Since c appears at most \sqrt{n} times in the grid, we compare each cell with letter c $O(\sqrt{n})$ times with other cells. Thus, the time used for processing all such cells is $O(n\sqrt{n})$. Then, suppose that we use Algorithm 2 for a letter c . There are at most \sqrt{n} such letters, so processing those letters also takes $O(n\sqrt{n})$ time.

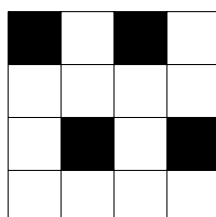
Batch processing

Our next problem also deals with a two-dimensional grid that contains n cells. Initially, each cell except one is white. We perform $n - 1$ operations, each of which first calculates the minimum distance from a given white cell to a black cell, and then paints the white cell black.

For example, consider the following operation:



First, we calculate the minimum distance from the white cell marked with $*$ to a black cell. The minimum distance is 2, because we can move two steps left to a black cell. Then, we paint the white cell black:



Consider the following two algorithms:

Algorithm 1: Use breadth-first search to calculate for each white cell the distance to the nearest black cell. This takes $O(n)$ time, and after the search, we can find the minimum distance from any white cell to a black cell in $O(1)$ time.

Algorithm 2: Maintain a list of cells that have been painted black, go through this list at each operation and then add a new cell to the list. An operation takes $O(k)$ time where k is the length of the list.

We combine the above algorithms by dividing the operations into $O(\sqrt{n})$ *batches*, each of which consists of $O(\sqrt{n})$ operations. At the beginning of each batch, we perform Algorithm 1. Then, we use Algorithm 2 to process the operations in the batch. We clear the list of Algorithm 2 between the batches. At each

operation, the minimum distance to a black cell is either the distance calculated by Algorithm 1 or the distance calculated by Algorithm 2.

The resulting algorithm works in $O(n\sqrt{n})$ time. First, Algorithm 1 is performed $O(\sqrt{n})$ times, and each search works in $O(n)$ time. Second, when using Algorithm 2 in a batch, the list contains $O(\sqrt{n})$ cells (because we clear the list between the batches) and each operation takes $O(\sqrt{n})$ time.

27.2 Integer partitions

Some square root algorithms are based on the following observation: if a positive integer n is represented as a sum of positive integers, such a sum always contains at most $O(\sqrt{n})$ *distinct* numbers. The reason for this is that to construct a sum that contains a maximum number of distinct numbers, we should choose *small* numbers. If we choose the numbers $1, 2, \dots, k$, the resulting sum is

$$\frac{k(k+1)}{2}.$$

Thus, the maximum amount of distinct numbers is $k = O(\sqrt{n})$. Next we will discuss two problems that can be solved efficiently using this observation.

Knapsack

Suppose that we are given a list of integer weights whose sum is n . Our task is to find out all sums that can be formed using a subset of the weights. For example, if the weights are $\{1, 3, 3\}$, the possible sums are as follows:

- 0 (empty set)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Using the standard knapsack approach (see Chapter 7.4), the problem can be solved as follows: we define a function $\text{possible}(x, k)$ whose value is 1 if the sum x can be formed using the first k weights, and 0 otherwise. Since the sum of the weights is n , there are at most n weights and all values of the function can be calculated in $O(n^2)$ time using dynamic programming.

However, we can make the algorithm more efficient by using the fact that there are at most $O(\sqrt{n})$ *distinct* weights. Thus, we can process the weights in groups that consists of similar weights. We can process each group in $O(n)$ time, which yields an $O(n\sqrt{n})$ time algorithm.

The idea is to use an array that records the sums of weights that can be formed using the groups processed so far. The array contains n elements: element k is 1 if the sum k can be formed and 0 otherwise. To process a group of weights, we scan the array from left to right and record the new sums of weights that can be formed using this group and the previous groups.

String construction

Given a string s of length n and a set of strings D whose total length is m , consider the problem of counting the number of ways s can be formed as a concatenation of strings in D . For example, if $s = \text{ABAB}$ and $D = \{A, B, AB\}$, there are 4 ways:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

We can solve the problem using dynamic programming: Let $\text{count}(k)$ denote the number of ways to construct the prefix $s[0 \dots k]$ using the strings in D . Now $\text{count}(n - 1)$ gives the answer to the problem, and we can solve the problem in $O(n^2)$ time using a trie structure.

However, we can solve the problem more efficiently by using string hashing and the fact that there are at most $O(\sqrt{m})$ distinct string lengths in D . First, we construct a set H that contains all hash values of the strings in D . Then, when calculating a value of $\text{count}(k)$, we go through all values of p such that there is a string of length p in D , calculate the hash value of $s[k - p + 1 \dots k]$ and check if it belongs to H . Since there are at most $O(\sqrt{m})$ distinct string lengths, this results in an algorithm whose running time is $O(n\sqrt{m})$.

27.3 Mo's algorithm

Mo's algorithm¹ can be used in many problems that require processing range queries in a *static* array, i.e., the array values do not change between the queries. In each query, we are given a range $[a, b]$, and we should calculate a value based on the array elements between positions a and b . Since the array is static, the queries can be processed in any order, and Mo's algorithm processes the queries in a special order which guarantees that the algorithm works efficiently.

Mo's algorithm maintains an *active range* of the array, and the answer to a query concerning the active range is known at each moment. The algorithm processes the queries one by one, and always moves the endpoints of the active range by inserting and removing elements. The time complexity of the algorithm is $O(n\sqrt{n}f(n))$ where the array contains n elements, there are n queries and each insertion and removal of an element takes $O(f(n))$ time.

The trick in Mo's algorithm is the order in which the queries are processed: The array is divided into blocks of $k = O(\sqrt{n})$ elements, and a query $[a_1, b_1]$ is processed before a query $[a_2, b_2]$ if either

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ or
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ and $b_1 < b_2$.

¹According to [12], this algorithm is named after Mo Tao, a Chinese competitive programmer, but the technique has appeared earlier in the literature [44].

Thus, all queries whose left endpoints are in a certain block are processed one after another sorted according to their right endpoints. Using this order, the algorithm only performs $O(n\sqrt{n})$ operations, because the left endpoint moves $O(n)$ times $O(\sqrt{n})$ steps, and the right endpoint moves $O(\sqrt{n})$ times $O(n)$ steps. Thus, both endpoints move a total of $O(n\sqrt{n})$ steps during the algorithm.

Example

As an example, consider a problem where we are given a set of queries, each of them corresponding to a range in an array, and our task is to calculate for each query the number of *distinct* elements in the range.

In Mo's algorithm, the queries are always sorted in the same way, but it depends on the problem how the answer to the query is maintained. In this problem, we can maintain an array `count` where `count[x]` indicates the number of times an element x occurs in the active range.

When we move from one query to another query, the active range changes. For example, if the current range is

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

and the next range is

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

there will be three steps: the left endpoint moves one step to the right, and the right endpoint moves two steps to the right.

After each step, the array count needs to be updated. After adding an element x , we increase the value of `count[x]` by 1, and if `count[x] = 1` after this, we also increase the answer to the query by 1. Similarly, after removing an element x , we decrease the value of `count[x]` by 1, and if `count[x] = 0` after this, we also decrease the answer to the query by 1.

In this problem, the time needed to perform each step is $O(1)$, so the total time complexity of the algorithm is $O(n\sqrt{n})$.

Chapter 28

Segment trees revisited

A segment tree is a versatile data structure that can be used to solve a large number of algorithm problems. However, there are many topics related to segment trees that we have not touched yet. Now is time to discuss some more advanced variants of segment trees.

So far, we have implemented the operations of a segment tree by walking *from bottom to top* in the tree. For example, we have calculated range sums as follows (Chapter 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;
}
```

However, in more advanced segment trees, it is often necessary to implement the operations in another way, *from top to bottom*. Using this approach, the function becomes as follows:

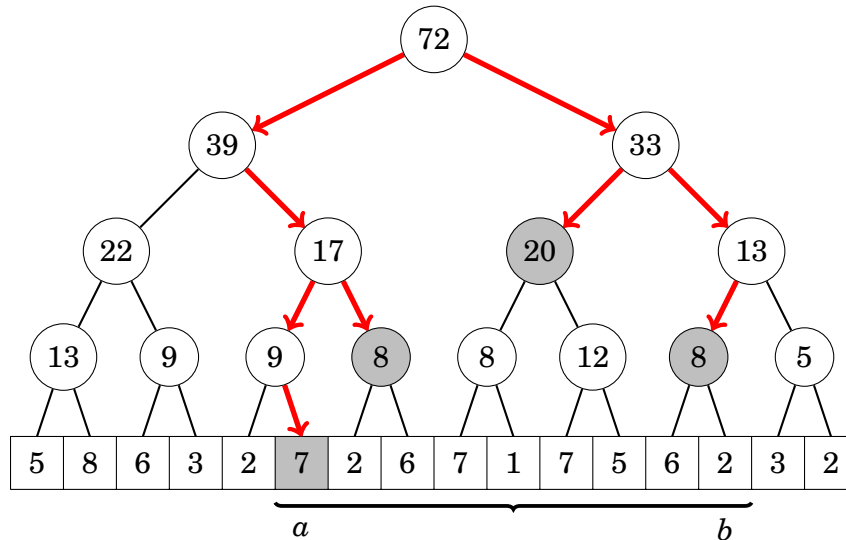
```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Now we can calculate any value of $\text{sum}_q(a, b)$ (the sum of array values in range $[a, b]$) as follows:

```
int s = sum(a, b, 1, 0, n-1);
```

The parameter k indicates the current position in tree. Initially k equals 1, because we begin at the root of the tree. The range $[x, y]$ corresponds to k and is initially $[0, n - 1]$. When calculating the sum, if $[x, y]$ is outside $[a, b]$, the sum is 0, and if $[x, y]$ is completely inside $[a, b]$, the sum can be found in tree. If $[x, y]$ is partially inside $[a, b]$, the search continues recursively to the left and right half of $[x, y]$. The left half is $[x, d]$ and the right half is $[d + 1, y]$ where $d = \lfloor \frac{x+y}{2} \rfloor$.

The following picture shows how the search proceeds when calculating the value of $\text{sum}_q(a, b)$. The gray nodes indicate nodes where the recursion stops and the sum can be found in tree.



Also in this implementation, operations take $O(\log n)$ time, because the total number of visited nodes is $O(\log n)$.

28.1 Lazy propagation

Using **lazy propagation**, we can build a segment tree that supports *both* range updates and range queries in $O(\log n)$ time. The idea is to perform updates and queries from top to bottom and perform updates *lazily* so that they are propagated down the tree only when it is necessary.

In a lazy segment tree, nodes contain two types of information. Like in an ordinary segment tree, each node contains the sum or some other value related to the corresponding subarray. In addition, the node may contain information related to lazy updates, which has not been propagated to its children.

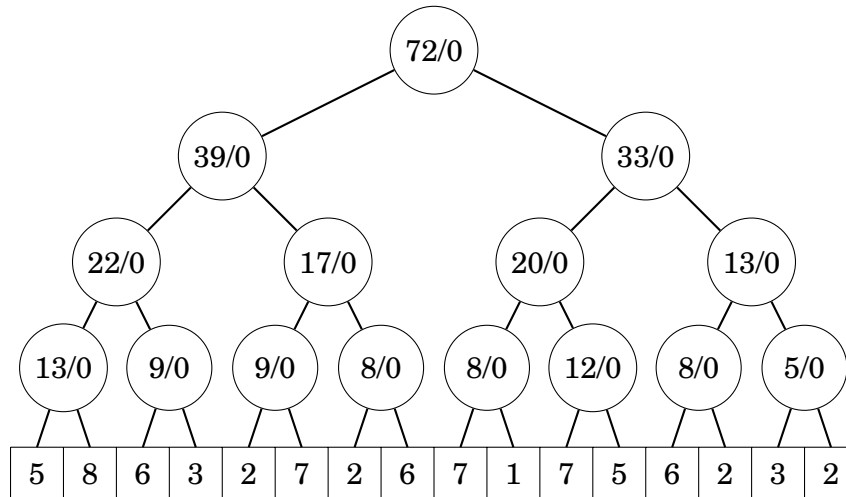
There are two types of range updates: each array value in the range is either *increased* by some value or *assigned* some value. Both operations can be implemented using similar ideas, and it is even possible to construct a tree that supports both operations at the same time.

Lazy segment trees

Let us consider an example where our goal is to construct a segment tree that supports two operations: increasing each value in $[a, b]$ by a constant and calculating

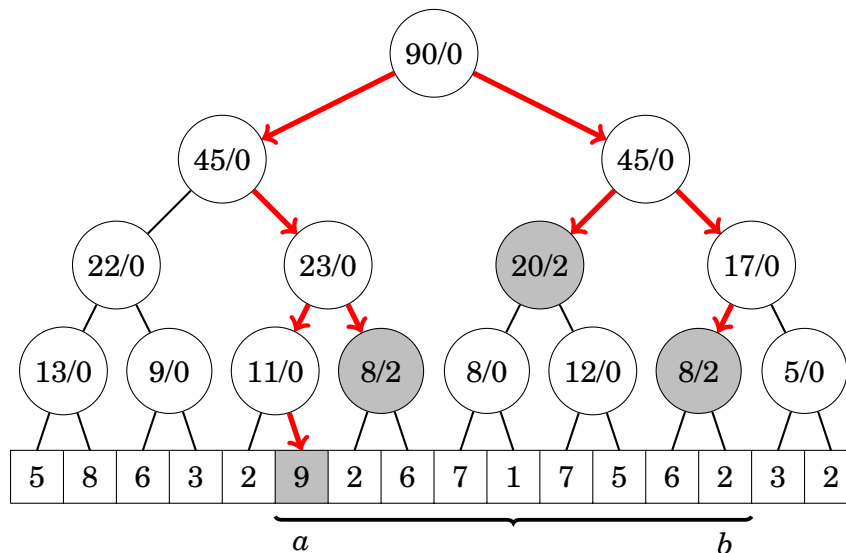
the sum of values in $[a, b]$.

We will construct a tree where each node has two values s/z : s denotes the sum of values in the range, and z denotes the value of a lazy update, which means that all values in the range should be increased by z . In the following tree, $z = 0$ in all nodes, so there are no ongoing lazy updates.



When the elements in $[a, b]$ are increased by u , we walk from the root towards the leaves and modify the nodes of the tree as follows: If the range $[x, y]$ of a node is completely inside $[a, b]$, we increase the z value of the node by u and stop. If $[x, y]$ only partially belongs to $[a, b]$, we increase the s value of the node by hu , where h is the size of the intersection of $[a, b]$ and $[x, y]$, and continue our walk recursively in the tree.

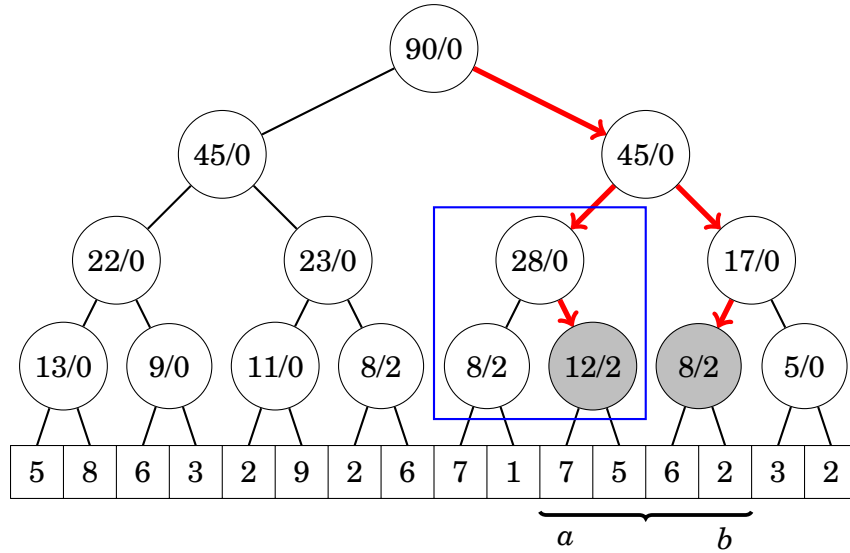
For example, the following picture shows the tree after increasing the elements in $[a, b]$ by 2:



We also calculate the sum of elements in a range $[a, b]$ by walking in the tree from top to bottom. If the range $[x, y]$ of a node completely belongs to $[a, b]$, we add the s value of the node to the sum. Otherwise, we continue the search recursively downwards in the tree.

Both in updates and queries, the value of a lazy update is always propagated to the children of the node before processing the node. The idea is that updates will be propagated downwards only when it is necessary, which guarantees that the operations are always efficient.

The following picture shows how the tree changes when we calculate the value of $\text{sum}_a(a, b)$. The rectangle shows the nodes whose values change, because a lazy update is propagated downwards.



Note that sometimes it is needed to combine lazy updates. This happens when a node that already has a lazy update is assigned another lazy update. When calculating sums, it is easy to combine lazy updates, because the combination of updates z_1 and z_2 corresponds to an update $z_1 + z_2$.

Polynomial updates

Lazy updates can be generalized so that it is possible to update ranges using polynomials of the form

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0.$$

In this case, the update for a value at position i in $[a, b]$ is $p(i - a)$. For example, adding the polynomial $p(u) = u + 1$ to $[a, b]$ means that the value at position a increases by 1, the value at position $a + 1$ increases by 2, and so on.

To support polynomial updates, each node is assigned $k + 2$ values, where k equals the degree of the polynomial. The value s is the sum of the elements in the range, and the values z_0, z_1, \dots, z_k are the coefficients of a polynomial that corresponds to a lazy update.

Now, the sum of values in a range $[x, y]$ equals

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

The value of such a sum can be efficiently calculated using sum formulas. For example, the term z_0 corresponds to the sum $(y - x + 1)z_0$, and the term $z_1 u$ corresponds to the sum

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

When propagating an update in the tree, the indices of $p(u)$ change, because in each range $[x, y]$, the values are calculated for $u = 0, 1, \dots, y - x$. However, this is not a problem, because $p'(u) = p(u + h)$ is a polynomial of equal degree as $p(u)$. For example, if $p(u) = t_2 u^2 + t_1 u - t_0$, then

$$p'(u) = t_2(u + h)^2 + t_1(u + h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

28.2 Dynamic trees

An ordinary segment tree is static, which means that each node has a fixed position in the array and the tree requires a fixed amount of memory. In a **dynamic segment tree**, memory is allocated only for nodes that are actually accessed during the algorithm, which can save a large amount of memory.

The nodes of a dynamic tree can be represented as structs:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Here value is the value of the node, $[x, y]$ is the corresponding range, and left and right point to the left and right subtree.

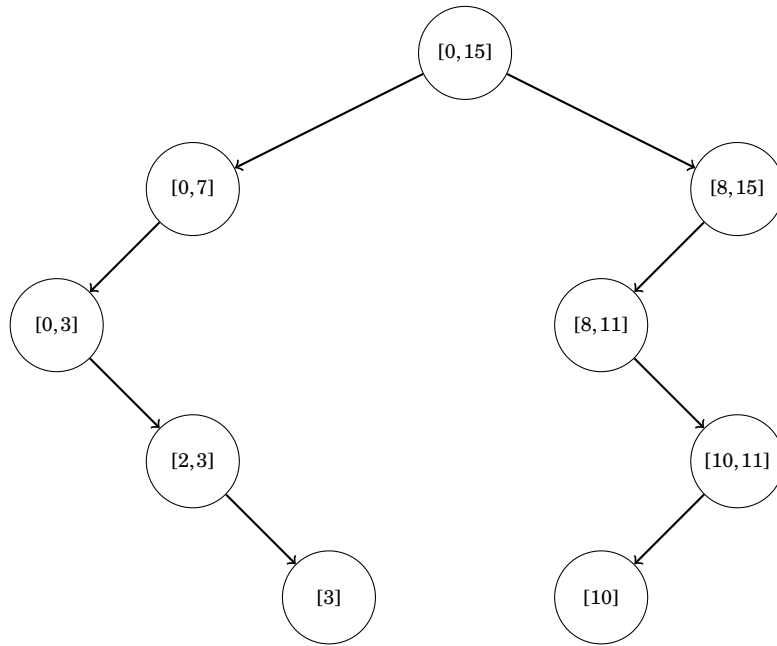
After this, nodes can be created as follows:

```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

Sparse segment trees

A dynamic segment tree is useful when the underlying array is *sparse*, i.e., the range $[0, n - 1]$ of allowed indices is large, but most array values are zeros. While an ordinary segment tree uses $O(n)$ memory, a dynamic segment tree only uses $O(k \log n)$ memory, where k is the number of operations performed.

A **sparse segment tree** initially has only one node $[0, n - 1]$ whose value is zero, which means that every array value is zero. After updates, new nodes are dynamically added to the tree. For example, if $n = 16$ and the elements in positions 3 and 10 have been modified, the tree contains the following nodes:



Any path from the root node to a leaf contains $O(\log n)$ nodes, so each operation adds at most $O(\log n)$ new nodes to the tree. Thus, after k operations, the tree contains at most $O(k \log n)$ nodes.

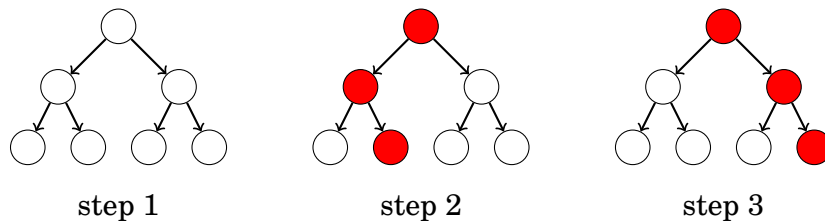
Note that if we know all elements to be updated at the beginning of the algorithm, a dynamic segment tree is not necessary, because we can use an ordinary segment tree with index compression (Chapter 9.4). However, this is not possible when the indices are generated during the algorithm.

Persistent segment trees

Using a dynamic implementation, it is also possible to create a **persistent segment tree** that stores the *modification history* of the tree. In such an implementation, we can efficiently access all versions of the tree that have existed during the algorithm.

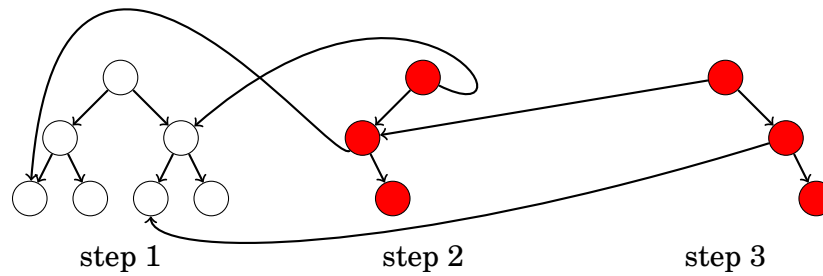
When the modification history is available, we can perform queries in any previous tree like in an ordinary segment tree, because the full structure of each tree is stored. We can also create new trees based on previous trees and modify them independently.

Consider the following sequence of updates, where red nodes change and other nodes remain the same:



After each update, most nodes of the tree remain the same, so an efficient way to store the modification history is to represent each tree in the history as a

combination of new nodes and subtrees of previous trees. In this example, the modification history can be stored as follows:



The structure of each previous tree can be reconstructed by following the pointers starting at the corresponding root node. Since each operation adds only $O(\log n)$ new nodes to the tree, it is possible to store the full modification history of the tree.

28.3 Data structures

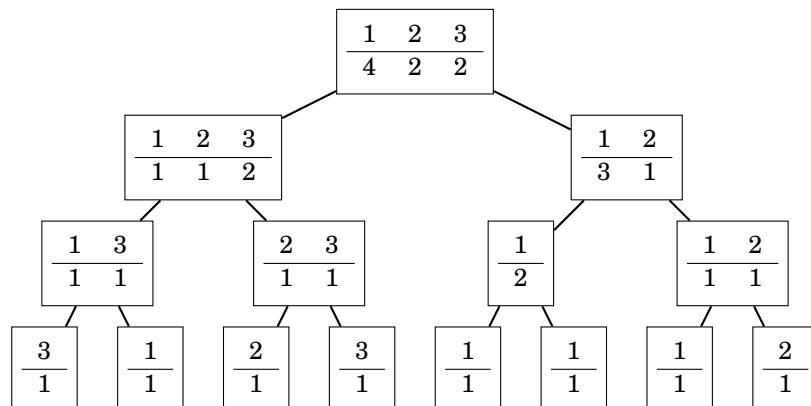
Instead of single values, nodes in a segment tree can also contain *data structures* that maintain information about the corresponding ranges. In such a tree, the operations take $O(f(n)\log n)$ time, where $f(n)$ is the time needed for processing a single node during an operation.

As an example, consider a segment tree that supports queries of the form "how many times does an element x appear in the range $[a, b]$?" For example, the element 1 appears three times in the following range:

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

To support such queries, we build a segment tree where each node is assigned a data structure that can be asked how many times any element x appears in the corresponding range. Using this tree, the answer to a query can be calculated by combining the results from the nodes that belong to the range.

For example, the following segment tree corresponds to the above array:



We can build the tree so that each node contains a map structure. In this case, the time needed for processing each node is $O(\log n)$, so the total time complexity of a query is $O(\log^2 n)$. The tree uses $O(n \log n)$ memory, because there are $O(\log n)$ levels and each level contains $O(n)$ elements.

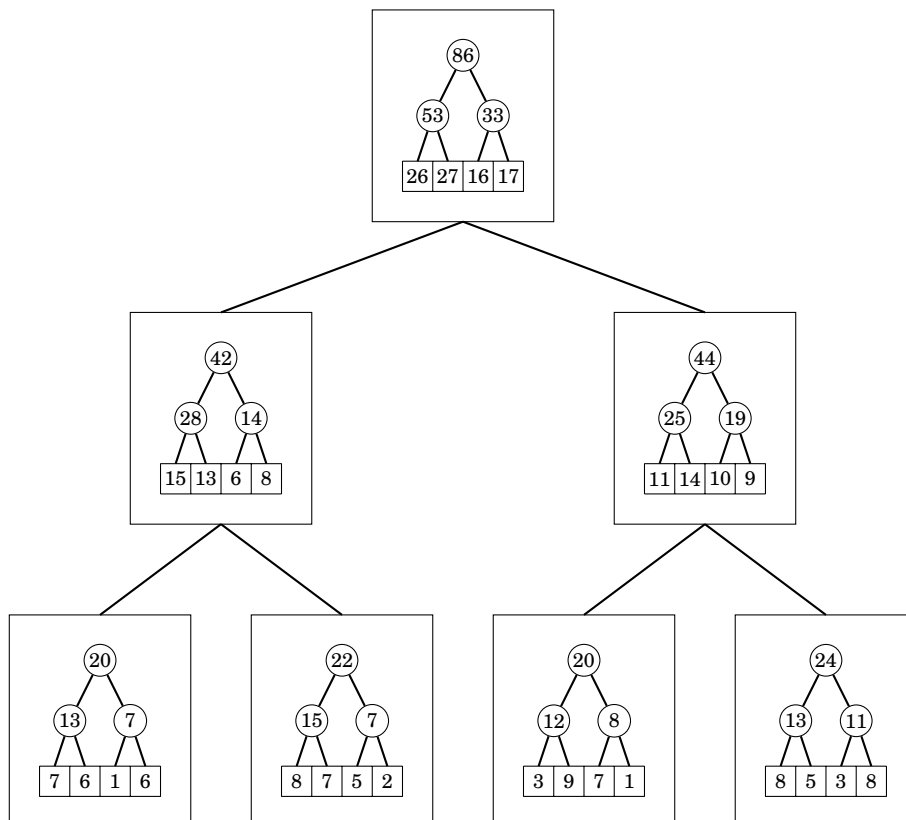
28.4 Two-dimensionality

A **two-dimensional segment tree** supports queries related to rectangular subarrays of a two-dimensional array. Such a tree can be implemented as nested segment trees: a big tree corresponds to the rows of the array, and each node contains a small tree that corresponds to a column.

For example, in the array

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

the sum of any subarray can be calculated from the following segment tree:



The operations of a two-dimensional segment tree take $O(\log^2 n)$ time, because the big tree and each small tree consist of $O(\log n)$ levels. The tree requires $O(n^2)$ memory, because each small tree contains $O(n)$ values.

Chapter 29

Geometry

In geometric problems, it is often challenging to find a way to approach the problem so that the solution to the problem can be conveniently implemented and the number of special cases is small.

As an example, consider a problem where we are given the vertices of a quadrilateral (a polygon that has four vertices), and our task is to calculate its area. For example, a possible input for the problem is as follows:



One way to approach the problem is to divide the quadrilateral into two triangles by a straight line between two opposite vertices:



After this, it suffices to sum the areas of the triangles. The area of a triangle can be calculated, for example, using **Heron's formula**

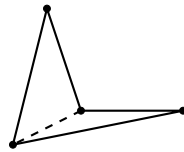
$$\sqrt{s(s-a)(s-b)(s-c)},$$

where a , b and c are the lengths of the triangle's sides and $s = (a + b + c)/2$.

This is a possible way to solve the problem, but there is one pitfall: how to divide the quadrilateral into triangles? It turns out that sometimes we cannot just pick two arbitrary opposite vertices. For example, in the following situation, the division line is *outside* the quadrilateral:



However, another way to draw the line works:



It is clear for a human which of the lines is the correct choice, but the situation is difficult for a computer.

However, it turns out that we can solve the problem using another method that is more convenient to a programmer. Namely, there is a general formula

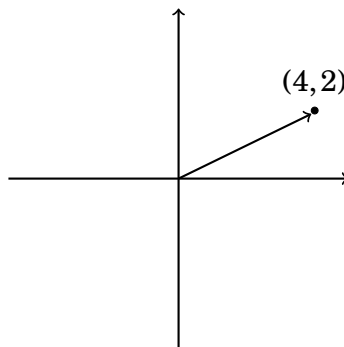
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

that calculates the area of a quadrilateral whose vertices are (x_1, y_1) , (x_2, y_2) , (x_3, y_3) and (x_4, y_4) . This formula is easy to implement, there are no special cases, and we can even generalize the formula to *all* polygons.

29.1 Complex numbers

A **complex number** is a number of the form $x + yi$, where $i = \sqrt{-1}$ is the **imaginary unit**. A geometric interpretation of a complex number is that it represents a two-dimensional point (x, y) or a vector from the origin to a point (x, y) .

For example, $4 + 2i$ corresponds to the following point and vector:



The C++ complex number class `complex` is useful when solving geometric problems. Using the class we can represent points and vectors as complex numbers, and the class contains tools that are useful in geometry.

In the following code, `C` is the type of a coordinate and `P` is the type of a point or a vector. In addition, the code defines macros `X` and `Y` that can be used to refer to `x` and `y` coordinates.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

For example, the following code defines a point $p = (4, 2)$ and prints its x and y coordinates:

```
P p = {4,2};  
cout << p.X << " " << p.Y << "\n"; // 4 2
```

The following code defines vectors $v = (3, 1)$ and $u = (2, 2)$, and after that calculates the sum $s = v + u$.

```
P v = {3,1};  
P u = {2,2};  
P s = v+u;  
cout << s.X << " " << s.Y << "\n"; // 5 3
```

In practice, an appropriate coordinate type is usually `long long` (integer) or `long double` (real number). It is a good idea to use integer whenever possible, because calculations with integers are exact. If real numbers are needed, precision errors should be taken into account when comparing numbers. A safe way to check if real numbers a and b are equal is to compare them using $|a - b| < \epsilon$, where ϵ is a small number (for example, $\epsilon = 10^{-9}$).

Functions

In the following examples, the coordinate type is `long double`.

The function `abs(v)` calculates the length $|v|$ of a vector $v = (x, y)$ using the formula $\sqrt{x^2 + y^2}$. The function can also be used for calculating the distance between points (x_1, y_1) and (x_2, y_2) , because that distance equals the length of the vector $(x_2 - x_1, y_2 - y_1)$.

The following code calculates the distance between points $(4, 2)$ and $(3, -1)$:

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.16228
```

The function `arg(v)` calculates the angle of a vector $v = (x, y)$ with respect to the x axis. The function gives the angle in radians, where r radians equals $180r/\pi$ degrees. The angle of a vector that points to the right is 0, and angles decrease clockwise and increase counterclockwise.

The function `polar(s, a)` constructs a vector whose length is s and that points to an angle a . A vector can be rotated by an angle a by multiplying it by a vector with length 1 and angle a .

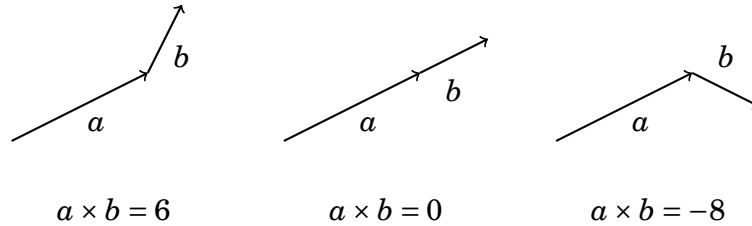
The following code calculates the angle of the vector $(4, 2)$, rotates it $1/2$ radians counterclockwise, and then calculates the angle again:

```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0,0.5);  
cout << arg(v) << "\n"; // 0.963648
```

29.2 Points and lines

The **cross product** $a \times b$ of vectors $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is calculated using the formula $x_1y_2 - x_2y_1$. The cross product tells us whether b turns left (positive value), does not turn (zero) or turns right (negative value) when it is placed directly after a .

The following picture illustrates the above cases:



For example, in the first case $a = (4, 2)$ and $b = (1, 2)$. The following code calculates the cross product using the class `complex`:

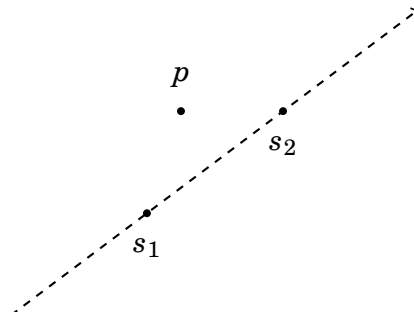
```
P a = {4,2};  
P b = {1,2};  
C p = (conj(a)*b).Y; // 6
```

The above code works, because the function `conj` negates the y coordinate of a vector, and when the vectors $(x_1, -y_1)$ and (x_2, y_2) are multiplied together, the y coordinate of the result is $x_1y_2 - x_2y_1$.

Point location

Cross products can be used to test whether a point is located on the left or right side of a line. Assume that the line goes through points s_1 and s_2 , we are looking from s_1 to s_2 and the point is p .

For example, in the following picture, p is on the left side of the line:

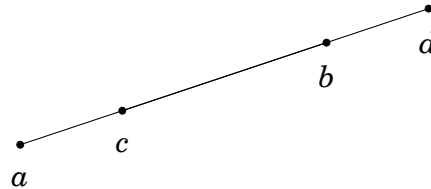


The cross product $(p - s_1) \times (p - s_2)$ tells us the location of the point p . If the cross product is positive, p is located on the left side, and if the cross product is negative, p is located on the right side. Finally, if the cross product is zero, points s_1 , s_2 and p are on the same line.

Line segment intersection

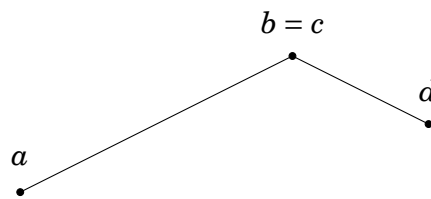
Next we consider the problem of testing whether two line segments ab and cd intersect. The possible cases are:

Case 1: The line segments are on the same line and they overlap each other. In this case, there is an infinite number of intersection points. For example, in the following picture, all points between c and b are intersection points:



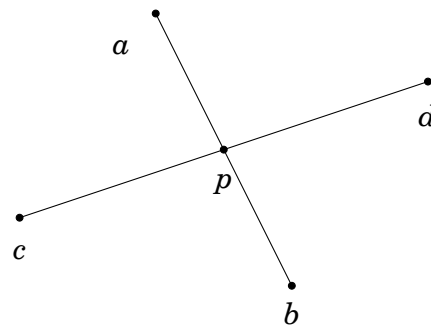
In this case, we can use cross products to check if all points are on the same line. After this, we can sort the points and check whether the line segments overlap each other.

Case 2: The line segments have a common vertex that is the only intersection point. For example, in the following picture the intersection point is $b = c$:



This case is easy to check, because there are only four possibilities for the intersection point: $a = c$, $a = d$, $b = c$ and $b = d$.

Case 3: There is exactly one intersection point that is not a vertex of any line segment. In the following picture, the point p is the intersection point:



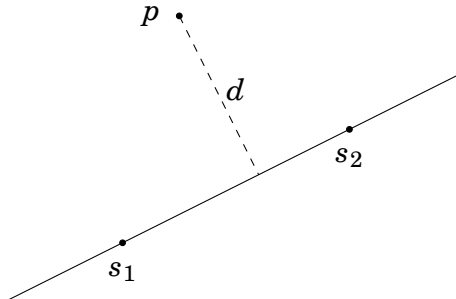
In this case, the line segments intersect exactly when both points c and d are on different sides of a line through a and b , and points a and b are on different sides of a line through c and d . We can use cross products to check this.

Point distance from a line

Another feature of cross products is that the area of a triangle can be calculated using the formula

$$\frac{|(a - c) \times (b - c)|}{2},$$

where a , b and c are the vertices of the triangle. Using this fact, we can derive a formula for calculating the shortest distance between a point and a line. For example, in the following picture d is the shortest distance between the point p and the line that is defined by the points s_1 and s_2 :

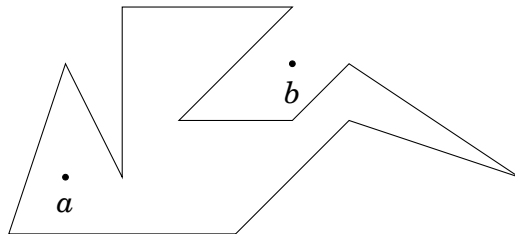


The area of the triangle whose vertices are s_1 , s_2 and p can be calculated in two ways: it is both $\frac{1}{2}|s_2 - s_1|d$ and $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Thus, the shortest distance is

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

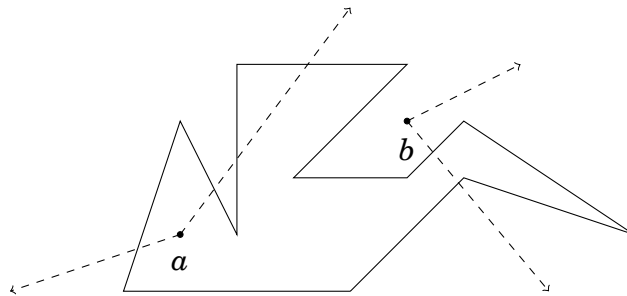
Point inside a polygon

Let us now consider the problem of testing whether a point is located inside or outside a polygon. For example, in the following picture point a is inside the polygon and point b is outside the polygon.



A convenient way to solve the problem is to send a *ray* from the point to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, and if the number is even, the point is outside the polygon.

For example, we could send the following rays:



The rays from a touch 1 and 3 times the boundary of the polygon, so a is inside the polygon. Correspondingly, the rays from b touch 0 and 2 times the boundary of the polygon, so b is outside the polygon.

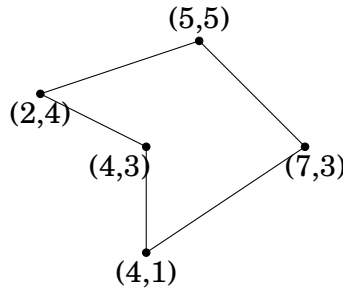
29.3 Polygon area

A general formula for calculating the area of a polygon, sometimes called the **shoelace formula**, is as follows:

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

Here the vertices are $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ in such an order that p_i and p_{i+1} are adjacent vertices on the boundary of the polygon, and the first and last vertex is the same, i.e., $p_1 = p_n$.

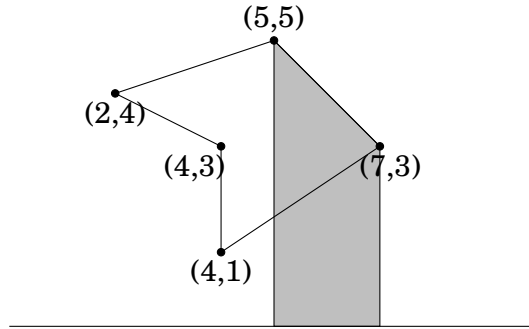
For example, the area of the polygon



is

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

The idea of the formula is to go through trapezoids whose one side is a side of the polygon, and another side lies on the horizontal line $y = 0$. For example:



The area of such a trapezoid is

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

where the vertices of the polygon are p_i and p_{i+1} . If $x_{i+1} > x_i$, the area is positive, and if $x_{i+1} < x_i$, the area is negative.

The area of the polygon is the sum of areas of all such trapezoids, which yields the formula

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Note that the absolute value of the sum is taken, because the value of the sum may be positive or negative, depending on whether we walk clockwise or counterclockwise along the boundary of the polygon.

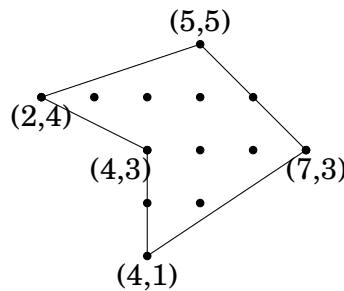
Pick's theorem

Pick's theorem provides another way to calculate the area of a polygon provided that all vertices of the polygon have integer coordinates. According to Pick's theorem, the area of the polygon is

$$a + b/2 - 1,$$

where a is the number of integer points inside the polygon and b is the number of integer points on the boundary of the polygon.

For example, the area of the polygon



is $6 + 7/2 - 1 = 17/2$.

29.4 Distance functions

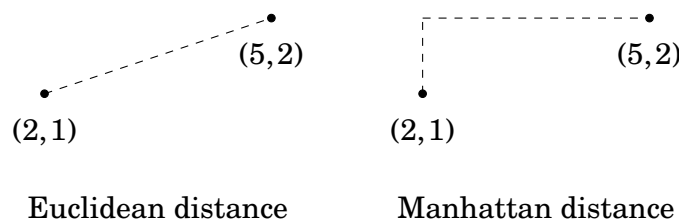
A **distance function** defines the distance between two points. The usual distance function is the **Euclidean distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

An alternative distance function is the **Manhattan distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$|x_1 - x_2| + |y_1 - y_2|.$$

For example, consider the following picture:



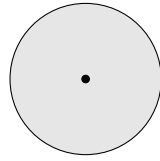
The Euclidean distance between the points is

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

and the Manhattan distance is

$$|5 - 2| + |2 - 1| = 4.$$

The following picture shows regions that are within a distance of 1 from the center point, using the Euclidean and Manhattan distances:



Euclidean distance

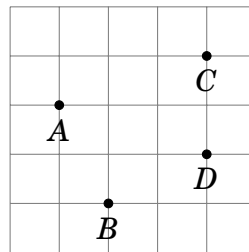


Manhattan distance

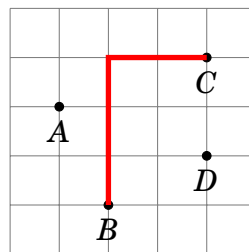
Rotating coordinates

Some problems are easier to solve if Manhattan distances are used instead of Euclidean distances. As an example, consider a problem where we are given n points in the two-dimensional plane and our task is to calculate the maximum Manhattan distance between any two points.

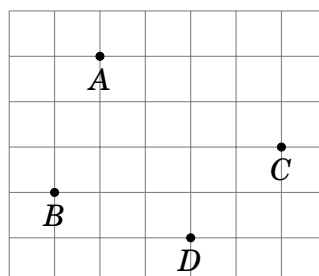
For example, consider the following set of points:



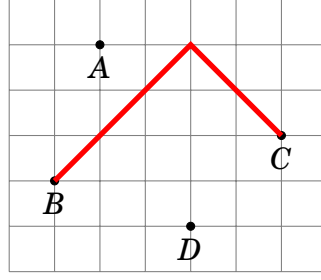
The maximum Manhattan distance is 5 between points B and C :



A useful technique related to Manhattan distances is to rotate all coordinates 45 degrees so that a point (x, y) becomes $(x + y, y - x)$. For example, after rotating the above points, the result is:



And the maximum distance is as follows:



Consider two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ whose rotated coordinates are $p'_1 = (x'_1, y'_1)$ and $p'_2 = (x'_2, y'_2)$. Now there are two ways to express the Manhattan distance between p_1 and p_2 :

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

For example, if $p_1 = (1, 0)$ and $p_2 = (3, 3)$, the rotated coordinates are $p'_1 = (1, -1)$ and $p'_2 = (6, 0)$ and the Manhattan distance is

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

The rotated coordinates provide a simple way to operate with Manhattan distances, because we can consider x and y coordinates separately. To maximize the Manhattan distance between two points, we should find two points whose rotated coordinates maximize the value of

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|).$$

This is easy, because either the horizontal or vertical difference of the rotated coordinates has to be maximum.

Chapter 30

Sweep line algorithms

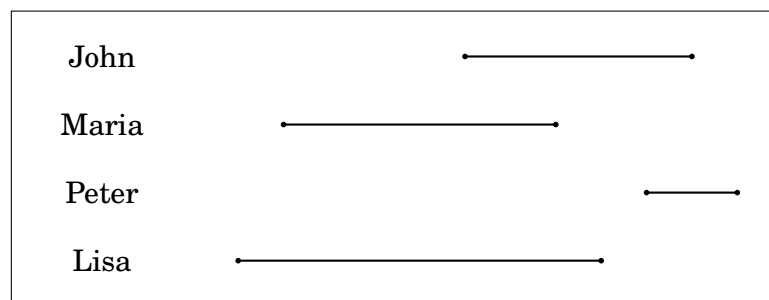
Many geometric problems can be solved using **sweep line** algorithms. The idea in such algorithms is to represent an instance of the problem as a set of events that correspond to points in the plane. The events are processed in increasing order according to their x or y coordinates.

As an example, consider the following problem: There is a company that has n employees, and we know for each employee their arrival and leaving times on a certain day. Our task is to calculate the maximum number of employees that were in the office at the same time.

The problem can be solved by modeling the situation so that each employee is assigned two events that correspond to their arrival and leaving times. After sorting the events, we go through them and keep track of the number of people in the office. For example, the table

person	arrival time	leaving time
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

corresponds to the following events:



We go through the events from left to right and maintain a counter. Always when a person arrives, we increase the value of the counter by one, and when a person leaves, we decrease the value of the counter by one. The answer to the problem is the maximum value of the counter during the algorithm.

In the example, the events are processed as follows:

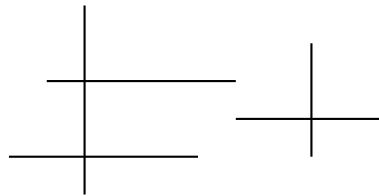


The symbols + and – indicate whether the value of the counter increases or decreases, and the value of the counter is shown below. The maximum value of the counter is 3 between John’s arrival and Maria’s leaving.

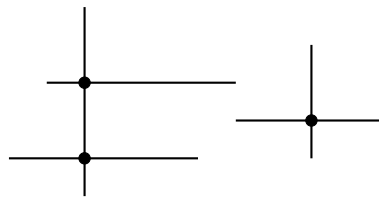
The running time of the algorithm is $O(n \log n)$, because sorting the events takes $O(n \log n)$ time and the rest of the algorithm takes $O(n)$ time.

30.1 Intersection points

Given a set of n line segments, each of them being either horizontal or vertical, consider the problem of counting the total number of intersection points. For example, when the line segments are



there are three intersection points:

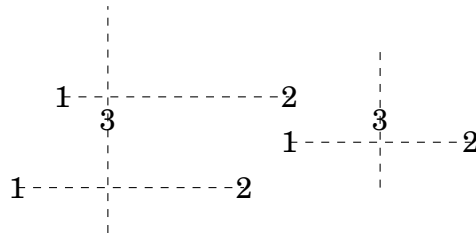


It is easy to solve the problem in $O(n^2)$ time, because we can go through all possible pairs of line segments and check if they intersect. However, we can solve the problem more efficiently in $O(n \log n)$ time using a sweep line algorithm and a range query data structure.

The idea is to process the endpoints of the line segments from left to right and focus on three types of events:

- (1) horizontal segment begins
- (2) horizontal segment ends
- (3) vertical segment

The following events correspond to the example:



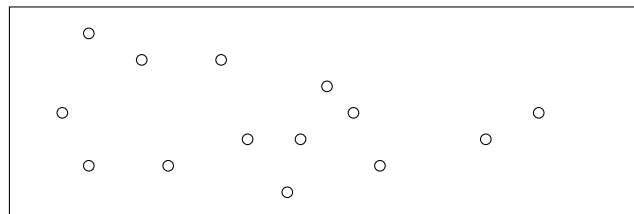
We go through the events from left to right and use a data structure that maintains a set of y coordinates where there is an active horizontal segment. At event 1, we add the y coordinate of the segment to the set, and at event 2, we remove the y coordinate from the set.

Intersection points are calculated at event 3. When there is a vertical segment between points y_1 and y_2 , we count the number of active horizontal segments whose y coordinate is between y_1 and y_2 , and add this number to the total number of intersection points.

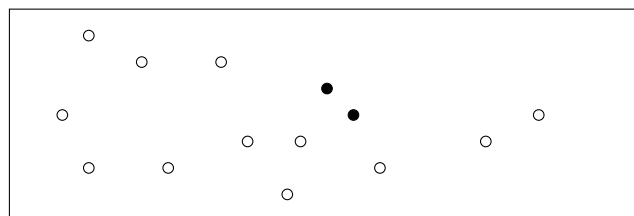
To store y coordinates of horizontal segments, we can use a binary indexed or segment tree, possibly with index compression. When such structures are used, processing each event takes $O(\log n)$ time, so the total running time of the algorithm is $O(n \log n)$.

30.2 Closest pair problem

Given a set of n points, our next problem is to find two points whose Euclidean distance is minimum. For example, if the points are



we should find the following points:



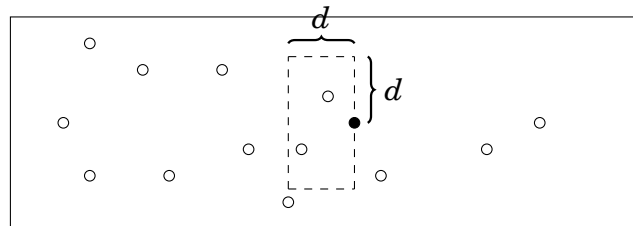
This is another example of a problem that can be solved in $O(n \log n)$ time using a sweep line algorithm¹. We go through the points from left to right and maintain a value d : the minimum distance between two points seen so far. At

¹Besides this approach, there is also an $O(n \log n)$ time divide-and-conquer algorithm [56] that divides the points into two sets and recursively solves the problem for both sets.

each point, we find the nearest point to the left. If the distance is less than d , it is the new minimum distance and we update the value of d .

If the current point is (x, y) and there is a point to the left within a distance of less than d , the x coordinate of such a point must be between $[x - d, x]$ and the y coordinate must be between $[y - d, y + d]$. Thus, it suffices to only consider points that are located in those ranges, which makes the algorithm efficient.

For example, in the following picture, the region marked with dashed lines contains the points that can be within a distance of d from the active point:



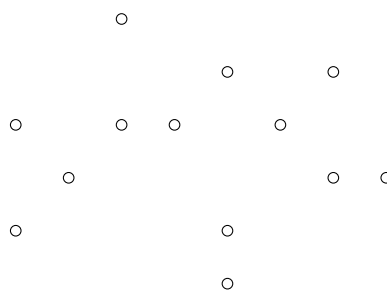
The efficiency of the algorithm is based on the fact that the region always contains only $O(1)$ points. We can go through those points in $O(\log n)$ time by maintaining a set of points whose x coordinate is between $[x - d, x]$, in increasing order according to their y coordinates.

The time complexity of the algorithm is $O(n \log n)$, because we go through n points and find for each point the nearest point to the left in $O(\log n)$ time.

30.3 Convex hull problem

A **convex hull** is the smallest convex polygon that contains all points of a given set. Convexity means that a line segment between any two vertices of the polygon is completely inside the polygon.

For example, for the points



the convex hull is as follows:



Andrew's algorithm [3] provides an easy way to construct the convex hull for a set of points in $O(n \log n)$ time. The algorithm first locates the leftmost and rightmost points, and then constructs the convex hull in two parts: first the upper hull and then the lower hull. Both parts are similar, so we can focus on constructing the upper hull.

First, we sort the points primarily according to x coordinates and secondarily according to y coordinates. After this, we go through the points and add each point to the hull. Always after adding a point to the hull, we make sure that the last line segment in the hull does not turn left. As long as it turns left, we repeatedly remove the second last point from the hull.

The following pictures show how Andrew's algorithm works:



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

- 2SAT problem, 168
- 3SAT problem, 170

- adjacency list, 121
- adjacency matrix, 122
- algoritmo cuadrático, 22
- algoritmo cúbico, 22
- Algoritmo de Kadane, 25
- algoritmo de tiempo constante, 22
- algoritmo lineal, 22
- algoritmo logarítmico, 22
- algoritmo polinomial, 23
- algoritmo voraz, 63
- alphabet, 251
- ancestor, 171
- Andrew's algorithm, 287
- antichain, 201
- análisis amortizado, 85
- aritmética modular, 7
- arreglo de diferencias, 101
- arreglo dinámico, 39

- backtracking, 56
- Bellman–Ford algorithm, 131
- binary tree, 147
- binomial coefficient, 216
- binomial distribution, 238
- bipartite graph, 120, 130
- birthday paradox, 255
- bitset, 46
- border, 252
- breadth-first search, 127
- Burnside's lemma, 222
- búsqueda binaria, 34

- Catalan number, 218
- Cayley's formula, 223
- child, 141

- Chinese remainder theorem, 213
- clases de complejidad, 22
- closest pair, 285
- Codificación de Huffman, 69
- cofactor, 227
- cola, 48
- cola de prioridad, 48
- collision, 254
- coloring, 120, 241
- combinatorics, 215
- complejidad temporal, 19
- complemento, 12
- complete graph, 119
- complex, 274
- complex number, 274
- component, 118
- component graph, 165
- compresión de datos, 68
- compresión de índices, 101
- conditional probability, 235
- conjunción, 13
- conjunto, 12
- conjunto universal, 12
- connected graph, 118, 129
- consulta de máximo, 91
- consulta de mínimo, 91
- consulta de rango, 91
- consulta de suma, 91
- coprime, 209
- cross product, 276
- cuantificador, 13
- cut, 190
- cycle, 117, 129, 157, 163
- cycle detection, 163
- código binario, 68

- De Bruijn sequence, 186

degree, 119
 depth-first search, 125
 deque, 47
 derangement, 221
 desplazamiento de bits, 105
 determinant, 227
 diameter, 143
 diferencia, 12
 Dijkstra's algorithm, 134, 161
 Dilworth's theorem, 201
 Diophantine equation, 212
 Dirac's theorem, 185
 directed graph, 118
 distance function, 280
 distancia de edición, 80
 Distancia de Hamming, 108
 distancia de Levenshtein, 80
 distribution, 237
 disyunción, 13
 divisibility, 205
 divisor, 205
 dynamic segment tree, 269

 edge, 117
 edge list, 123
 Edmonds–Karp algorithm, 192
 elementos menores más cercanos, 87
 encuentro en el medio, 61
 entero, 6
 entrada y salida, 4
 equivalencia, 13
 estructura de datos, 39
 Euclid's algorithm, 208
 Euclid's formula, 214
 Euclidean distance, 280
 Euler tour technique, 176
 Euler's theorem, 210
 Euler's totient function, 209
 Eulerian circuit, 182
 Eulerian path, 181
 expected value, 237
 extended Euclid's algorithm, 212

 factor, 205
 factor constante, 23
 factorial, 14
 Fermat's theorem, 210

 Fibonacci number, 214, 228
 flow, 189
 Floyd's algorithm, 164
 Floyd–Warshall algorithm, 137
 Ford–Fulkerson algorithm, 190
 Freivalds' algorithm, 240
 función de comparación, 34
 functional graph, 162
 fórmula de Binet, 14
 Fórmula de Faulhaber, 11

 geometric distribution, 238
 geometry, 273
 Goldbach's conjecture, 207
 graph, 117
 greatest common divisor, 208
 Grundy number, 246
 Grundy's game, 249

 Hall's theorem, 197
 Hamiltonian circuit, 185
 Hamiltonian path, 185
 harmonic sum, 208
 hash value, 253
 hashing, 253
 Heron's formula, 273
 heuristic, 187
 Hierholzer's algorithm, 183

 identity matrix, 226
 implicación, 13
 in-order, 147
 inclusion-exclusion, 220
 indegree, 119
 independence, 236
 independent set, 198
 intersección, 12
 intersection point, 284
 inverse matrix, 228
 inversión, 28
 iterador, 43

 Kirchhoff's theorem, 231
 knight's tour, 187
 Kosaraju's algorithm, 166
 Kruskal's algorithm, 150
 König's theorem, 197

Lagrange's theorem, 213
 Laplacean matrix, 232
 Las Vegas algorithm, 239
 lazy propagation, 266
 lazy segment tree, 266
 leaf, 141
 least common multiple, 208
 Legendre's conjecture, 207
 lenguaje de programación, 3
 lexicographical order, 252
 line segment intersection, 277
 linear recurrence, 228
 logaritmo, 15
 logaritmo natural, 15
 logica, 13
 losing state, 243
 lowest common ancestor, 175

 macro, 9
 Manhattan distance, 280
 mapa, 42
 Markov chain, 238
 matching, 195
 matrix, 225
 matrix multiplication, 226, 240
 matrix power, 227
 maximum flow, 189
 maximum independent set, 198
 maximum matching, 195
 maximum spanning tree, 150
 memoización, 73
 mex function, 246
 minimum cut, 190, 193
 minimum node cover, 197
 minimum spanning tree, 149
 misère game, 246
 Mo's algorithm, 263
 mochila, 79
 modular arithmetic, 209
 modular inverse, 210
 Monte Carlo algorithm, 239
 montículo, 48
 multinomial coefficient, 218
 método de dos punteros, 85
 mínimo de ventana deslizante, 89

 negación, 13

negative cycle, 133
 neighbor, 119
 next_permutation, 55
 nim game, 245
 nim sum, 245
 node, 117
 node cover, 197
 number theory, 205
 número de Fibonacci, 14
 número de punto flotante, 7

 operación and, 104
 operación not, 105
 operación or, 105
 operador de comparación, 33
 ordenamiento, 27
 ordenamiento burbuja, 27
 ordenamiento por conteo, 31
 ordenamiento por mezcla, 29
 order statistic, 240
 Ore's theorem, 185
 outdegree, 119

 palabra clave, 68
 pair, 33
 parent, 141
 parenthesis expression, 219
 Pascal's triangle, 217
 path, 117
 path cover, 198
 pattern matching, 251
 perfect matching, 197
 perfect number, 206
 period, 251
 permutación, 55
 persistent segment tree, 270
 Pick's theorem, 280
 point, 274
 polynomial hashing, 253
 post-order, 147
 pre-order, 147
 predicado, 13
 prefix, 251
 Prim's algorithm, 155
 prime, 205
 prime decomposition, 205
 probability, 233

Problema 2SUM, 86
 problema 3SUM, 87
 problema de las reinas, 56
 problema NP-difícil, 23
 programación dinámica, 71
 progresión aritmética, 11
 progresión geométrica, 11
 Prüfer code, 224
 Pythagorean triple, 214

 quickselect, 240
 quicksort, 240

 random variable, 236
 random_shuffle, 44
 randomized algorithm, 239
 regular graph, 119
 representación de bits, 103
 residuo, 7
 reverse, 44
 root, 141
 rooted tree, 141
 rotation, 251

 scaling algorithm, 193
 segment tree, 265
 set, 41
 shoelace formula, 279
 shortest path, 131
 sieve of Eratosthenes, 208
 simple graph, 120
 sort, 32, 44
 spanning tree, 149, 231
 sparse segment tree, 269
 SPFA algorithm, 134
 Sprague–Grundy theorem, 246
 square matrix, 225
 square root algorithm, 259
 stack, 47
 string, 40, 251
 string hashing, 253
 strongly connected component, 165
 strongly connected graph, 165

 subconjunto, 12, 53
 subsecuencia creciente más larga, 76
 subsequence, 251
 substring, 251
 subtree, 141
 successor graph, 162
 suffix, 251
 suma armónica, 12
 suma máxima de subarreglo, 24
 sweep line, 283

 tabla dispersa, 93
 teoría de conjuntos, 12
 topological sorting, 157
 transpose, 225
 tree, 118, 141
 tree query, 171
 tree traversal array, 172
 trie, 252
 tuple, 33
 typedef, 8
 twin prime, 207
 two-dimensional segment tree, 272

 uniform distribution, 238
 union-find structure, 153
 unión, 12

 vector, 39, 225, 274
 ventana deslizante, 89

 Warnsdorf’s rule, 187
 weighted graph, 119
 Wilson’s theorem, 214
 winning state, 243

 Z-algorithm, 255
 Z-array, 255
 Zeckendorf’s theorem, 214

 árbol binario indexado, 94
 árbol de Fenwick, 94
 árbol de segmentos, 97