

Manual del Programador Competitiu

Antti Laaksonen

Draft 4 de gener de 2022

Índex

Prefaci	vii
I Tècniques bàsiques	1
1 Introducció	3
1.1 Llenguatges de programació	3
1.2 Entrada i sortida	4
1.3 Treballar amb nombres	6
1.4 Escurçar el codi	8
1.5 Matemàtiques	10
1.6 Concursos i recursos	15
2 Complexitat temporal	19
2.1 Regles de càlcul	19
2.2 Clases de complexitat	22
2.3 Estimació de l'eficiència	23
2.4 Suma màxima d'un subvector	24
3 Ordenació	27
3.1 Teoria de l'ordenació	27
3.2 Ordenació en C++	31
3.3 Cerca binària	33
4 Estructures de dades	37
4.1 Vectors dinàmics	37
4.2 Estructures conjunt	39
4.3 Estructures mapa	40
4.4 Iteradors i intervals	41
4.5 Altres estructures	43
4.6 Comparació amb l'ordenació	47
5 Cerca completa	49
5.1 Generar subconjunts	49
5.2 Generar permutacions	51
5.3 Backtracking	52
5.4 Podar la cerca	54
5.5 Trobar-se al mig	57

6	Algorismes greedy	59
6.1	Problema de les monedes	59
6.2	Scheduling	60
6.3	Tasques i terminis	62
6.4	Minimitzar sumes	63
6.5	Compressió de dades	64
7	Programació dinàmica	69
7.1	Problema de les monedes	69
7.2	Subseqüència creixent més llarga	74
7.3	Camins en una quadrícula	75
7.4	Problemes de motxilla	77
7.5	Distància d'edició	78
7.6	Comptar rajoles	80
8	Anàlisi amortitzada	83
8.1	Mètode dels dos punters	83
8.2	Element menor més propers	85
8.3	Mínim de finestra lliscant	87
9	Consultes d'interval	89
9.1	Consultes de vector estàtiques	90
9.2	Arbre binari indexat	92
9.3	Arbre de segments	95
9.4	Tècniques addicionals	99
10	Manipulació de bits	101
10.1	Representació de bits	101
10.2	Operacions de bits	102
10.3	Representació de conjunts	104
10.4	Optimitzacions de bits	106
10.5	Programació dinàmica	108
II	Algorismes de grafs	113
11	Introducció als grafs	115
11.1	Vocabulari de grafs	115
11.2	Representació de grafs	118
12	Recorreguts en grafs	123
12.1	Cerca en profunditat	123
12.2	Cerca en amplada	125
12.3	Aplicacions	127

13 Camins més curts	131
13.1 Algorisme de Bellman–Ford	131
13.2 Algorisme de Dijkstra	134
13.3 Algorisme de Floyd-Warshall	137
14 Algorismes d'arbres	141
14.1 Recorregut d'arbres	142
14.2 Diàmetre	143
14.3 Tots els camins més llargs	145
14.4 Arbres binaris	147
15 Arbres d'expansió	149
15.1 Algorisme de Kruskal	150
15.2 Estructura <i>union-find</i>	153
15.3 Algorisme de Prim	155
III Temes avançats	159
Bibliografia	161

Prefaci

L'objectiu d'aquest llibre és donar-vos una introducció completa a la programació competitiva. Se suposa que ja coneixeu els fonaments bàsics de la programació, però no cal cap formació prèvia en programació competitiva.

El llibre està especialment pensat per a estudiants que vulguin aprendre algorismes i possiblement participar en l'Olimpíada Internacional d'Informàtica (IOI) o en el Concurs Internacional de Programació Col·legiata (ICPC). Per descomptat, el llibre també és adequat per a qualsevol altra persona interessada en la programació competitiva.

Es necessita molt de temps per convertir-se en un bon programador competitiu, però també és una oportunitat per aprendre molt. Podeu estar segur que obtindreu una bona comprensió general dels algorismes si passeu temps llegint el llibre, resolent problemes i participant en concursos.

El llibre està en desenvolupament continu. Sempre podeu enviar comentaris sobre el llibre a `ahslaaks@cs.helsinki.fi`.

Helsinki, August 2019
Antti Laaksonen

(Nota del Traductor) L'Antti Laaksonen és l'autor de "Guide to Competitive Programming" (Springer, 2017). Aquest PDF és la traducció al català del seu manual "Competitive Programmer's Handbook" (<https://github.com/p11k/cphb>). Aquesta traducció no hagués estat possible si l'Antti no hagués distribuït les fonts \LaTeX del seu manual sota llicència Creative Common.

Amb aquesta traducció vull permetre que els participants de l'Olimpíada Informàtica de Catalunya *que no saben prou d'anglès* puguin millorar el seu nivell de programació competitiva. Si ets una d'aquestes persones, espero que aquesta traducció et sigui útil! Però recorda que no poder llegir en anglès és literalment una *minusvalia* que t'obliga a dependre de la voluntat i criteri de traductors com jo. Si vols continuar aprenent coses, millora el teu anglès.

Eugene, Desembre 2021
Omer Giménez Llach

Part I

Tècniques bàsiques

Capítol 1

Introducció

La programació competitiva combina dos temes: (1) el disseny d'algorismes i (2) la implementació d'algorismes.

El **disseny d'algorismes** consisteix en la resolució de problemes i el pensament matemàtic. Fan falta habilitats per analitzar problemes i resoldre'ls creativament. Un algorisme per resoldre un problema ha de ser correcte i eficient, i la dificultat consisteix sovint sobre inventar un algorisme eficient.

Els coneixements teòrics d'algorismes són important per als programadors competitius. Normalment, una solució a un problema és una combinació de tècniques conegudes i noves idees. Les tècniques que apareixen en la programació competitiva també constitueixen la base per a la investigació científica d'algorismes.

La **implementació d'algorismes** requereix bones habilitats de programació. En la programació competitiva, les solucions es classifiquen provant un algorisme implementat fent servir un conjunt de casos de prova. Per tant, no n'hi ha prou que la idea del l'algoritme sigui correcte, la implementació també ha de ser correcte.

L'estil correcte de programació en els concursos és ser directe i concís. Els programes s'han d'escriure ràpidament, perquè no hi ha gaire temps disponible. A diferència de l'enginyeria de software tradicional, els programes són curts (normalment com a molt uns pocs centenars de línies de codi) i no cal mantenir el codi un cop acabat el concurs.

1.1 Llenguatges de programació

En l'actualitat, els llenguatges de programació que es fan servir més als concursos són el C++, el Python i el Java. Per exemple, a Google Code Jam 2017, entre els 3.000 millors participants, el 79 % va fer servir C++, el 16 % Python i el 8 % Java [29]. Alguns participants també utilitzaven varis llenguatges.

Molta gent pensa que el C++ és la millor opció per a un programador competitiu, i el C++ gairebé sempre està disponible en els sistemes de concurs. Els avantatges d'utilitzar C++ són que és un llenguatge de programació molt eficient i que la seva biblioteca estàndard conté un gran col·lecció d'estructures de dades i algorismes.

D'altra banda, és bo dominar diversos llenguatges de programació i comprendre els seus punts forts. Per exemple, si es necessiten nombres enters grans en el problema, Python pot ser una bona opció, perquè aquest llenguatge conté operacions integrades per càlculs amb nombres enters grans. Tot i així, la majoria dels problemes en els concursos de programació s'intenten preparar per a que fer servir un llenguatge de programació específic no sigui un avantatge injust.

Tots els programes d'exemple d'aquest llibre estan escrits en C++, i sovint es fa servir les estructures de dades i algorismes de la llibreria estàndard. Els programes segueixen l'estàndard C++11, que es pot fer servir en la majoria dels concursos actuals. Si encara no podeu programar en C++, ara és un bon moment per començar a aprendre.

plantilla de codi C++

Aquesta és una plantilla de codi C++ típica per a la programació competitiva:

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

using namespace std;

int main() {
    // Aquí va el codi.
}
```

La línia `#include` al principi del codi és una característica del compilador `g++` que ens permet incloure tota la biblioteca estàndard. Per tant, no cal incloure per separat biblioteques com ara `iostream`, `vector` i `algorithm`, sinó que estan disponibles automàticament.

La línia `using` declara que les classes i funcions de la biblioteca estàndard es poden fer servir directament al codi. Sense la línia `using` tindríem que escriure, per exemple, `std::cout`, però ara n'hi ha prou amb escriure `cout`.

El codi es pot compilar mitjançant l'ordre següent:

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

Aquesta ordre produeix un fitxer binari `test` a partir del codi font `test.cpp`. El compilador segueix l'estàndard C++11 (`-std=c++11`), optimitza el codi (`-O2`) i mostra avisos sobre possibles errors (`-Wall`).

1.2 Entrada i sortida

A la majoria de concursos es fan servir els canals estàndard per a la lectura d'entrada i l'escriptura de sortida. En C++, els canals estàndard són cin per a l'entrada i `cout` per a la sortida. També es poden fer servir les funcions de C `scanf` i `printf`.

L'entrada per al programa normalment consisteix en nombres i cadenes de text que es separen amb espais i salts de línia. Es poden llegir des del canal cin

com segueix:

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

Aquest tipus de codi sempre funciona, suposant que hi ha almenys un espai o salt de línia entre cada element de l'entrada. Per exemple, el codi anterior pot llegir les dues entrades següents:

```
123 456 monkey
```

```
123 456  
monkey
```

El canal cout es fa servir per a la sortida:

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

L'entrada i la sortida són de vegades un coll d'ampolla en el programa. Les línies següents al principi del codi fan que l'entrada i la sortida siguin més eficients:

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

Tingueu en compte que el salt de línia "\n" funciona més ràpid que endl, perquè endl sempre provoca una operació de buidat del buffer.

Les funcions C scanf i printf són una alternativa als canals estàndard de C++. Normalment són una mica més ràpides, però també són més difícils d'utilitzar. El codi següent llegeix dos nombres enters de l'entrada:

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

El codi següent imprimeix dos nombres enters:

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

De vegades, el programa hauria de llegir una línia sencera de l'entrada, possiblement amb espais. Això es pot aconseguir mitjançant l'ús de la funció getline:

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

Si es desconeix la quantitat de dades, podem fer servir el següent bucle:

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

Aquest bucle llegeix elements de l'entrada un rere l'altre, fins que no n'hi ha hagut més dades disponibles.

En alguns sistemes de concurs es fa servir fitxers per a l'entrada i la sortida. Una solució fàcil per a això és escriure el codi com de costum utilitzant canals estàndard, però afegint les línies següents al començament del codi:

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

Després d'això, el programa llegeix l'entrada del fitxer "input.txt" i escriu la sortida al fitxer "output.txt".

1.3 Treballar amb nombres

Nombres enters

El tipus enter més utilitzat en la programació competitiva és el tipus `int`, que és un tipus de 32 bits amb un rang de valors de $-2^{31} \dots 2^{31} - 1$ o aproximadament $-2 \cdot 10^9 \dots 2 \cdot 10^9$. Si el tipus `int` no és suficient, es pot fer servir el tipus de 64 bits `long long`. Té un rang de valors de $-2^{63} \dots 2^{63} - 1$ o aproximadament $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

El codi següent defineix una variable de tipus `long long`:

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

El sufix `LL` significa que el tipus del nombre és `long long`.

Un error comú quan es fa servir el tipus `long long` és fer servir algun tipus `int` en algun lloc al codi. Per exemple, el codi següent conté un error subtil:

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

Tot i que la variable `b` és del tipus `long long`, els dos nombres de l'expressió `a*a` són del tipus `int` i el resultat és també del tipus `int`. Per això, la variable `b` tindrà un resultat incorrecte. El problema es pot resoldre fent que el tipus de `a` sigui `long long` o canviant l'expressió a `(long long)a*a`.

Normalment els problemes de concurs es plantegen de manera que amb el tipus `long long` n'hi ha prou. Tot i així, és bo saber que el compilador `g++` també proporciona un tipus de 128 bits `__int128_t` amb un rang de valors de $-2^{127} \dots 2^{127} - 1$ o aproximadament $-10^{38} \dots 10^{38}$. Tanmateix, aquest tipus no està disponible en tots els sistemes de concurs.

Aritmètica modular

Denotem per $x \bmod m$ el residu de dividir x per m . Per exemple, $17 \bmod 5 = 2$, perquè $17 = 3 \cdot 5 + 2$.

De vegades, la resposta a un problema és a nombre molt gran però es demana que s'escrigui la solució “mòdul m ”, és a dir, el residu quan es divideix la resposta per m (per exemple, “mòdul $10^9 + 7$ ”). D'aquesta manera, encara que la resposta real sigui molt gran, n'hi ha prou amb utilitzar els tipus `int` i `long long`.

Una propietat important del residu és que en la suma, la resta i la multiplicació, el residu es pot prendre abans de l'operació:

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

Així, podem agafar el residu després de cada operació i les xifres mai seran massa grans.

Per exemple, el codi següent calcula $n!$, el factorial de n , mòdul m :

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

Normalment volem que el residu estigui sempre en $0 \dots m - 1$. Tanmateix, en el C++ i altres llenguatges, el residu d'un nombre negatiu és zero o negatiu. Una manera fàcil d'assegurar-s'hi que no tenim residus negatius es cal calcular primer el residu com de costum i després afegeix m si el resultat és negatiu:

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

Això només és necessari quan hi ha restes en el codi i el residu pot arribar a ser negatiu.

Nombres de coma flotant

Els tipus de coma flotant en la programació competitiva són el `double` de 64 bits i, com a extensió al compilador `g++`, el `long double` de 80 bits. En la majoria dels casos, `double` és suficient, però `long double` és més precís.

La precisió requerida de la resposta normalment es dona a l'enunciat del problema. Una manera fàcil d'emetre la resposta és fer servir la funció `printf` i donar el nombre de decimals a la cadena de text que especifica el format. Per exemple, el codi següent escriu el valor de x amb 9 decimals:

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

Una dificultat quan s'utilitzen nombres de coma flotant és que alguns nombres no es poden representar amb precisió com a nombres de coma flotant, i hi haurà errors d'arrodoniment. Per exemple, el resultat del codi següent és sorprenent:

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

El valor de x és una mica més petit que 1 degut als errors d'arrodoniment, quan el valor correcte seria 1.

És arriscat comparar nombres de coma flotant amb l'operador `==`, perquè és possible que, encara els valors haurien de ser iguals, en realitat no ho són degut a errors de precisió. Una millor manera de comparar nombres de coma flotant és suposar que dos nombres són iguals si la diferència entre ells és menor que ε , on ε és un nombre petit.

A la pràctica, els nombres es poden comparar de la següent manera ($\varepsilon = 10^{-9}$):

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

Tingueu en compte que, tot i que els nombres de coma flotant són inexactes, els enters fins a un cert límit encara es poden representar amb precisió. Per exemple, fent servir `double`, és possible representar amb precisió nombres enters el valor absolut dels quals és com a màxim 2^{53} .

1.4 Esgurçar el codi

El codi curt és ideal en programació competitiva, perquè els programes s'han d'escriure el més ràpid possible. Per això, els programadors competitius sovint defineixen noms més curts per a tipus de dades i altres parts del codi.

Noms dels tipus

Fent servir l'ordre `typedef` és possible donar un nom més curt a un tipus de dades. Per exemple, com que el nom `long long` és llarg, podem definir un nom més curt `ll` d'aquesta manera:

```
typedef long long ll;
```

Després d'això, el codi

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

es pot escurçar de la següent manera:

```
ll a = 123456789;
```



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

L'ordre typedef també es pot fer servir amb tipus més complexos. Per exemple, el codi següent dona el nom vi per a un vector de nombres enters i el nom pi per a una parella que conté dos nombres enters.

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

Macros

Una altra manera d'escurçar el codi és definir **macros**. Una macro significa que hi ha determinades cadenes de text que el codi es canviarà abans de la compilació. En C++, les macros es defineixen mitjançant la paraula clau #define.

Per exemple, podem definir les macros següents:

```
#define F primer
#define S segon
#define PB push_back
#define MP make_pair
```

Després d'això, el codi

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

esdevé:

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

Una macro també pot tenir paràmetres, la qual cosa fa possible escurçar bucles i altres estructures. Per exemple, podem definir la macro següent:

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

Després d'això, el codi

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

esdevé:

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

```
}
```

De vegades, les macros provoquen errors que poden ser difícils per detectar. Per exemple, considereu la macro següent que calcula el quadrat d'un nombre:

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

Aquesta macro *no* sempre funciona com s'esperava. Per exemple, el codi

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

correspon al codi

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

Una versió millor de la macro és la següent:

```
#definir SQ(a) (a)*(a)
```

Ara el codi

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

correspon al codi

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

1.5 Matemàtiques

Les matemàtiques tenen un paper important en la competició programació, i no és possible arribar a ser un programador competitiu exitós sense tenir bones habilitats matemàtiques. En aquesta secció es discuteixen alguns conceptes i fórmules matemàtiques que són necessàries més endavant al llibre.

Fórmules de suma

Cada suma de la forma

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

on k és un nombre enter positiu, té una fórmula tancada que és un polinomi de grau $k + 1$. Per exemple¹,

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

¹ Fins i tot hi ha una fórmula general per a aquestes sumes, anomenada **fórmula de Faulhaber**, però és massa complexa per a presentar-la aquí.

i

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

Una **progressió aritmètica** és una seqüència de nombres on la diferència entre dos nombres consecutius és una constant. Per exemple,

$$3, 7, 11, 15$$

és una progressió aritmètica amb constant 4. Es pot calcular la suma d'una progressió aritmètica fent servir la fórmula

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

on a és el primer nombre, b és l'últim nombre i n és la quantitat de nombres. Per exemple,

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

La fórmula es basa en el fet que la suma consta de n nombres i el valor de cada nombre és $(a+b)/2$ de mitjana.

Una **progressió geomètrica** és una seqüència de nombres on el rati entre dos consecutius qualsevol nombres és una constant. Per exemple,

$$3, 6, 12, 24$$

és una progressió geomètrica amb constant 2. Es pot calcular la suma d'una progressió geomètrica fent servir la fórmula

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

on a és el primer nombre, b és l'últim nombre i el la rati entre nombres consecutius és k . Per exemple,

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

Aquesta fórmula es pot derivar de la següent manera. Sigui

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

Quan multipliquem els dos costats per k , obtenim

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

i resolent l'equació

$$kS - S = bk - a$$

ens dona la fórmula.

Un cas especial de la suma d'una progressió geomètrica és la fórmula

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

Una **suma harmònica** és una suma de la forma

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

Un límit superior per a una suma harmònica és $\log_2(n) + 1$. En concret, podem modificar cada terme $1/k$ de manera que es converteixi en la potència més propera de dos que no superi k . Per exemple, quan $n = 6$, podem estimar la suma de la següent manera:

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

Aquest límit superior consta de $\log_2(n) + 1$ parts (1 , $2 \cdot 1/2$, $4 \cdot 1/4$, etc.), i el valor de cada part és com a màxim 1 .

Teoria de conjunts

Un **conjunt** és una col·lecció d'elements. Per exemple, el conjunt

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

conté els elements 2 , 4 i 7 . El símbol \emptyset denota un conjunt buit, i $|S|$ denota la mida d'un conjunt S , és a dir, el nombre d'elements del conjunt. Per exemple, en el conjunt anterior, $|X| = 3$.

Si un conjunt S conté un element x , escrivim $x \in S$, i en cas contrari escrivim $x \notin S$. Per exemple, en el conjunt anterior

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

Es poden construir nous conjunts mitjançant operacions de conjunts:

- La **intersecció** $A \cap B$ és el conjunt dels elements que estan tant en A i en B . Per exemple, si $A = \{1, 2, 5\}$ i $B = \{2, 4\}$, llavors $A \cap B = \{2\}$.
- La **unión** $A \cup B$ és el conjunt dels elements que estan en A o en B o en tots dos. Per exemple, si $A = \{3, 7\}$ i $B = \{2, 3, 8\}$, llavors $A \cup B = \{2, 3, 7, 8\}$.
- El **complement** \bar{A} és el conjunt dels elements que no estan en A . La interpretació d'un complement depèn de el **conjunt universal**, que és el conjunt que conté tots els elements possibles. Per exemple, si $A = \{1, 2, 5, 7\}$ i el conjunt universal és $\{1, 2, \dots, 10\}$, aleshores $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- La **diferència** $A \setminus B = A \cap \bar{B}$ és el conjunt dels elements que estan en A però no en B . Tingueu en compte que B pot contenir elements que no es troben a A . Per exemple, si $A = \{2, 3, 7, 8\}$ i $B = \{3, 5, 8\}$, aleshores $A \setminus B = \{2, 7\}$.

Si cada element de A també pertany a S , diem que A és un **subconjunt** de S , i ho denotem $A \subset S$. Un conjunt S sempre té $2^{|S|}$ subconjunts, inclòs el conjunt buit. Per exemple, els subconjunts del conjunt $\{2, 4, 7\}$ són

$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}$ i $\{2, 4, 7\}$.

Alguns conjunts fets servir sovint són \mathbb{N} (nombres naturals), \mathbb{Z} (nombres enters), \mathbb{Q} (nombres racionals) i \mathbb{R} (nombres reals). El conjunt \mathbb{N} es pot definir de dues maneres, segons la situació: o $\mathbb{N} = \{0, 1, 2, \dots\}$ o $\mathbb{N} = \{1, 2, 3, \dots\}$.

També podem construir un conjunt utilitzant una expressió de la forma

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

on $f(n)$ és una funció. Aquest conjunt conté tots els elements de la forma $f(n)$, on n és un element de S . Per exemple, el conjunt

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

conté tots els nombres enters parells.

Lògica

El valor d'una expressió lògica és **cert** (1) o **fals** (0). Les operacions lògiques més importants són \neg (**negació**), \wedge (**conjunció**), \vee (**disjunció**), \Rightarrow (**implicació**) i \Leftrightarrow (**equivalència**). La taula següent mostra el significat d'aquestes operacions:

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

L'expressió $\neg A$ té el valor oposat de A . L'expressió $A \wedge B$ és certa si A i B són certes, i l'expressió $A \vee B$ és certa si A o B o tots dos són certes. L'expressió $A \Rightarrow B$ és certa si sempre que A és cert, B també ho és. L'expressió $A \Leftrightarrow B$ és certa quan A i B són les dues certes o les dues falses.

Un **predicat** és una expressió que és certa o falsa en funció dels seus paràmetres. Els predicats solen indicar-se amb majúscules. Per exemple, considerem el predicat $P(x)$ que és cert exactament quan x és un nombre primer. Amb aquesta definició, $P(7)$ és cert però $P(8)$ és fals.

Un **quantificador** connecta una expressió lògica als elements d'un conjunt. Els quantificadors més importants són \forall (**per a tots**) i \exists (**existeix un**). Per exemple,

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

significa que per a cada element x del conjunt, existeix un element y al conjunt que compleix que y és més petit que x . Això és cert en el conjunt de nombres enters, però fals en el conjunt dels nombres naturals.

Utilitzant la notació descrita anteriorment, podem expressar molts tipus de proposicions lògiques. Per exemple,

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

significa que si un nombre x és més gran que 1 i no és un nombre primer, aleshores hi ha els nombres a i b que són més grans que 1 i el producte dels quals és x . Aquesta proposició és certa en el conjunt dels nombres enters.

Funcions

La funció $\lfloor x \rfloor$ arrodoneix (cap avall) el nombre x fins a un nombre enter, i la funció $\lceil x \rceil$ arrodoneix (cap amunt) el nombre x fins a un nombre enter. Per exemple,

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

Les funcions $\min(x_1, x_2, \dots, x_n)$ i $\max(x_1, x_2, \dots, x_n)$ donen el valor més petit i més gran dels valors x_1, x_2, \dots, x_n . Per exemple,

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

El **factorial** $n!$ es pot definir

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

o, recursivament,

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

Els **nombres de Fibonacci** sorgeixen en moltes situacions. Es poden definir recursivament de la següent manera:

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

Els primers nombres de Fibonacci són

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

També hi ha una fórmula tancada per calcular els nombres de Fibonacci, que de vegades s'anomena **fórmula de Binet**:

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

Logaritmes

El **logaritme** d'un nombre x es denota per $\log_k(x)$, on k és la base del logaritme. Segons la definició, $\log_k(x) = a$ exactament quan $k^a = x$.

Una propietat útil dels logaritmes és que $\log_k(x)$ és igual al nombre de vegades hem de dividir x per k abans d'arribar el nombre 1. Per exemple, $\log_2(32) = 5$ perquè calen 5 divisions per 2:

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

Els logaritmes s'utilitzen sovint en l'anàlisi de algorismes, perquè molts algorismes eficients redueixen alguna cosa a la meitat a cada pas. Per tant, podem estimar l'eficiència d'aquests algorismes fent servir logaritmes.

El logaritme d'un producte és

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

i en conseqüència,

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

A més, el logaritme d'un quocient és

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

Una altra fórmula útil és

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

i amb això, és possible calcular logaritmes a qualsevol base si hi ha una manera de calcular logaritmes en una base fixa.

El **logaritme natural** $\ln(x)$ d'un nombre x és el logaritme la base del qual és $e \approx 2.71828$. Una altra propietat dels logaritmes és que el nombre de dígits d'un enter x en base b és $\lfloor \log_b(x) + 1 \rfloor$. Per exemple, la representació de 123 en base 2 és 1111011 i $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Concursos i recursos

IOI

L'Olimpíada Internacional d'Informàtica (IOI, International Olympiad in Informatics) és un concurs de programació anual per a alumnes de secundària. Cada país pot enviar un equip de quatre alumnes al concurs. Normalment hi ha uns 300 participants de 80 països.

L'IOI consta de dos concursos de cinc hores de durada. En ambdós concursos, es demana als participants resoldre tres tasques algorísmiques de diferent dificultat. Les tasques es divideixen en subtasques, cadascuna dels quals té una puntuació assignada. Encara que els concursants estiguin dividits en equips, competeixen com a individus.

El pla d'estudis de l'IOI [41] regula els temes que poden aparèixer a les tasques IOI. Gairebé tots els temes del temari de l'IOI són coberts per aquest llibre.

Els participants de l'IOI es seleccionen mitjançant concursos nacionals. Abans de l'IOI, s'organitzen molts concursos regionals, com l'Olimpíada Bàltica d'Informàtica (BOI), Olimpíada Centro-Europea d'Informàtica (CEOI) i l'Olimpíada d'Informàtica Àsia-Pacífic (APIO).

Alguns països organitzen concursos de pràctiques on-line per als futurs participants de l'IOI, com el Concurs Obert d'Informàtica de Croàcia [11] i l'Olimpíada d'Informàtica dels EUA [68]. A més, hi ha una gran col·lecció de problemes dels concursos polonesos disponibles on-line línia [60].

ICPC

El Concurs Internacional de Programació Col·legial (ICPC, International Collegiate Programming Contest) és un concurs de programació anual per a estudiants universitaris. Cada equip del concurs està format per tres estudiants, i a diferència de l'IOI, els alumnes treballen conjuntament; només hi ha un ordinador disponible per a cada equip.

L'ICPC consta de diverses fases, i només els millors equips estan convidats a la fase final (World Finals). Tot i que hi ha desenes de milers de participants al concurs, només hi ha un petit nombre² d'espais per equips, de manera que fins i tot avançar a la final ja és un gran èxit en algunes regions.

En cada concurs de l'ICPC, els equips disposen de cinc hores de temps per a resoldre uns deu problemes algorísmics. La solució d'un problema només s'accepta si es resolen tots els casos de prova de manera eficient. Durant el concurs, els competidors poden veure els resultats d'altres equips, però durant l'última hora el marcador està congelat i no és possible veure els resultats dels últims enviaments.

Els temes que poden aparèixer a l'ICPC no estan tan bé especificats com els de l'IOI. En tot cas, és evident que calen més coneixements a l'ICPC, sobretot més habilitats matemàtiques.

Concursos on-line

També hi ha molts concursos on-line oberts a tothom. En l'actualitat, la pàgina web amb més concursos actius és Codeforces, que organitza concursos de forma setmanal. A Codeforces, els participants es divideixen en dues divisions: els principiants competeixen a la Div2 i els programadors més experimentats a la Div1. Altres llocs de concurs inclouen AtCoder, CS Academy, HackerRank i Topcoder.

Algunes empreses organitzen concursos on-line amb finals presencials. Exemples d'aquests concursos són Facebook Hacker Cup, Google Code Jam i Yandex.Algorithm. Per descomptat, les empreses també utilitzen aquests concursos per a reclutar programadors: tenir un bon resultat en un concurs és una bona manera de demostrar les teves habilitats.

Llibres

Existeixen alguns llibres (a més d'aquest llibre) que es centren en la programació competitiva i la resolució de problemes algorísmics:

- S. S. Skiena i M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim i F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]

²El nombre exacte d'equips a la fase final varia d'un any a un altre; el 2017, hi havia 133 equips.

- K. Diks et al.: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

Els dos primers llibres estan pensats per a principiants, mentre que l'últim llibre conté material avançat.

Per descomptat, els llibres generals d'algorismia també són adequats per als programadors competitius. Alguns llibres populars són:

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

Capítol 2

Complexitat temporal

L'eficiència dels algorismes és important en la programació competitiva. Normalment, és fàcil dissenyar un algorisme que resol el problema lentament, però el veritable repte és inventar un algorisme que sigui ràpid. Si l'algorisme és massa lent només obtindrem punts parcials o cap punt.

La **complexitat temporal** d'un algorisme és el temps, o nombre d'operacions, que l'algorisme necessita per a resoldre entrades. La idea és representar l'eficiència com a una funció de la mida de l'entrada. Quan calculem el cost d'un algorisme podem esbrinar si serà prou ràpid sense implementar-lo.

2.1 Regles de càlcul

La complexitat temporal d'un algorisme es denota per $O(\dots)$ on els punts suspensius representen alguna funció.¹ Normalment, la variable n denota la mida de l'entrada. Per exemple, si l'entrada és una matriu de nombres, és usual prendre n com la mida de la matriu, i si l'entrada és una cadena de text, n serà la mida de la cadena.

Bucles

Sovint els algorismes són lents perquè contenen molts bucles que treballen amb l'entrada. Com més bucles niats (bucles dintre d'altres bucles) contingui l'algorisme, més lent és. Si hi ha k bucles niats, i cadascun d'ells fa n iteracions, la complexitat temporal és $O(n^k)$.

Per exemple, la complexitat temporal del codi següent és $O(n)$:

```
for (int i = 1; i <= n; i++) {  
    // codi  $O(1)$   
}
```

¹Afegit: la complexitat temporal d'un algorisme mesura com de ràpid creix el nombre d'operacions que l'algorisme executa com més fem créixer la mida de l'entrada. Formalment, la notació $O(f(n))$ és el conjunt de funcions que creixen com a molt tan ràpid com $f(n)$, llevat de factors constants: $O(f(n)) = \{g(n) | \exists n_0, c \forall n \geq n_0 \ g(n) \leq c \cdot f(n)\}$.

I la complexitat temporal del codi següent és $O(n^2)$:

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

Ordre de magnitud

La complexitat temporal no ens diu el nombre exacte de vegades que s'executa el codi dintre d'el bucle, sinó l'ordre de magnitud. En els exemples següents, el codi dins del bucle s'executa $3n$, $n + 5$ i $\lceil n/2 \rceil$ vegades, però tots ells tenen complexitat de $O(n)$.

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

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

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

En aquest altre exemple, la complexitat temporal del codi següent és $O(n^2)$:

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

Fases

Si l'algorisme consta de fases consecutives, la complexitat temporal és la complexitat temporal més gran de les fases. El motiu d'això és que la fase més lenta esdevé el coll d'ampolla del codi.

Per exemple, el següent codi consta de tres fases amb complexitats temporals $O(n)$, $O(n^2)$ i $O(n)$. Així, la complexitat total del temps és $O(n^2)$.

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

```

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

```

Diverses variables

De vegades, la complexitat temporal depèn de varis factors. En aquest cas, podem expressar la complexitat temporal com una fórmula de vàries variables.

Per exemple, la complexitat temporal del el codi següent és $O(nm)$:

```

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

```

Recursió

La complexitat temporal d'una funció recursiva depèn del nombre de vegades que es crida la funció multiplicat per la complexitat temporal d'una única crida.

Per exemple, considereu la funció següent:

```

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

```

La crida $f(n)$ provoca n crides a funcions, i la complexitat temporal de cadascuna d'aquestes crides (sense comptar la crida recursiva) és $O(1)$. Així, la complexitat total del temps és $O(n)$.

Com a altre exemple, considereu la funció següent:

```

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

```

En aquest cas, cada crida a la funció genera dues altres crides, excepte quan $n = 1$. Vegem què passa quan es crida g amb el paràmetre n . La taula següent mostra el nombre de crides produït per aquesta crida original:

crida de funció	nombre de crides
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

En base a això, la complexitat temporal és

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

2.2 Clases de complexitat

El llistat següent conté les complexitats temporals més comuns dels algorismes:

$O(1)$ El temps d'execució d'un algorisme de **temps constant** no depèn de la mida d'entrada. Un algorisme típic de temps constant és una fórmula que calcula la resposta directament.

$O(\log n)$ Un algorisme **logarítmic** sovint divideix cada entrada per la meitat a cada pas. El seu cost és logarítmic, perquè $\log_2 n$ és igual al nombre de vegades que és necessari dividir n per 2 fins obtenir 1.

$O(\sqrt{n})$ Un **algorisme d'arrel quadrada** és més lent que $O(\log n)$ però més ràpid que $O(n)$. Una propietat especial de les arrels quadrades és que $\sqrt{n} = n/\sqrt{n}$, de manera que l'arrel quadrada \sqrt{n} està, en certa manera, a la meitat (geomètrica) de l'entrada.

$O(n)$ Un algorisme **linial** és aquell que recorre l'entrada un nombre constant de vegades. Sovint aquesta és la millor complexitat temporal possible en els concursos, perquè normalment cal accedir a cada element de l'entrada com a mínim una vegada abans de produir la resposta.

$O(n \log n)$ Aquest cost sovint indica que l'algorisme ordena l'entrada, perquè aquest és el cost dels algorismes d'ordenació eficients. Una altra possibilitat és que l'algorisme utilitzi una estructura de dades on cada operació triga $O(\log n)$ temps.

$O(n^2)$ Un algorisme **quadràtic** sovint conté dos bucles niats un dintre de l'altre. És possible passar per totes les parelles d'elements de l'entrada en temps $O(n^2)$.

$O(n^3)$ Un algorisme **cúbic** sovint conté tres bucles niats. És possible passar per totes les tripletes d'elements de l'entrada en temps $O(n^3)$.

$O(2^n)$ Aquesta complexitat de temps sovint indica que l'algorisme itera per tot els subconjunts dels elements d'entrada. Per exemple, els subconjunts de $\{1, 2, 3\}$ són \emptyset , $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ i $\{1, 2, 3\}$.

$O(n!)$ Aquesta complexitat de temps sovint indica l'algorisme itera per totes les permutacions dels elements d'entrada. Per exemple, les permutacions de $\{1, 2, 3\}$ són $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ i $(3, 2, 1)$.

Un algorisme és **polinòmic** quan el seu cost és $O(n^k)$ per alguna constant k . Tots els costos anteriors, exceptuant $O(2^n)$ i $O(n!)$, són polinòmics. A la pràctica, la constant k sol ser petita, i sovint s'associen costos polinòmics amb que l'algorisme és *eficient*.

La majoria dels algorismes d'aquest llibre són polinòmics. Tot i així, hi ha molts problemes importants per als quals no es coneix cap algorisme polinòmic, és a dir, ningú sap com resoldre'ls de manera eficient. Els problemes **NP-hard** són un conjunt important de problemes, per als quals no hi ha cap algorisme polinòmic conegut².

2.3 Estimació de l'eficiència

Quan calculem la complexitat temporal d'un algorisme podem comprovar, abans d'implementar-lo, si és prou eficient per al problema. El punt de partida per les estimacions és el fet que un ordinador modern pot realitzar alguns centenars de milions d'operacions en un segon.

Per exemple, considerem un problema on el límit de temps és d'un segon i la mida d'entrada és $n = 10^5$. Si la complexitat temporal és $O(n^2)$, l'algorisme realitzarà unes $(10^5)^2 = 10^{10}$ operacions. Això hauria de trigar almenys unes desenes de segons, de manera que l'algorisme probablement sigui massa lent per resoldre el problema.

D'altra banda, donada la mida de l'entrada, podem intentar *endevinar* la complexitat de l'algorisme que hem de programar per a resoldre el problema. La taula següent conté algunes estimacions útils suposant un límit de temps d'un segon.³

mida d'entrada	complexitat esperada
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ o $O(n)$
n és gran	$O(1)$ o $O(\log n)$

Per exemple, si la mida de l'entrada és $n = 10^5$, segurament la solució del problema sigui un algorisme $O(n)$ o $O(n \log n)$. Aquesta informació facilita el disseny de l'algorisme, perquè descarta enfocaments que donarien un algorisme amb un cost massa gran.

²Un llibre clàssic sobre el tema és de M. R. Garey i D. S. Johnson *Informàtica i intractabilitat: una guia per a la teoria de NP-Complexitat* [28].

³No totes les operacions costen el mateix: per exemple, les operacions d'entrada i sortida són més costoses que les operacions aritmètiques.

Tot i així, és important recordar que la complexitat temporal és només una estimació de l'eficiència, perquè amaga els *factors constants*. Per exemple, un algorisme que s'executa en $O(n)$ temps potser fa $n/2$ o $5n$ operacions. Això és un factor important de cara al temps real que farà servir l'algorisme.

2.4 Suma màxima d'un subvector

Sovint hi ha diversos algorismes que poden resoldre un problema i que tenen distintes complexitats temporals. Aquesta secció tracta d'un problema clàssic que té una solució $O(n^3)$ senzilla. Tanmateix, dissenyant un algorisme millor, és possible resoldre el problema en temps $O(n^2)$ temps o fins i tot en temps $O(n)$.

Donat un vector de n nombres, la nostra tasca és calcular el **suma màxima d'un subvector**, és a dir, la suma més gran possible d'una seqüència de valors consecutius del vector⁴. El problema és interessant quan hi ha valors negatius en el vector. Per exemple, en el vector

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

el subvector següent té suma màxima 10:

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

Suposem que es permet triar un subvector buit, de manera que la suma màxima és sempre com a mínim 0.

Algorisme 1

Una manera senzilla de resoldre el problema és passar per tots els subvectors possibles, calcular la suma de valors de cada subvector i mantenir la suma màxima. El codi següent implementa aquest algorisme:

```
int millor = 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 += v[k];
        }
        millor = max(millor, suma);
    }
}
cout << millor << "\n";
```

Les variables a i b contenen el primer i el darrer índex del subvector, i la suma de valors es calcula a la variable suma . La variable millor conté la suma màxima trobada durant la cerca.

⁴El llibre *Programming Pearls* [8] de J. Bentley va popularitzar aquest problema.

La complexitat temporal de l'algorisme és $O(n^3)$, perquè consta de tres bucles niats que recorren l'entrada.

Algorisme 2

És fàcil fer que l'algorisme 1 sigui més eficient eliminant-ne un bucle. Això és possible si calculem la suma a la vegada que moguem el darrer índex del subvector. El resultat és el codi següent:

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

Després d'aquest canvi, la complexitat temporal és $O(n^2)$.

Algorisme 3

Sorprenentment, és possible⁵ resoldre el problema en temps $O(n)$, que significa que n'hi ha prou amb un sol bucle. La idea és calcular, per a cada posició del vector, la suma màxima d'un subvector que acaba en aquesta posició. Després d'això, la resposta al problema és el màxim d'aquestes sumes.

Considereu el subproblema de trobar la suma màxima del subvector que acaba a la posició k . Hi ha dues possibilitats:

1. El subvector només conté l'element a la posició k .
2. El subvector consisteix en un subvector que acaba a la posició $k - 1$, seguit de l'element a la posició k .

En aquest darrer cas, ja que volem trobar un subvector amb suma màxima, el subvector que acaba a la posició $k - 1$ també ha de tenir suma màxima. Així, podem resoldre el problema de manera eficient calculant la suma màxima del subvector per a cada posició final d'esquerra a dreta.

El codi següent implementa l'algorisme:

```
int millor = 0, suma = 0;
for (int k = 0; k < n; k++) {
    suma = max(v[k], suma+v[k]);
    millor = max(millor, suma);
}
cout << millor << "\n";
```

⁵En [8], aquest algorisme de temps lineal s'atribueix a J. B. Kadane, i l'algorisme de vegades s'anomena **algorisme de Kadane**.

L'algorisme només conté un bucle que passa per l'entrada, per tant, la complexitat temporal és $O(n)$. Aquesta és també la millor complexitat temporal possible, perquè qualsevol algorisme per aquest problema ha d'examinar tots els elements del vector almenys una vegada.

Comparació d'eficiència

És interessant estudiar com d'eficients són els algorismes a la pràctica. La taula següent mostra els temps de funcionament dels algorismes anteriors per a diferents valors de n en un ordinador modern.

En cada prova, vam generar l'entrada aleatòriament, i no vam mesurar el temps necessari per llegir l'entrada.

mida del vector n	algorisme 1	algorisme 2	algorisme 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 comparació mostra que tots els algorismes són eficients quan la mida de l'entrada és petita, però les entrades més grans mostren diferències notables entre els temps d'execució dels algorismes. L'algorisme 1 es torna lent quan $n = 10^4$, i l'algorisme 2 es torna lent quan $n = 10^5$. Només l'algorisme 3 pot processar fins i tot les entrades més grans instantàniament.

Capítol 3

Ordenació

El problema de l'**ordenació** és un problema fonamental del disseny d'algoritmes. Molts algorismes eficients fan servir l'ordenació com a subrutina, perquè sovint és més fàcil de processar dades si els elements estan ordenats.

Per exemple, el problema de “té un vector dos elements iguals?” és fàcil de resoldre mitjançant l'ordenació. Si el vector conté dos elements iguals, després d'ordenar-los estaran l'un al costat de l'altre, de manera que és fàcil trobar-los. El problema “quin és l'element més freqüent d'un vector?” es pot resoldre de la mateixa manera.

Hi ha molts algorismes per ordenar, i aquests són també bons exemples de com aplicar diferents tècniques del disseny d'algoritmes. Els algorismes d'ordenació general eficients treballen en temps $O(n \log n)$, i molts algorismes que utilitzen l'ordenació com a subrutina tenen també aquesta complexitat temporal.

3.1 Teoria de l'ordenació

El problema bàsic de l'ordenació és el següent:

Donat un vector que conté n elements, ordena els elements en ordre creixent.

Per exemple, el vector

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

queda de la manera següent després d'ordenar-lo:

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

Algorismes $O(n^2)$

Alguns algorismes senzills per ordenar un vector treballen en temps $O(n^2)$. Aquests algorismes són curts i generalment consten de dos bucles niats. Un

algorisme famós amb complexitat $O(n^2)$ és **l'ordenació amb bombolla (bubble sort)** on els elements del vector es mouen com si fossin “bombolles”.

L'ordenació amb bombolla consta de n rondes. A cada ronda, l'algoritme itera els elements del vector. Sempre que es troben dos elements consecutius que no estan en ordre correcte, l'algoritme els intercanvia. L'algorisme es pot implementar de la següent manera:

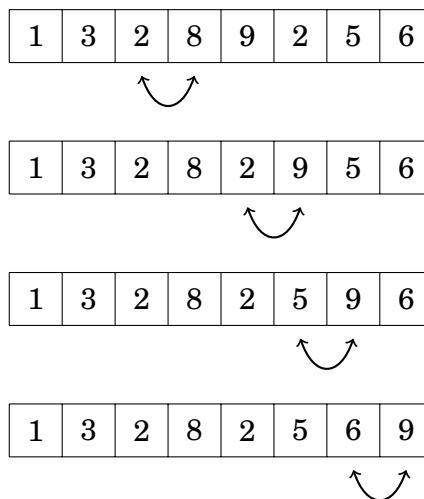
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n-1; j++) {  
        if (v[j] > v[j+1]) {  
            swap(v[j], v[j+1]);  
        }  
    }  
}
```

Després de la primera ronda de l'algorisme, l'element més gran estarà en la posició correcta, i en general, després de k rondes, els k elements més grans estaran en les posicions correctes. Així, després de n rondes, el vector quedarà ordenat.

Per exemple, en el vector

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

la primera ronda d'ordenació amb bombolla intercanvia els elements com segueix:



Inversions

L'ordenació amb bombolla és un exemple d'algorisme d'ordenació que sempre intercanvia elements *consecutius* del vector. La complexitat temporal d'aquest algorisme és $O(n^2)$, donat que aquest és el nombre de comparacions que es fan. (I, en el pitjor cas, és el nombre de swaps).

Un concepte útil a l'hora d'analitzar l'ordenació algorismes és el concepte d'**inversió**, que són els parells d'elements situats en l'ordre incorrecte, és a dir, els parells $(v[a], v[b])$ tal que $a < b$ i $v[a] > v[b]$. Per exemple, el vector

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

té tres inversions: (6,3), (6,5) i (9,8). El nombre d'inversions indica quanta feina es necessària per ordenar el vector. Un vector està completament ordenat quan no hi ha inversions. D'altra banda, si tots els elements del vector estan en ordre invers, el nombre d'inversions és el més gran possible:

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

Intercanviar un parell d'elements consecutius en ordre incorrecte elimina exactament una inversió del vector. Per tant, si un algorisme d'ordenació només intercanvia elements consecutius, cada intercanvi elimina com a màxim una inversió i la complexitat temporal de l'algorisme és almenys $O(n^2)$.

Algorismes $O(n \log n)$

És possible ordenar una vector de manera eficient en temps $O(n \log n)$ fent servir algorismes que no es limiten a intercanviar elements consecutius. Un d'aquests algorismes és **merge sort**¹, que es basa en la recursivitat.

El merge sort ordena un subvector $v[a \dots b]$ de la següent manera:

1. Si $a = b$, no feu res, perquè el subvector ja està ordenat.
2. Calcula la posició de l'element central: $k = \lfloor (a + b)/2 \rfloor$.
3. Ordena recursivament el subvector $v[a \dots k]$.
4. Ordena recursivament el subvector $v[k + 1 \dots b]$.
5. Fusiona (*merge*) els subvectors ordenats $v[a \dots k]$ i $v[k + 1 \dots b]$ en un subvector ordenat $v[a \dots b]$.

El merge sort és un algorisme eficient, perquè cada pas redueix a la meitat la mida del subvector. La recursivitat consisteix en $O(\log n)$ nivells, i cada nivell triga temps $O(n)$. És possible fusionar i ordenar els subvectors $v[a \dots k]$ i $v[k + 1 \dots b]$ en temps linial perquè ja estan ordenats.

Per exemple, considerem el vector següent:

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

Dividim el vector en dos subvectors:

1	3	6	2
8	2	5	9

Ordenem els subvectors de manera recursiva:

1	2	3	6
2	5	8	9

Finalment, l'algorisme fusiona els subvectors ordenats i crea el vector final:

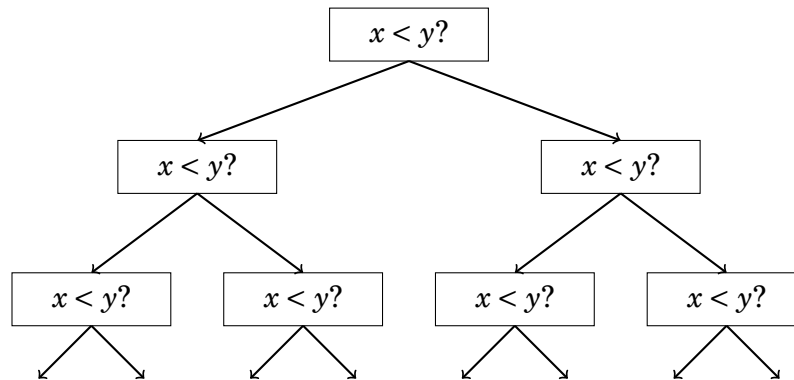
1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

¹Segons [47], el merge sort va ser inventat per J. von Neumann el 1945.

Límit inferior per l'ordenació

És possible ordenar més ràpidament que en temps $O(n \log n)$? Resulta que això *no* és possible quan fem servir algorismes d'ordenació que es basen en comparar els elements d'un vector.

Aquest límit inferior $O(n \log n)$ es demostra si hom considera l'ordenació com un procés on cada comparació de dos elements proporciona més informació sobre el contingut del vector original. El procés crea el següent arbre:



Aquí “ $x < y$?” significa que comparem alguns elements x i y . Si $x < y$, el procés continua cap a l'esquerra, i en cas contrari cap a la dreta. Els resultats d'aquest procés són les $n!$ maneres en que els n elements poden aparèixer en el vector. Per aquest motiu, l'alçada de l'arbre ha de ser almenys

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

Obtenim un límit inferior per a aquesta suma escollint els últims $n/2$ elements i canviant el valor de cada element per $\log_2(n/2)$. Això dóna una estimació

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

de manera que l'alçada de l'arbre i el mínim nombre possible de passos en un algorisme d'ordenació és $O(n \log n)$.

Ordenació per comptatge

El límit inferior $n \log n$ no s'aplica a algorismes que no comparen els elements del vector però que utilitzen altra informació. Un exemple d'aquests algorismes és l'ordenació per comptatge (**counting sort**) que ordena un vector en temps $O(n)$ assumint que tots els elements del vector són enters entre $0 \dots c$ i $c = O(n)$.

L'algorisme crea un vector addicional de mida c , els índexos del qual són els elements del vector original. L'algoritme itera a través del vector original i calcula quantes vegades cada element apareix al vector.

Per exemple, el vector

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

li correspon el següent vector de comptatge:

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

Per exemple, el valor a la posició 3 del vector de comptatge és 2, perquè l'element 3 apareix 2 vegades al vector original.

La construcció del vector de comptatge triga temps $O(n)$. Després d'això, el vector ordenat es crea en temps $O(n)$ perquè recuperem el nombre d'ocurrències de cada element en el vector de comptatge. Per tant, la complexitat temporal total és $O(n)$.

L'ordenació per comptatge és un algorisme molt eficient però només es pot fer servir quan el valor c és prou petit, ja que en cas contrari no podem fer servir els elements del vector com a índexos del vector de comptatge.

3.2 Ordenació en C++

Gairebé mai és bona idea fer servir algorismes d'ordenació casolans en un concurs, perquè els llenguatges de programació ja venen amb molt bones implementacions. Per exemple, la biblioteca estàndard de C++ conté la funció `sort` que es pot fer servir per a ordenar vectors i altres estructures de dades.

Hi ha molts avantatges en fer servir les funcions de les biblioteques. Primer, estalvia temps perquè no cal implementar la funció. Segon, la implementació de la biblioteca és certament correcta i eficient: no és probable que la teva funció de classificació casolana sigui millor.

En aquesta secció veurem com fer servir la funció C++ `sort`. El codi següent ordena un vector en ordre creixent:

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

Després de l'ordenació, el contingut del vector és `[2,3,3,4,5,5,8]`. Els elements s'ordenen per defecte en ordre creixent, però també és possible ordenar-los en ordre invers:

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

També és possible ordenar un array ordinari de C:

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

El codi següent ordena la cadena `s`:

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

Ordenar una cadena significa que ordenem tots els seus caràcters. Per exemple, la cadena “monkey” es converteix en “ekmnoy”.

Operadors de comparació

La funció `sort` requereix que es defineix un **operador de comparació** per al tipus de dades dels elements a ordenar. Quan ordenem, es farà servir aquest operador sempre que sigui necessari esbrinar l'ordre de dos elements.

La majoria dels tipus de dades C++ tenen un operador de comparació definit per defecte, i els elements d'aquests tipus es poden ordenar automàticament. Per exemple, els nombres s'ordenen segons els seus valors i les cadenes estan ordenades per ordre alfabètic.

Els parells (`pair`) s'ordenen en funció dels seus primers elements (`first`). Si els primers elements dels dos parells són iguals, aleshores s'ordenen segons els segons elements (`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());
```

Després d'això, l'ordre de les parelles és (1,2), (1,5) i (2,3).

De manera semblant, les tuples (`tuple`) s'ordenen pel primer element, seguit pel segon element en cas d'empat, el tercer, 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());
```

Després d'això, l'ordre de les tuples és (1,5,3), (2,1,3) i (2,1,4).

Aquesta idea de les tuples també s'extén als vectors.

Estructures definides per l'usuari

Les estructures definides per l'usuari no tenen un operador de comparació definit per defecte. Una manera de fer-ho és definir l'operador a dintre de la estructura de dades com a funció `operator<`, el paràmetre del qual és un altre element del mateix tipus. L'operador hauria de retornar `true` si l'element és més petit que el paràmetre, i `false` en cas contrari.

Per exemple, l'estructura `P` següent conté les coordenades `x` i `y` d'un punt. L'operador de comparació es defineix de manera que els punts s'ordenen per la coordenada `x` i, en cas d'empat, per la coordenada `y`.

²Tingueu en compte que en alguns compiladors antics, és necessari fer servir la funció `make_tuple` en lloc de les claus (per exemple, `make_tuple(2,1,4)` en lloc de `{2,1,4}`).


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

Funcions de comparació

També és possible donar una **funció de comparació** externa (callback) a la funció `sort`. Per exemple, la següent funció de comparació `comp` ordena les cadenes primer per longitud i segon per ordre alfabètic:

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

Ara podem ordenar un vector de cadenes així:

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

N. del T.: També podem fer servir expressions lambda:

```
sort(v.begin(), v.end(), [](const string& a, const string& b) {
    return (a.size() != b.size()) ? (a.size() < b.size()) : (a < b);
});
```

3.3 Cerca binària

Un mètode general per cercar un element en una vector és fer servir un bucle `for` que iteri els elements del vector. Per exemple, el codi següent cerca un element x en el vector v :

```
for (int i = 0; i < n; i++) {
    if (v[i] == x) {
        // trobem x a la posicio i
    }
}
```

La complexitat temporal d'aquest codi és $O(n)$, perquè en el pitjor dels casos, cal comprovar tots els elements del vector. Si l'ordre dels elements és arbitrari, aquesta és també la millor solució, perquè no tenim informació addicional per saber en quin lloc del vector hem de cercar l'element x .

Tanmateix, si la vector està *ordenat*, la situació és diferent. En aquest cas és possible realitzar el cerca molt més ràpid, perquè l'ordre dels elements del vector guia la cerca. El següent algorisme de **cerca binària** cerca eficientment un element en un vector ordenat en temps $O(\log n)$.

Mètode 1

La forma habitual d'implementar la cerca binària s'assembla a buscar una paraula en un diccionari. La cerca manté una regió activa al vector, que inicialment conté tots els elements del vector. Aleshores, es fa una sèrie de passos, cadascun dels quals redueix a la meitat la mida de la regió activa.

A cada pas, la cerca comprova l'element central de la regió activa. Si l'element central és l'element objectiu, la cerca acaba. En cas contrari, la cerca continua de forma recursiva a la meitat esquerra o dreta de la regió, en funció del valor de l'element mitjà.

La idea anterior es pot implementar de la següent manera:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (v[k] == x) {
        // trobem x a la posicio k
    }
    if (v[k] > x) b = k-1;
    else a = k+1;
}
```

En aquesta implementació, la regió activa és $a \dots b$, i la regió inicial és $0 \dots n-1$. L'algorisme redueix a la meitat la mida de la regió a cada pas, per tant, la complexitat temporal és $O(\log n)$.

Mètode 2

Un mètode alternatiu per implementar la cerca binària es basa en una manera eficient d'iterar els elements de la vector. La idea és fer salts i reduir la velocitat quan ens acostem a l'element objectiu.

La cerca passa per la vector d'esquerra a dreta, i la longitud inicial del salt és $n/2$. A cada pas, la longitud del salt es reduirà a la meitat: primer $n/4$, després $n/8$, $n/16$, etc., fins que finalment la longitud és 1. Després dels salts, l'element objectiu o bé s'ha trobat o bé sabem que no apareix al vector.

El codi següent implementa la idea anterior:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && v[k+b] <= x) k += b;
}
if (v[k] == x) {
    // trobem x a la posicio x
}
```

```
}
```

Durant la cerca, la variable b conté la longitud actual del salt. La complexitat temporal és també $O(\log n)$, perquè el codi del bucle `while` es realitza com a màxim dues vegades per a cada llargada de salt.

Funcions C++

La biblioteca estàndard de C++ conté les funcions següents que es basen en cerca binària i triguen temps logarítmic:

- `lower_bound` retorna un punter al primer element del vector el valor del qual és almenys x .
- `upper_bound` retorna un punter a primer element del vector el valor del qual és més gran que x .
- `equal_range` retorna els dos punters anteriors.

Les funcions assumeixen que el vector està ordenat. Si aquest primer element no existeix, les funcions retornen un punter a una posició més enllà de l'últim element del vector³. Per exemple, el codi següent esbrina si un vector ordenat conté un element amb el valor x :

```
int k = lower_bound(v.begin(), v.end(), x) - v.begin();
if (k < n && v[k] == x) {
    // trobem x a la posicio k
}
```

El codi següent compta el nombre d'elements amb valor x :

```
auto a = lower_bound(v.begin(), v.end(), x);
auto b = upper_bound(v.begin(), v.end(), x);
cout << b-a << "\n";
```

Utilitzant `equal_range`, el codi queda més curt:

```
auto r = equal_range(v.begin(), v.end(), x);
cout << r.second-r.first << "\n";
```

Trobar la solució més petita

Un ús important de la cerca binària és trobar la posició on canvia el valor d'una funció. Suposem que volem trobar el valor més petit k que és una solució vàlida per a un problema. Ens donen una funció `ok(x)` que retorna `true` si x és vàlida i `false` en cas contrari. A més, sabem que `ok(x)` és `false` quan $x < k$ i `true` quan $x \geq k$. És a dir:

³N. del T: Dit d'altra manera: l'interval semi-obert `[lower_bound(x), upper_bound(x))` assenya-la en quin lloc són, o haurien de ser, els elements amb valor x .

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	f	f	...	f	t	t	...

Ara, el valor de k es pot trobar mitjançant la cerca binària:

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

< La cerca troba el valor més gran de x per al qual $ok(x)$ és false. Així, el següent valor $k = x + 1$ és necessàriament el valor més petit possible per al qual $ok(k)$ és true. La longitud inicial del salt z ha de ser prou gran, per exemple algun valor per al qual sabem per endavant que $ok(z)$ és true.

L'algorisme crida a la funció ok $O(\log z)$ vegades, per tant la complexitat total depèn de la funció ok . Per exemple, si aquesta funció triga temps $O(n)$, la complexitat total és $O(n \log z)$.

Trobar el valor màxim

La cerca binària també es pot fer servir per trobar el valor màxim d'una funció que primer creix i després decreix. La nostra tasca és trobar una posició k tal que

- $f(x) < f(x+1)$ quan $x < k$, i
- $f(x) > f(x+1)$ quan $x \geq k$.

La idea és fer servir la cerca binària per trobar el valor més gran de x tal que $f(x) < f(x+1)$. Això implica que $k = x + 1$ perquè $f(x+1) > f(x+2)$. El codi següent implementa la cerca:

```
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;
```

Tingueu en compte que, a diferència de la cerca binària ordinària, en aquest cas no permetem que els valors consecutius de la funció siguin iguals, donat que no sabríem en quina direcció continuar la cerca.

Capítol 4

Estructures de dades

Una **estructura de dades** és una manera d'emmagatzemar dades a la memòria d'un ordinador. És important escollir l'estructura de dades adequada per a un problema, perquè cada estructura de dades té els seus avantatges i inconvenients. La pregunta crucial és: quines operacions són eficients en l'estructura de dades escollida?

Aquest capítol presenta les estructures de dades més importants a la biblioteca estàndard de C++. És molt bona idea fer servir la biblioteca estàndard sempre que sigui possible, perquè ens estalviarà molt de temps. Més endavant parlarem d'estructures de dades més sofisticades no estan disponibles a la biblioteca estàndard.

4.1 Vectors dinàmics

Un **vector dinàmic** és un vector la mida del qual pot canviar durant l'execució del programa. El vector dinàmic més popular en C++ és l'estructura `vector`, que també pot fer-se servir gairebé com un array de C normal.

El codi següent crea un vector buit i hi afegeix tres elements:

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

Després d'això, podem accedir als elements com si fos un array de C normal:

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

La funció `size` retorna el nombre d'elements del vector. El codi següent itera el vector i escriu tots els elements:

```
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```

```
}
```

Una manera més concisa d'iterar un vector és la següent:

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

La funció `back` retorna l'últim element en el vector, i la funció `pop_back` elimina l'últim element:

```
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 codi següent crea un vector de cinc elements:

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

Una altra manera de crear un vector és donar el nombre d'elements i el valor inicial de cada element:

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

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

La implementació interna d'un vector fa servir un array ordinari. Si la mida del vector augmenta i l'espai reservat es massa petit, es crea un nou array i tot els elements es mouen al nou espai. Però això no passa sovint, i la complexitat temporal promig de fer un `push_back` és $O(1)$.

L'estructura cadena (`string`) és un vector dinàmic de caràcters. A més, hi ha una sintaxi especial per a les cadenes que no està disponible en les altres estructures de dades. Les cadenes es poden concatenar fent servir el símbol `+`. La funció `substr(k,x)` retorna la subcadena que comença a la posició *k* i té longitud *x*, i la funció `find(t)` troba la posició de la primera ocurrència d'una subcadena *t*.

El codi següent presenta algunes operacions de cadenes:

```
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 Estructures conjunt

Un **conjunt** és una estructura de dades que manté una col·lecció d'elements. Les operacions bàsiques dels conjunts són inserir, cercar i eliminar.

La biblioteca estàndard de C++ conté dues implementacions: L'estructura `set` es basa en arbres binari equilibrats i les seves operacions triguen $O(\log n)$ en cas pitjor. L'estructura `unordered_set` fa serving hashing, i les seves operacions triguen $O(1)$ en mitjana.

Quina implementació triar és sovint qüestió de gustos. El benefici de l'estructura `set` és que manté l'ordre dels elements i ofereix funcions que no estan disponibles a `unordered_set`. D'altra banda, `unordered_set` pot ser més eficient.

El codi següent crea un conjunt que conté nombres enters, i mostra algunes de les operacions. La funció `insert` afegeix un element al conjunt, la funció `count` retorna el nombre d'ocurrències d'un element del conjunt, i la funció `erase` elimina un element del conjunt.

```
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 conjunt es pot fer servir com un vector, però no es pot accedir als elements fent servir la notació `[]`. El codi següent crea un conjunt, escriu el nombre d'elements, i itera per tots els elements:

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

Una propietat important dels conjunts és que tots els seus elements són *diferents*. Així, la funció `count` sempre retorna 0 (l'element no està al conjunt) o 1 (l'element és al conjunt), i la funció `insert` no s'afegeix mai un element al conjunt si ja hi és. Això es pot veure al codi següent:

```
set<int> s;  
s.insert(5);  
s.insert(5);
```

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

C++ també conté les estructures `multiset` i `unordered_multiset`, que funcionen com `set` i `unordered_set` excepte que poden contenir vàries instàncies d'un element. Per exemple, al codi següent afegeix tres instàncies del número 5 a un multiconjunt:

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

La funció `erase` elimina totes les instàncies d'un element d'un multiconjunt:

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

Sovint només volem eliminar una instància, que podem fer de la següent manera:

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

4.3 Estructures mapa

Un **mapa** és un vector que consisteix de parells clau-valor. Mentre que les claus d'un vector normal sempre són els nombres enters consecutius $0, 1, \dots, n-1$, on n és la mida del vector, les claus d'un mapa poden ser de qualsevol tipus de dades i no tenen perquè ser consecutius.

La biblioteca estàndard de C++ conté dos implementacions de mapa que es corresponen amb les implmentacions de conjunts: l'estructura `map` es basa en un arbre binari equilibrat i accedir als elements triga $O(\log n)$, mentre que l'estructura `unordered_map` fa servir hashing i accedir als elements triga $O(1)$ de mitjana.

El codi següent crea un mapa on les claus són cadenes i els valors són nombres enters:

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

Si es demana el valor d'una clau però el mapa no la conté, la clau s'afegeix automàticament al mapa amb un valor per defecte. Per exemple, en el codi següent, la clau "aybaltu" s'afegeix al mapa amb valor 0.


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

La funció count comprova si el mapa conté una clau:

```
if (m.count("aybabbu")) {  
    // la clau existeix  
}
```

El codi següent escriu totes les claus i valors d'un mapa:

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

4.4 Iteradors i intervals

Moltes funcions de la biblioteca estàndard de C++ operen amb iteradors. Un **iterador** (iterator) és una variable que apunta a un element d'una estructura de dades.

Els iteradors begin i end defineixen un interval que conté tots els elements d'una estructura de dades. L'iterador begin apunta al primer element de l'estructura de dades, i l'iterador end apunta a la posició *després de* l'últim element. Per exemple:

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

Observeu l'asimetria dels iteradors: s.begin() apunta a un element de l'estructura de dades, mentre que s.end() apunta fora de l'estructura de dades. És a dir, l'interval definit pels iteradors és *semi-obert*.

Treballar amb intervals

Els iteradors es fan servir en la biblioteca estàndard de C++ per a passar intervals d'elements en una estructura de dades. Normalment, volem processar tots els elements d'una estructura de dades, i per tant fem servir els iteradors begin i end.

Per exemple, el codi següent ordena un vector fent servir la funció sort, després inverteix l'ordre dels elements fent servir la funció reverse, i finalment barreja l'ordre de els elements fent servir la funció random_shuffle.

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

Aquestes funcions també es poden fer servir amb arrays ordinaris de C. En aquest cas, les funcions reben punters a l'array en lloc d'iteradors:

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

Iteradors de conjunts

Els iteradors es fan servir sovint per a accedir als elements d'un conjunt. El codi següent crea un iterador `it` que assenyalava a l'element més petit d'un conjunt:

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

També es pot escriure així:

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

Per a accedir a l'element que l'iterador assenyalava es fa servir el símbol `*`. Per exemple, el següent codi imprimeix el primer element del conjunt:

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

Els iteradors es poden moure mitjançant els operadors `++` (endavant) i `--` (enrere), és a dir, moure l'iterador a l'element següent o l'element anterior.

El codi següent escriu tots els elements en ordre creixent:

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

El codi següent escriu l'element més gran del conjunt:

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

La funció `find(x)` retorna un iterador que apunta a un element el valor del qual és `x`. Si el conjunt no conté `x`, l'iterador retornat serà `end`.

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

La funció `lower_bound(x)` retorna un iterador al primer element del conjunt el valor del qual és *almenys* `x`, i la funció `upper_bound(x)` retorna un iterador al primer element del conjunt el valor del qual és *més gran que* `x`. En ambdós

casos, si aquest primer element del conjunt no existeix, retornen end.¹ Aquestes funcions no són compatibles amb l'estructura `unordered_set` perquè aquesta no manté l'ordre dels elements.

Per exemple, el codi següent troba l'element més proper 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 codi suposa que el conjunt no està buit, i repassa tots els casos possibles utilitzant un iterador `it`. L'iterador apunta al valor més petit el valor del qual és almenys x . Si `it` és igual a `begin`, l'element corresponent és el més proper a x . Si `it` és igual a `end`, l'element més gran del conjunt és el més proper a x . Si no es compleix cap dels dos casos anteriors, l'element més proper a x és o bé l'element assenyalat per `it` o bé l'element anterior.

4.5 Altres estructures

Conjunt de bits

Un conjunt de bits, o **bitset**, és un vector on cada valor és 0 o 1. El codi següent crea un conjunt de bits amb 10 elements:

```
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
```

L'avantatge de fer servir conjunts de bits és que requereixen menys memòria que els vectors normals, perquè cada element només fa servir un únic bit de memòria. Per exemple, si emmagatzemèssim n bits en un vector de tipus `int`, necessitaríem $32n$ bits de memòria, però en un conjunt de bits només fan falta n bits de memòria. A més, els valors d'un conjunt de bits es poden manipular de

¹N. del T: És a dir, `[lower_bound(x), upper_bound(x))` és l'interval semi-obert que assenyalava als elements amb valor x .

manera eficient fent servir operadors de bits, amb la qual cosa encara és possible optimitzar més alguns algorismes.

El codi següent mostra una altra manera de crear el conjunt de bits anterior:

```
bitset<10> s(string("0010011010")); // de dreta a esquerra
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

La funció count retorna el nombre d'uns en el conjunt de bits:

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

El codi següent mostra exemples d'operacions 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

Una **deque** és un vector dinàmic la mida del qual es pot canviar eficientment a ambdós extrems del vector. Al igual que els vectors, un deque proporciona les funcions push_back i pop_back, però també inclou les funcions push_front i pop_front que no estan disponibles en un vector normal.

Un deque es pot fer servir de la manera següent:

```
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 implementació interna d'un deque és més complexa que el d'un vector, i per aquest motiu, un deque és més lent que un vector. Tot i així, afegir o eliminar elements per qualsevol dels dos extrems triga $O(1)$ de mitjana.

Pila

Una **pila** (stack) és una estructura de dades que proporciona dos operacions que triguen $O(1)$: afegint un element a la part superior, i retirar l'element de la part superior.

El codi següent mostra com fer servir una pila:

```
stack<int> s;
```

```
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Queue

Una **queue** també ofereix dos operacions de temps $O(1)$: afegir un element al final de la cua, i treure el primer element de la cua. Només és possible accedir al primer i al darrer element de la cua.

El codi següent mostra com fer servir una cua:

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

Cua de prioritats

Una **cua de prioritats** manté un conjunt d'elements. S'ofereixen les operacions d'inserció i, depenent del tipus de cua, recuperar o eliminar l'element mínim o màxim (només un dels dos). Inserir i eliminar elements triga $O(\log n)$, mentre que obtenir l'element mínim o màxim triga $O(1)$.

En principi, es podria fer servir un conjunt ordenat per a implementar una cua de prioritats, però l'avantatge de fer servir una cua de prioritats és que els factors constants són més petits. Les cues de prioritats s'implementen fent servir una estructura de heap que és molt més senzilla que els arbres binaris balancejats dels conjunts ordenats.

Per defecte, els elements en una cua de prioritats de C++ s'ordenen en ordre decreixent, i només podem trobar i eliminar l'element més gran de la cua. Per exemple:

```
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 volem crear una cua de prioritat que permeti trobar i eliminar l'element més petit, podem fer el següent:

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

Estructures de dades “policy-based”

El compilador g++ també ofereix algunes estructures de dades que no formen part de la biblioteca estàndard C++. Aquestes estructures s'anomenen *policy-based*. Per a fer-les servir, hem d'afegir les línies següents:

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

Després d'això, podem fer servir una estructura de dades `indexed_set` que és un set que es pot indexar com un vector. La definició per a valors de tipus `int` és la següent:

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

Ara podem crear un conjunt de la manera següent:

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

L'especialitat d'aquest conjunt és que tenim accés als índexs que tindrien els elements en un vector ordenat. La funció `find_by_order` retorna un iterador a l'element que ocupa una posició determinada:

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

I la funció `order_of_key` retorna la posició d'un element donat:

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

Si l'element no apareix al conjunt, obtenim la posició que tindria l'element al conjunt:

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

Ambdues funcions funcionen en temps logarítmic.

4.6 Comparació amb l'ordenació

Sovint és possible resoldre un problema fent servir estructures de dades o ordenació. De vegades hi ha diferències notables en l'eficiència real d'aquests enfocaments, que poden quedar amagats en les seves complexitats temporals.

Considerem un problema on se'ns donen dues llistes A i B , ambdues de n elements. La nostra tasca és calcular el nombre d'elements que pertanyen a les dues llistes. Per exemple, per a les llistes

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

la resposta és 3 perquè els números 2, 5 i 9 pertanyen a les dues llistes.

La solució directa al problema és recórrer tots els parells d'elements en temps $O(n^2)$, però a continuació ens centrarem en els algorismes més eficients.

Algorisme 1

Construïm un conjunt amb els elements que apareixen a A , i després d'això, iterem a través dels elements de B i comprovem si cadascun dels elements també pertanyen a A . Això és eficient perquè els elements de A estan en un conjunt. Utilitzant l'estructura `set`, la complexitat temporal de l'algorisme és $O(n \log n)$.

Algorisme 2

No necessitem mantenir un conjunt ordenat per a A , per tant, en lloc d'un `set` fem servir un `unordered_set`. Aquesta és una manera fàcil de fer l'algorisme més eficient, perquè només hem de canviar l'estructura de dades, sense canviar l'algorisme. La complexitat temporal del nou algorisme és $O(n)$.

Algorisme 3

Ordenem en lloc de fer servir estructures de dades. Primer, ordenem les dues llistes A i B . Després d'això, recorrem les dues llistes alhora i trobem els elements comuns. El cost de l'ordenació és $O(n \log n)$, i la resta de l'algorisme triga $O(n)$, per tant, la complexitat total és $O(n \log n)$.

Comparació d'eficiència

La taula següent mostra l'eficiència els algorismes anteriors quan n varia i els elements de les llistes són nombres enters aleatoris entre $1 \dots 10^9$:

n	Algorisme 1	Algorisme 2	Algorisme 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

Els Algorismes 1 i 2 són iguals excepte que fem servir diferents estructures de conjunt. En aquest problema, aquesta elecció té un efecte important en el temps d'execució, perquè l'Algorisme 2 és de 4 a 5 vegades més ràpid que l'Algorisme 1.

Tanmateix, l'algorisme més eficient és el que fa servir l'ordenació, l'Algorisme 3. Només triga la meitat del temps que triga l'Algorisme 2. Curiosament, la complexitat temporal tant de l'Algorisme 1 com de l'Algorisme 3 és $O(n \log n)$ però, malgrat això, l'Algorisme 3 és deu vegades més ràpid. Això es pot explicar pel fet que ordenar és un procediment senzill i només s'executa una vegada al començament de l'Algorisme 3, i la resta de l'algorisme triga temps lineal. Per altra banda, l'Algorisme 1 manté un arbre binari equilibrat complex durant tot l'algorisme.²

²N. del T.: L'Algorisme 2 triga $O(n)$, però també és més lent a la pràctica que l'Algorisme 3, que triga temps $O(n \log n)$. En principi, valors cada cops més grans de n haurien d'afavorir l'Algorisme 2, fins a ser un nombre arbitràriament gran de vegades més ràpid que l'Algorisme 3. A la pràctica, $O(\log n)$ creix molt lentament; a més a més, algunes operacions, com ara fer servir memòria addicional, poden ser més costoses com més gran sigui la n i això pot negar l'avantatge teòric de ser $O(\log n)$ cops més ràpid.

Capítol 5

Cerca completa

Cerca completa és un mètode general que es pot fer servir per a resoldre gairebé qualsevol problema d'algorísmic. La idea és generar totes les possibles solucions del problema amb força bruta i, a continuació, seleccionar la millor solució, o comptar el nombre de solucions, segons el problema.

La cerca completa és una bona tècnica si hi ha prou temps per a generar totes les solucions, ja que la cerca sol ser fàcil d'implementar i sempre dóna la resposta correcta. Si la cerca completa és massa lenta haurem de fer servir altres tècniques, com els algorismes *greedy* o la programació dinàmica.

5.1 Generar subconjunts

Considerem el problema de generar tots els subconjunts d'un conjunt de n elements. Per exemple, els subconjunts de $\{0, 1, 2\}$ són \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ i $\{0, 1, 2\}$. Hi ha dos mètodes comuns per a generar subconjunts: fer una cerca recursiva o aprofitar la representació binària dels nombres enters.

Mètode 1

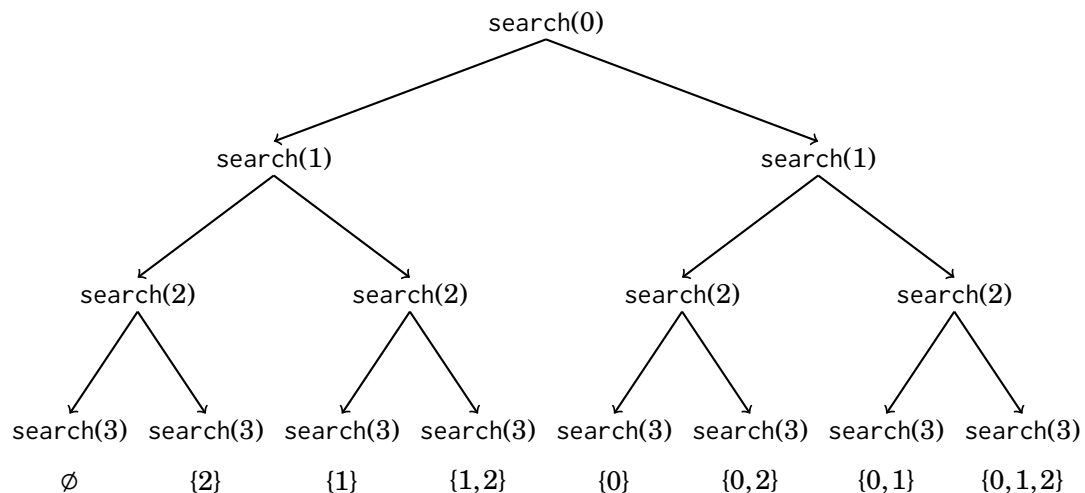
La recursivitat és una manera elegant de passar per tots els subconjunts d'un conjunt. La següent funció `search` genera els subconjunts de $\{0, 1, \dots, n-1\}$. La funció manté un vector global `v` que anirà contenint els elements de cada subconjunt. La cerca comença quan es crida la funció amb el paràmetre 0.

```
vector<int> v;  
  
void cerca(int k) {  
    if (k == n) {  
        // tracta el subconjunt v;  
    } altrament {  
        cerca(k+1);  
        v.push_back(k);  
        cerca(k+1);  
        v.pop_back();  
    }  
}
```

}

Quan la funció cerca es crida amb el paràmetre k , aquesta decideix si inclou o no l'element k al subconjunt i , en ambdós casos, es crida recursivament a ella mateixa amb paràmetre $k + 1$. Tanmateix, si $k = n$, la funció s'adona que ha processat tots els elements i , per tant, s'ha generat un nou subconjunt v .

L'arbre següent il·lustra les crides de funció quan $n = 3$. Sempre podem triar la branca esquerra (k no s'inclou al subconjunt) o la branca dreta (k s'inclou al subconjunt).



Mètode 2

Una altra manera de generar subconjunts es fer servir la representació en binari dels nombres enters. Cada subconjunt d'un conjunt de n elements es pot representar com una seqüència de n bits, que correspon a un nombre enter entre $0 \dots 2^n - 1$. Els uns en la seqüència de bits indiquen quins elements formen part del subconjunt.

La convenció habitual és que l'últim bit correspon a l'element 0, el penúltim bit és l'element 1, etcètera. Per exemple, la representació binària de 25 és 11001, que correspon al subconjunt $\{0, 3, 4\}$.

El codi següent¹ tracta tots els subconjunts d'un conjunt de n elements

```
for (int b = 0; b < (1<<n); b++) {
    // tracta el subconjunt b
}
```

El codi següent mostra com trobar el subconjunt que conté els elements corresponents a una seqüència de bits. Quan tractem cada subconjunt b , el codi construeix el vector v amb els elements del subconjunt.

```
for (int b = 0; b < (1<<n); b++) {
```

¹N. del T: La notació $1 \ll n$ és l'operador *shift*, i indica que el nombre 1 es desplaça n posicions a l'esquerra, és a dir, 2^n

```

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

```

5.2 Generar permutacions

A continuació considerem el problema de generar totes les permutacions d'un conjunt de n elements. Per exemple, les permutacions de $\{0, 1, 2\}$ són $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ i $(2, 1, 0)$. De nou, hi ha dos enfocaments: fer servir la recursivitat o recórrer les permutacions iterativament.

Mètode 1

A l'igual que amb els subconjunts, podem generar permutacions fent servir recursivitat. La següent funció cerca recorre les permutacions del conjunt $\{0, 1, \dots, n-1\}$. La funció construeix un vector global `perm` que conté la permutació, i la cerca comença quan la funció és crida sense paràmetres.

```

vector<int> perm;
vector<bool> triat(n);

void cerca() {
    if (permutacio.size() == n) {
        // tracta la permutacio perm
    } else {
        for (int i = 0; i < n; i++) {
            if (triat[i]) continue;
            triat[i] = true;
            perm.push_back(i);
            cerca();
            triat[i] = false;
            perm.pop_back();
        }
    }
}

```

Cada crida a la funció afegeix un nou element al vector `perm`. El vector `triat` indica quins elements ja han estat inclosos a la permutació. Cada cop que la mida del vector `perm` coincideix amb la mida del conjunt vol dir que hem generat una permutació nova.

Mètode 2

Un altre mètode per generar permutacions és començar amb la permutació $\{0, 1, \dots, n-1\}$ i fer servir repetidament una funció que construeix la següent

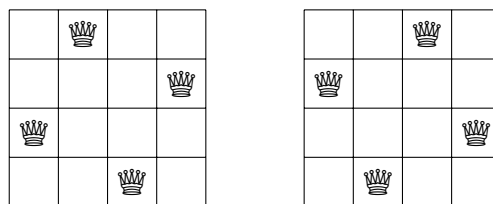
permutació en ordre creixent. La biblioteca estàndard de C++ conté la funció `next_permutation` que es pot fer servir per a això:

```
vector<int> perm;
per (int i = 0; i < n; i++) {
    perm.push_back(i);
}
do {
    // tractar la permutacio perm
} while (next_permutation(perm.begin(), perm.end()));
```

5.3 Backtracking

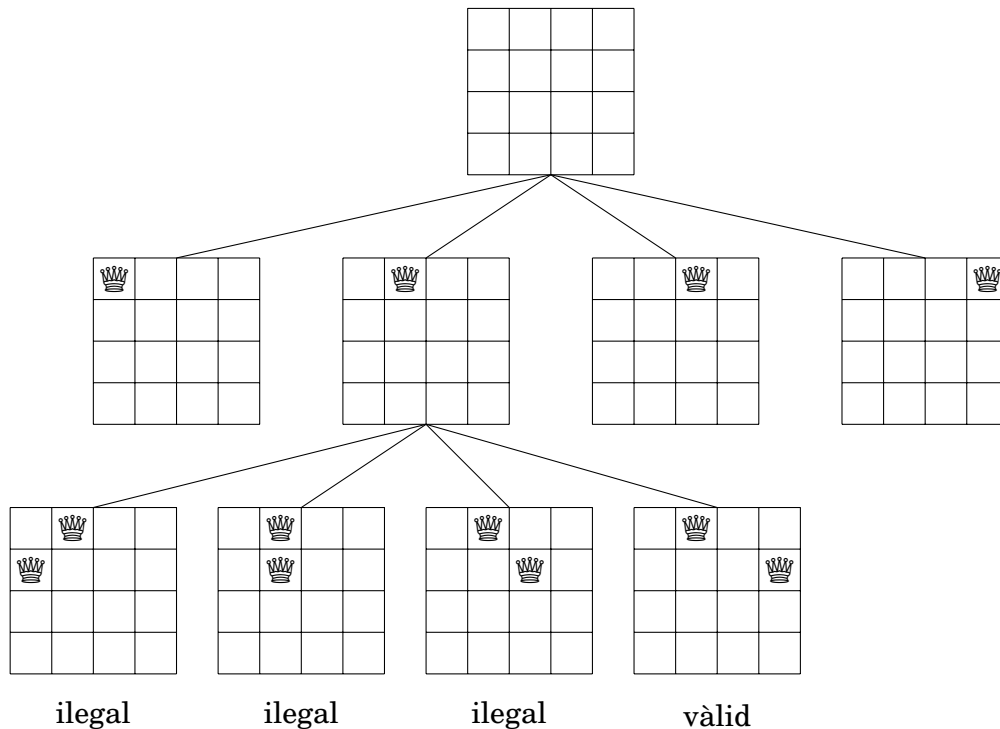
Un algorisme de **backtracking** comença amb una solució buida i amplia la solució pas a pas. La cerca passa recursivament per les diferents maneres en que es pot construir una solució.

Com a exemple, considerem el problema de calcular el nombre de maneres en què es poden col·locar n reines en un taulell d'escacs de mida $n \times n$ sense que dues reines s'amenacin. Per exemple, quan $n = 4$, hi ha dues solucions possibles:



El problema es pot resoldre fent servir backtracking, col·locant les reines al taulell fila per fila. Més precisament, per a cada fila, col·locarem una sola reina i de manera que cap de les reines anteriors l'ataqui. Quan col·loquem n reines haurem trobat una solució al problema.

Per exemple, quan $n = 4$, aquestes són algunes solucions parcials generades per l'algorisme de backtracking:



Al nivell inferior, els tres primers taulells són il·legals, perquè les reines s'ataquen entre elles. Tanmateix, el quart taulell és vàlid, i es pot estendre a una solució completa posant dues reines més al tauler. Només hi ha una manera de col·locar aquestes dues reines.

L'algorisme es pot implementar de la següent manera:

```
int compte = 0;
vector<bool> columna(n), diag1(2*n), diag2(2*n);

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

La cerca comença cridant `cerca(0)`. La mida del tauler és $n \times n$, i el codi calcula el nombre de solucions a compte.

El codi assumeix que les files i columnes del tauler estan numerats de 0 a $n - 1$. Quan la funció `cerca` és crida amb el paràmetre y , col·loca una dama a la fila y i després es crida recursivament amb el paràmetre $y + 1$. Quan $y = n$ s'ha trobat una solució i la variable `compte` s'incrementa en un.

El vector columna porta el compte de les columnes que tenen una reina, i els vectors diag1 i diag2 porten el compte de les diagonals amb reina. No està permès afegir una altra reina a una columna o diagonal que ja en conté una. Per exemple, les columnes i diagonals de el tauler 4×4 es numeren de la manera següent:

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

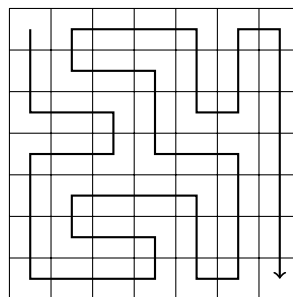
diag2

Sigui $q(n)$ el nombre de maneres per posar n reines en un tauler d'escacs $n \times n$. El procés de backtracking anterior ens diu que, per exemple, $q(8) = 92$. Quan n augmenta, la cerca es torna lenta ràpidament, perquè el nombre de solucions augmenta exponencialment. Per exemple, calcular $q(16) = 14772512$ fent servir l'algorisme anterior triga aproximadament un minut en un ordinador modern².

5.4 Podar la cerca

Sovint podem optimitzar el backtracking podant l'arbre de cerca. La idea és afegir “intel·ligència” a l'algorisme perquè es doni compte com més aviat possible que una solució parcial no es pot ampliar a una solució completa. Aquestes optimitzacions poden tenir un gran impacte sobre l'eficiència de la cerca.

Considerem el problema de calcular el nombre de camins en un taulell $n \times n$ des de la cantonada superior esquerra a la cantonada inferior dreta de manera que el camí visiti cada casella exactament una vegada. Per exemple, en un taulell 7×7 , hi ha 111712 camins d'aquest tipus. Un dels camins és el següent:



Ens centrem en el cas de 7×7 , perquè el seu nivell de dificultat és adequat a les nostres necessitats. Començarem amb un algorisme de backtracking senzill, i després l'optimitzarem pas a pas fent servir observacions de com podem podar la cerca. Després de cada optimització, mesurarem el temps d'execució de l'algorisme i el nombre de crides recursives, per a veure clarament l'efecte de cadascuna d'aquestes optimitzacions.

²No es coneix cap manera de calcular eficientment valors més grans de $q(n)$. El rècord actual és $q(27) = 234907967154122528$, calculat el 2016 [55].

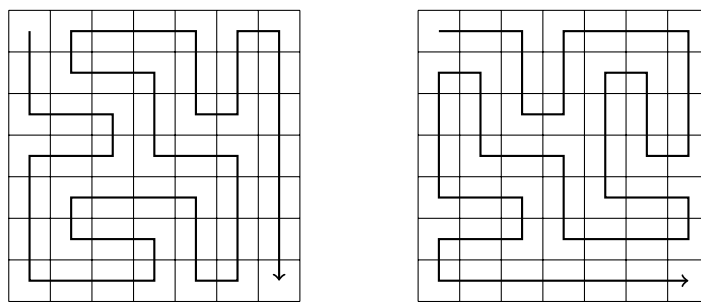
Algorisme bàsic

La primera versió de l'algorisme no conté cap optimització. Simplement fem servir el backtracking per generar tots els camins possibles des de la cantonada superior esquerra fins a la cantonada inferior dreta i comptar el nombre d'aquests camins.

- temps de funcionament: 483 segons
- nombre de crides recursives: 76 mil milions

Optimització 1

En tota solució, primer avancem un pas cap avall o cap a la dreta. Sempre hi ha dos camins que són simètrics sobre la diagonal del taulell que passa pel primer pas. Per exemple, els camins següents són simètrics:

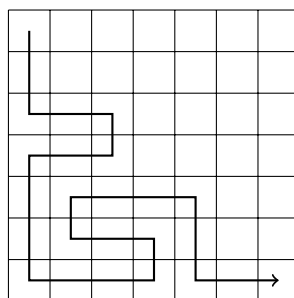


Per tant, podem decidir que el primer pas és sempre cap avall (o cap a la dreta), i multiplicar per dos el nombre de solucions.

- temps de funcionament: 244 segons
- nombre de crides recursives: 38 mil milions

Optimització 2

Si el camí arriba al quadrat inferior dret abans d'haver visitat tots els altres quadrats de la quadrícula, està clar que no serà possible completar la solució. Un exemple d'això és el camí següent:

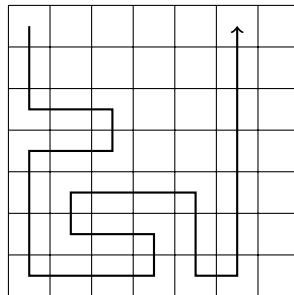


Amb aquesta observació, podem acabar la cerca immediatament si arribem massa aviat a la casella inferior dreta.

- temps de funcionament: 119 segons
- nombre de crides recursives: 20 mil milions

Optimització 3

Si el camí toca una paret i pot girar a l'esquerra o a la dreta, la graella es divideix en dues parts que contenen caselles no visitades. Per exemple, en la situació següent, el camí pot girar a l'esquerra o a la dreta:

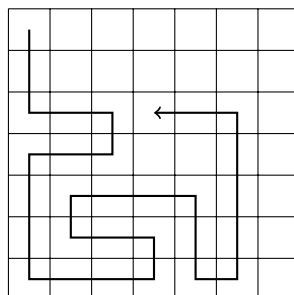


En aquest cas, ja no podem visitar totes les caselles, i podem acabar la cerca. Aquesta optimització és molt útil:

- temps de funcionament: 1.8 segons
- nombre de crides recursives: 221 milions

Optimització 4

La idea de l'optimització 3 es pot generalitzar: si el camí no pot continuar endavant però pot girar a l'esquerra o a la dreta, el taulell es divideix en dues parts que contenen caselles no visitades. Per exemple, considerem el camí següent:



Està clar que ja no podem visitar totes les caselles, de manera que acabem la cerca. Després d'aquesta optimització, la cerca és molt eficient:

- temps de funcionament: 0.6 segons
- nombre de crides recursives: 69 milions

Ara és un bon moment per deixar d'optimitzar l'algorisme i veure què hem aconseguit. El temps d'execució de l'algorisme original era 483 segons i, després de les optimitzacions, el temps ha baixat a només 0.6 segons. Les optimitzacions han fet que l'algorisme sigui gairebé 1000 vegades més ràpid.

Aquest és un fenomen habitual en el backtracking, perquè els arbres de cerca solen ser molt grans i fins i tot observacions simples poden podar eficaçment la cerca. Les optimitzacions que es produeixen durant els primers passos de l'algorisme, és a dir, a la part superior de l'arbre de cerca, són especialment útils.

5.5 Trobar-se al mig

Trobar-se al mig és una tècnica on es divideix l'espai de cerca en dues parts d'aproximadament la mateixa mida. Es realitzen cerques independents per a cada part, i finalment es combinen els resultats de les cerques.

La tècnica es pot fer servir si hi ha una manera eficient de combinar el resultat de les cerques. En aquesta situació, les dues cerques poden requerir menys temps que una cerca gran. Típicament, fent servir la tècnica de trobar-se al mig podem transformar un factor 2^n en un factor $2^{n/2}$.

Com a exemple, considerem el problema on se'ns dona una llista de n nombres i un nombre x , i volem saber si és possible triar alguns números de la llista de manera que la seva suma sigui x . Per exemple, donada la llista $[2, 4, 5, 9]$ i $x = 15$, podem triar els números $[2, 4, 9]$ per obtenir $2 + 4 + 9 = 15$. Tanmateix, fent servir la mateixa llista i $x = 10$, ja no és possible obtenir la suma.

Un algorisme senzill que resol el problema és iterar tots els subconjunts dels elements i comprovar si la suma d'algun dels subconjunts és x . El temps d'execució d'aquest algorisme és $O(2^n)$, perquè hi ha 2^n subconjunts. No obstant això, fent servir la tècnica de trobar-se al mig, podem aconseguir un algorisme de temps $O(2^{n/2})$ més eficient³. Tingueu en compte que $O(2^n)$ i $O(2^{n/2})$ són diferents complexitats perquè $2^{n/2}$ és igual a $\sqrt{2^n}$.

La idea és dividir la llista en dues llistes A i B de manera que cada llista contingui aproximadament la meitat dels números. La primera cerca genera tots els subconjunts de A i emmagatzema les seves sumes en una llista S_A . De manera semblant, la segona cerca crea una llista S_B a partir de B . Després d'això, n'hi ha prou comprovant si és possible triar un element de S_A i un altre de S_B de manera que la seva suma sigui x . Això només és possible si hi ha alguna manera de formar la suma x amb els números de la llista original.

Per exemple, suposem que la llista és $[2, 4, 5, 9]$ i $x = 15$. Primer, dividim la llista en $A = [2, 4]$ i $B = [5, 9]$. Després d'això, creem llistes $S_A = [0, 2, 4, 6]$ i $S_B = [0, 5, 9, 14]$. En aquest cas, podem formar la suma $x = 15$, perquè S_A conté la suma 6, S_B conté la suma 9 i $6 + 9 = 15$. Això correspon a la solució $[2, 4, 9]$.

Podem implementar l'algorisme de manera que la seva complexitat temporal és $O(2^{n/2})$. Primer, generem llistes ordenades S_A i S_B , que es pot fer en $O(2^{n/2})$, fent servir una tècnica semblant a la fusió (merge). Després, donat que les llistes

³Aquesta idea va ser introduïda l'any 1974 per E. Horowitz i S. Sahni [39].

están ordenadas, podem comprovar en temps $O(2^{n/2})$ si la suma x es pot crear a partir de S_A i S_B .

Capítol 6

Algorismes greedy

Un **algorisme greedy** (cobdiciós) es aquell que construeix una solució al problema fent sempre la tria que sembla millor en aquell moment. Un algorisme greedy mai desfà una opció ja triada, sinó que construeix la solució final directament. Per aquest motiu, els algorismes greedy solen ser molt eficients.

La dificultat de dissenyar algorismes greedy és trobar una estratègia que sempre produeixi una solució òptima al problema. En un algorisme greedy ha de passar que les eleccions que són localment òptimes siguin també globalment òptimes. Sovint no és fàcil d'argumentar perquè un algorisme greedy concret funciona.

6.1 Problema de les monedes

Com a primer exemple, considerem un problema on se'ns dóna un conjunt de monedes i la nostra feina és formar una quantitat de diners n fent servir les monedes. Els valors de les monedes són $\text{monedes} = \{c_1, c_2, \dots, c_k\}$, i cada moneda es pot utilitzar tantes vegades com vulguem. Quin és el nombre mínim de monedes necessàries?

Per exemple, si les monedes són les monedes d'euro (en cèntims)

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

i $n = 520$, necessitem almenys quatre monedes. La solució òptima és seleccionar monedes $200 + 200 + 100 + 20$ la suma dels quals és 520.

Algorisme greedy

Un algorisme greedy senzill que resol el problema consisteix en seleccionar sempre la moneda més gran possible, fins que s'hagi construït la suma de diners requerida. Aquest algorisme funciona en el cas d'exemple, perquè primer seleccionem dues monedes de 200 cèntims, després una moneda de 100 cèntims i finalment una moneda de 20 cèntims. Però, com sabem que aquest algorisme sempre funciona?

Resulta que si les monedes són les monedes d'euro, l'algorisme greedy *sempre* funciona, és a dir, sempre produeix una solució amb el mínim nombre possible de monedes. Això es pot argumentar de la manera següent:

En primer lloc, cada moneda 1, 5, 10, 50 i 100 pot aparèixer com a màxim una vegada en una solució òptima, perquè si la solució contingués dues d'aquestes monedes, podríem canviar-les per una sola moneda i obtenir una solució millor. Per exemple, si la solució contingués les monedes $5 + 5$, podríem substituir-les per una moneda 10.

De la mateixa manera, les monedes 2 i 20 només poden aparèixer com a màxim dues vegades en una solució òptima, perquè d'altra forma podríem reemplaçar les monedes $2 + 2 + 2$ per monedes $5 + 1$ i les monedes $20 + 20 + 20$ per monedes $50 + 10$. A més, una solució òptima no pot contenir les monedes $2 + 2 + 1$ o $20 + 20 + 10$, perquè podríem substituir-les per monedes 5 i 50.

Fent servir aquestes observacions, hem de veure que per cada moneda x no és possible construir de manera òptima una suma x o qualsevol suma més gran utilitzant només monedes que són més petits que x . Per exemple, si $x = 100$, la suma òptima més gran fent servir només monedes més petites és $50 + 20 + 20 + 5 + 2 + 2 = 99$. Així, l'algorisme greedy que sempre selecciona la moneda més gran produeix la solució òptima.

Aquest exemple mostra que pot ser difícil argumentar perquè un algorisme greedy funciona, fins i tot si l'algorisme mateix és simple.

Cas general

En el cas general, el conjunt de monedes pot contenir qualsevol moneda i l'algorisme greedy ja *no* produeix necessàriament una solució òptima.

Podem demostrar que un algorisme greedy no funciona mostrant un contraexemple on l'algorisme ens dona una resposta incorrecta. En aquest problema és fàcil trobar-ne un: si les monedes són $\{1, 3, 4\}$ i la suma objectiu és 6, l'algorisme greedy produeix la solució $4 + 1 + 1$ mentre que la solució òptima és $3 + 3$.

No se sap si el problema generalitzat de la moneda es pot resoldre fent servir algun algorisme greedy¹. Tanmateix, com veurem al capítol 7, en alguns casos el problema generalitzat pot ser resolt eficientment fent servir un algorisme de programació dinàmica que sempre dona la resposta correcta.

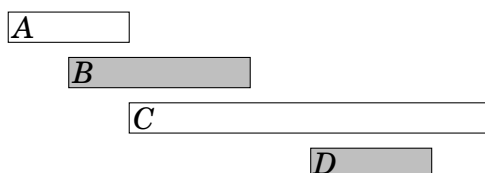
6.2 Scheduling

Molts problemes de *scheduling* (planificació horària) es poden resoldre fent servir algorismes greedy. Un problema clàssic és el següent: donats n esdeveniments amb els corresponents temps d'inici i de final, troba un scheduling que inclogui tants esdeveniments com sigui possible. No és permet seleccionar un esdeveniment parcialment. Per exemple, considerem els esdeveniments següents:

¹No obstant això, és possible *comprovar* en temps polinòmic si l'algorisme greedy que s'ha presentat en aquest capítol funciona amb un conjunt determinat de monedes [53].

esdeveniment	hora d'inici	hora final
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

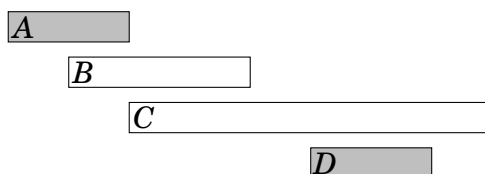
En aquest cas, el nombre màxim d'esdeveniments és dos. Per exemple, podem seleccionar els esdeveniments *B* i *D* com segueix:



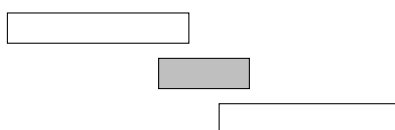
És possible inventar diversos algorismes greedy per al problema, però quin d'ells funciona en tots els casos?

Algorisme 1

La primera idea és començar seleccionant els esdeveniments més *curts* possibles. En l'exemple anterior l'algorisme selecciona els següent esdeveniments:



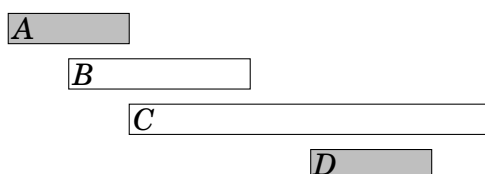
Tanmateix, seleccionar esdeveniments curts no sempre és una estratègia correcta. Per exemple, l'algorisme falla en el cas següent:



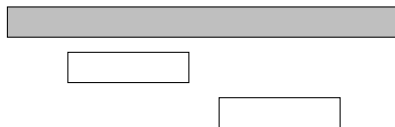
Si seleccionem l'esdeveniment més curt, només podem seleccionar un, quan en realitat seria possible seleccionar-ne dos.

Algorisme 2

Una altra idea és seleccionar sempre l'esdeveniment que *comença* tan d'hora com sigui possible. Aquest algorisme selecciona els esdeveniments següents:



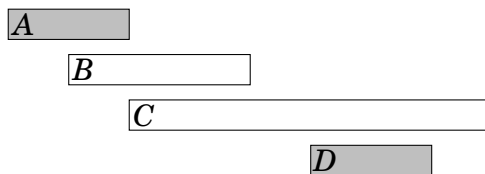
Tanmateix, també podem trobar un contraexemple per a aquest algorisme. Per exemple, en el cas següent, l'algorisme només selecciona un esdeveniment:



Si seleccionem el primer esdeveniment, ja no és possible seleccionar-ne cap d'altre, quan en realitat hauríem pogut seleccionar dos esdeveniments

Algorisme 3

La tercera idea és seleccionar l'esdeveniment que *acabi* tan *d'hora* com sigui possible. Aquest algorisme selecciona els esdeveniments següents:



Resulta que aquest algorisme *sempre* produeix una solució òptima. La raó d'això és que triar l'esdeveniment que acabi tan aviat com sigui possible sempre és una elecció òptima. Després d'aquesta tria, també és una elecció òptima triar el següent esdeveniment fent servir la mateixa estratègia, etc., fins que no puguem seleccionar-ne més.

Perquè l'algorisme funciona? Sigui X l'esdeveniment que acaba primer, i considerem una solució (òptima o no) que no tingui X . Sigui Y el primer esdeveniment d'aquesta solució. Com que X acaba abans (o igual) que Y , podem reemplaçar Y per X en la solució sense causar cap conflicte amb els esdeveniments posteriors. Així doncs, per a qualsevol solució òptima sense X , n'existeix una altra solució òptima amb X , la qual cosa demostra que triar l'esdeveniment que acaba abans mai ens pot portar a error.

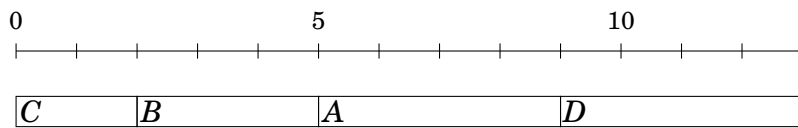
6.3 Tasques i terminis

Considerem ara un problema on se'ns dona n tasques amb durades i terminis i la nostra feina és triar en quin ordre s'han de fer les tasques. Per a cada tasca, guanyem $d - x$ punts on d és la data límit de la tasca i x és el moment en què acabem la tasca. Quina és la puntuació total més gran que podem obtenir?

Per exemple, suposem que les tasques són les següents:

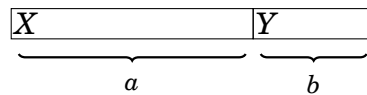
tasca	durada	termini
A	4	2
B	3	5
C	2	7
D	4	5

En aquest cas, aquesta és una assignació òptima:

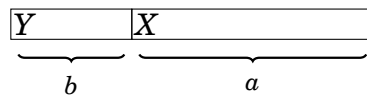


En aquesta solució, C ens dóna 5 punts, B ens dóna 0 punts, A ens dóna -7 punts i D ens dóna -8 punts, per tant, la puntuació total és de -10.

Sorprenentment, la solució òptima a aquest problema no depèn en absolut dels terminis. Una estratègia greedy correcta és simplement realitzar les tasques ordenades per la seva durada en ordre creixent. La raó d'això és que si mai fem dues tasques una darrere l'altra de manera que la primera tasca triga més que la segona tasca, podem millorar la solució intercanviant les tasques. Per exemple, considerem l'assignació següent:



Com que es dóna $a > b$, hauríem d'intercanviar les tasques:



Ara X ens dóna b punts menys però Y ens dóna a punts més, de manera que la puntuació total augmenta en $a - b > 0$. En una solució òptima, per cada dues tasques consecutives qualsevol, la tasca més curta s'ha de fer abans que la tasca més llarga. Per tant, les tasques s'han de fer ordenades en funció de la seva durada.

6.4 Minimitzar sumes

Considerem el problema on se'ns donen n nombres a_1, a_2, \dots, a_n i la nostra tasca és trobar un valor x que minimitzi la suma

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

Ens centrem en els casos $c = 1$ i $c = 2$.

Cas $c = 1$

En aquest cas hem de minimitzar la suma

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

Per exemple, si els números són $[1, 2, 9, 2, 6]$, la millor solució és seleccionar $x = 2$ que produeix la suma

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

En el cas general, la millor opció per x és la *mediana* dels nombres, és a dir, el nombre del mig després d'ordenar-los. Per exemple, la llista $[1, 2, 9, 2, 6]$ es converteix en $[1, 2, 2, 6, 9]$ després d'ordenar, i la mediana és 2.

La mediana és una opció òptima, perquè si x és més petit que la mediana, la suma es fa més petita en augmentar x , i si x és més gran que la mediana, la suma es fa més petita en disminuir x . Per tant, la solució òptima és que x sigui la mediana. Si n és parell i hi ha dues medianes, qualsevol de les medianes o els valors entre les dues són òptimes.

Cas $c = 2$

En aquest cas, hem de minimitzar la suma

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

Per exemple, si els nombres són $[1, 2, 9, 2, 6]$, la millor solució és seleccionar $x = 4$ que dona lloc a la suma

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

En el cas general, la millor opció per x és la *mitjana* dels nombres. A l'exemple, la mitjana és $(1 + 2 + 9 + 2 + 6)/5 = 4$. Aquest resultat s'obté presentant la suma de la següent manera:

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

Podem ignorar l'última part perquè no depèn de x . Les parts restants formen una funció $nx^2 - 2xs$ on $s = a_1 + a_2 + \cdots + a_n$. Aquesta és una paràbola que s'obre cap amunt amb arrels $x = 0$ i $x = 2s/n$. El valor mínim és la mitjana de les arrels $x = s/n$, és a dir, la mitjana dels nombres a_1, a_2, \dots, a_n .

6.5 Compresió de dades

Un **codificació binària** assigna a cada caràcter d'una cadena un **codi** format per bits. Podem *comprimir* la cadena fent servir la codificació que reemplaça cada caràcter pel seu codi corresponent. Per exemple, la següent codificació assigna aquests codis als caràcters: A–D:

caràcter	codi
A	00
B	01
C	10
D	11

Aquesta codificació té **longitud constant** perquè cada codi té la mateixa mida. Fent servir aquesta codificació, la cadena AABACDACA es transforma en

000001001011001000

i la cadena queda comprimida a 18 bits. Tanmateix, podem comprimir encara més la cadena si fem servir codificacions de **longitud variable**, on cada codi pot tenir longituds diferents. D'aquesta manera podem fer servir codis curts per a caràcters que apareixen sovint i codis llargs per a caràcters que apareixen poques vegades. Es dona que la següent codificació és **òptima** per a la cadena anterior:

caràcter	codi
A	0
B	110
C	10
D	111

Una codificació òptima comprimeix la cadena en el mínim espai possible. En aquest cas, es genera la cadena comprimida

001100101110100,

i només calen 15 bits en lloc de 18 bits. D'aquesta manera, la millor codificació ens permet estalviar-nos 3 bits.

Exigim també que no hi hagi cap codi que sigui prefix d'un altre codi. Per exemple, està prohibit que la codificació contingui els codis 10 i 1011. El motiu és que volem poder recuperar la cadena original a partir de la cadena comprimida. Si un codi pogués ser prefix d'un altre això no sempre seria possible. Per exemple, la codificació següent *no* és vàlida:

caràcter	codi
A	10
B	11
C	1011
D	111

Amb aquesta codificació, no sabem si 1011 és la compressió de AB o de C.

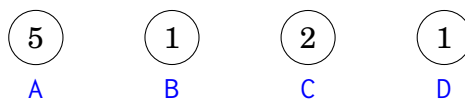
Codificació Huffman

La **codificació Huffman**² és un algorisme greedy que construeix una codificació òptima per a comprimir una cadena determinada. L'algorisme construeix un arbre binari en funció de les freqüències dels caràcters de la cadena, i trobem el codi que assignem a cada caràcter recorrent un camí des de l'arrel fins al node corresponent. Quan cop que ens movem a l'esquerra escribim un bit 0, i quan ens movem a la dreta escribim un bit 1.

Inicialment, cada caràcter és un node el pes del qual és el nombre de vegades que el caràcter apareix a la cadena. A continuació, combinem els dos nodes de pes mínim per a crear un nou node el pes del qual és la suma dels pesos dels nodes originals. El procés continua fins que combinem tots els nodes.

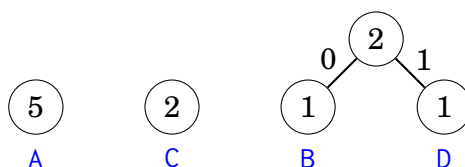
A continuació mostrem com es crea el codi Huffman per a la cadena AABACDACA. Al principi, hi ha quatre nodes, un per cada caràcters de la cadena:

²D. A. Huffman va descobrir aquest mètode en un treball de final de curs a l'universitat i va publicar l'algorisme el 1952 [40].

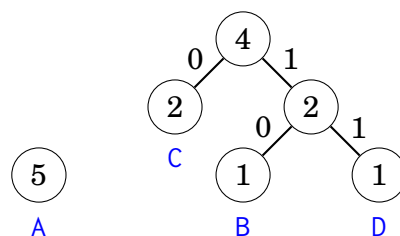


El node que representa el caràcter A té pes 5 perquè el caràcter A apareix 5 vegades a la cadena, i el mateix per la resta de pesos.

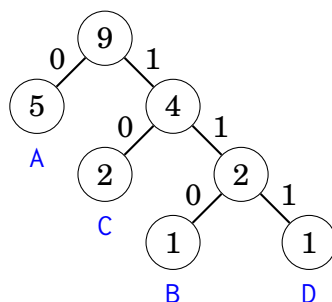
El primer pas és combinar els nodes dels caràcters B i D, tots dos amb pes 1. El resultat és:



Després d'això, combinen els nodes amb pes 2:



Finalment, combinen els dos nodes restants:



Ara tenim tots els nodes estan connectats en un arbre, i podem recuperar la codificació llegint l'arbre:

caràcter	codi
A	0
B	110
C	10
D	111

Mostrem perquè la codificació de Huffman és òptima. Primer, en tota solució òptima passa que com més freqüent és un caràcter més curt és el seu codi. Altrament, intercanviariem els codis corresponents i obtindríem una codificació millor. Per tant, els dos caràcters que apareixen amb menor freqüència han de tenir els codis més llargs. A més, aquestes dos codis han de ser de la mateixa mida. Si no és així, escurcem el codi més llarga fins la mida del codi més

curt i obtindríem una codificació. Per exemple, si els codis són 0110 i 010101, reemplacem 010101 per 0101. El nou codi no és part de la codificació perquè és un prefix d'un codi existent, i com que aquests dos són els codis més llargs, el nou codi escurçat no és prefix de cap altre. L'algorisme greedy anterior surgeix d'aplicar repetidament aquesta propietat, i fer servir l'arbre com a manera de trobar una codificació arbitrària que compleix les restriccions de llargada.

Capítol 7

Programació dinàmica

La **programació dinàmica** (*dynamic programming* o *DP*) és una tècnica que combina la correcció de la cerca completa amb l'eficiència dels algoritmes greedy. La programació dinàmica es pot aplicar si el problema es pot dividir en subproblemes superposats però que es poden resoldre independentment.

La programació dinàmica té dos usos:

- **Trobar una solució òptima:** Volem trobar una solució que sigui tan gran (o tan petita) com sigui possible.
- **Comptar el nombre de solucions:** Volem calcular el nombre total de solucions.

Primer veurem com la programació dinàmica es pot fer servir per trobar una solució òptima, i després la farem servir per comptar les solucions.

Entendre la programació dinàmica és una fita en la carrera de qualsevol programador competitiu. Tot i que la idea bàsica és senzilla, el repte és com aplicar programació dinàmica als diferents problemes. Aquest capítol presenta un conjunt de problemes clàssics que són un bon punt de partida.

7.1 Problema de les monedes

Primer ens centrem en un problema que ja vam veure al capítol 6: Donat un conjunt de valors de monedes $\text{monedes} = \{c_1, c_2, \dots, c_k\}$ i una suma objectiu de diners n , la nostra feina és obtenir suma n fent servir el menor nombre de monedes possible.

Al capítol 6, vam resoldre el problema amb un algorisme greedy que sempre triava la moneda amb valor més gran possible. L'algorisme greedy funciona, per exemple, quan les monedes són les monedes d'euro, però en el cas general l'algorisme greedy no produeix necessàriament una solució òptima.

Ara és el moment de resoldre el problema de manera eficient fent servir la programació dinàmica, i fer que l'algorisme funcioni per qualsevol conjunt de monedes. L'algorisme de programació dinàmica es basa en una funció recursiva que passa per totes les possibles maneres de formar la suma, com si fos un algorisme de força bruta. No obstant això, l'algorisme de programació dinàmica perquè fa

servir *memoization* (escriure notes) i calcula la resposta a cada subproblema un sol cop.

Formulació recursiva

La idea en la programació dinàmica és formular el problema de manera recursiva de manera que la solució al problema pugui ser calculada a partir de solucions a subproblemes més petits. En el problema de la moneda, un problema natural recursiu és el següent: quin és el menor nombre de monedes necessari per a obtenir una suma x qualsevol?

Sigui $\text{resol}(x)$ el mínim nombre de monedes necessàries per a obtenir x . El resultat de la funció depèn dels valors de les monedes. Per exemple, si $\text{monedes} = \{1, 3, 4\}$, els primers valors de la funció són els següents:

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

Per exemple, $\text{resol}(10) = 3$, perquè calen almenys 3 monedes per formar la suma 10. La solució òptima és $3 + 3 + 4 = 10$.

La propietat essencial de resol és que els seus valors poden ser calculada recursivament a partir dels seus valors més petits. La idea és centrar-se en la *primera* moneda que triem per la suma. Per exemple, en l'escenari anterior, la primera moneda pot ser 1, 3 o 4. Si primer triem la moneda 1, la feina restant és obtenir la suma 9 fent servir el nombre mínim de monedes, que és un subproblema del problema original. Per descomptat, el mateix s'aplica si triem les monedes 3 o 4. Per tant, podem calcular el nombre mínim de monedes amb la següent fórmula recursiva:

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

El cas base de la recursivitat és $\text{solve}(0) = 0$, perquè no calen monedes per formar una suma buida. Per exemple,

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

Ara estem llestos per donar una fórmula recursiva general que calcula el

nombre mínim de monedes necessàries per obtenir una suma x :

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

Primer, si $x < 0$, el valor és ∞ , perquè és impossible formar una quantitat negativa de diners. Després, si $x = 0$, el valor és 0, perquè no fan falta monedes per obtenir una suma buida. Finalment, si $x > 0$, la variable c recorre totes les maneres de triar la primera moneda de la suma.

Una vegada hem trobat una fórmula recursiva, podem implementar directament la solució en C++ (la constant INF denota infinit):

```
int resol(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int millor = INF;
    for (auto c : monedes) {
        millor = min(millor, resol(x-c)+1);
    }
    return millor;
}
```

Amb tot i això, aquesta funció no és eficient, perquè hi ha un nombre exponencial de maneres de construir la suma. A continuació, veurem com podem fer que aquesta funció sigui eficient fent servir una tècnica anomenada memoization.

Memoization

La idea de la programació dinàmica és fer servir **memoization** per calcular de manera eficient els valors d'una funció recursiva. Això vol dir que els valors de la funció s'emmagatzemen en un vector un cop calculats. Per a cada paràmetre, el valor de la funció es calcula recursivament només una vegada, i després d'això, el valor es pot obtenir directament consultant el vector.

En aquest problema, fem servir un vector

```
vector<int> valor(N, 0);
```

on $\text{valor}[x]$ indica el valor de $\text{resol}(x)$ o 0 si encara no l'hem calculat. La constant N ha estat triada per a que tots els valors necessaris capiguin als vectors.

Ara podem implementar la funció eficientment de la manera següent:

```
int resol(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (valor[x]) return valor[x];
    int millor = INF;
    for (auto c : monedes) {
```

```

        millor = min(millor, resol(x-c)+1);
    }
    valor[x] = millor;
    return millor;
}

```

La funció gestiona els casos bàsics $x < 0$ i $x = 0$ com abans. A continuació, la funció comprova si el valor ja s'ha desat anteriorment a `valor[x]` i, si és així, el la retorna directament. En cas contrari, la funció calcula el valor `resol(x)` recursivament i l'emmagatzema en `valor[x]`.

Aquest codi és eficient perquè la resposta per a cada entrada x només es calcula una vegada. Un cop emmagatzemem el valor `resol(x)` a `valor[x]` el podem recuperar eficientment quan la funció es torna a cridar amb el paràmetre x . La complexitat de l'algorisme és $O(nk)$, on n és la suma objectiu i k és el nombre de monedes.

Tingueu en compte que també podem contruir el vector `valor` de manera *iterativa* amb un bucle que calculi tots els valors de `resol` per als paràmetres $0 \dots n$:

```

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

```

De fet, la majoria dels programadors competitius prefereixen aquesta implementació, perquè és més curta i té factors constants més petits. A partir d'ara, també farem servir implementacions iteratives en els nostres exemples. Tot i això, sovint és més fàcil pensar en les solucions de programació dinàmica en termes de funcions recursives.

Construir una solució

De vegades se'ns demana trobar tant el valor d'una solució òptima com donar un exemple un exemple de solució. En el problema de les monedes, per exemple, podem declarar un altre vector que es guardi per cada suma de diners la primera moneda d'una solució òptima:

```

vector<int> primera(N);

```

Podem modificar l'algorisme de la següent manera:

```

valor[0] = 0;
for (int x = 1; x <= n; x++) {

```



```

    valor[x] = INF;
    per (auto c : monedes) {
        if (x-c >= 0 && valor[x-c]+1 < valor[x]) {
            valor[x] = valor[x-c]+1;
            primera[x] = c;
        }
    }
}

```

Després d'això, podem fer servir el codi següent per a imprimir les monedes que apareixen en una solució òptima amb suma n :

```

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

```

Comptar el nombre de solucions

Considerem ara una altra versió del problema de les monedes on la nostra feina és calcular el nombre de maneres de produir la suma x fent servir les monedes. Per exemple, si $\text{monedes} = \{1, 3, 4\}$ i $x = 5$, hi ha un total de 6 maneres:

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

De nou, podem resoldre el problema de manera recursiva. Sigui $\text{solve}(x)$ el nombre de maneres d'obtenir formar la suma x . Per exemple, si $\text{monedes} = \{1, 3, 4\}$, aleshores $\text{solve}(5) = 6$ i la fórmula recursiva és

$$\begin{aligned} \text{resol}(x) = & \text{resol}(x-1) + \\ & \text{resol}(x-3) + \\ & \text{resol}(x-4). \end{aligned} \quad (7.1)$$

La funció recursiva general és la següent:

$$\text{resol}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{monedes}} \text{resol}(x-c) & x > 0 \end{cases} \quad (7.2)$$

Si $x < 0$, el valor és 0, perquè no hi ha solucions. Si $x = 0$, el valor és 1, perquè només hi ha manera d'obtenir la suma buida. En cas contrari calculem la suma de tots els valors de la forma $\text{resol}(x-c)$ on c pertany a monedes .

El codi següent construeix un vector num tal que $\text{num}[x]$ és igual el valor de $\text{solve}(x)$ per a $0 \leq x \leq n$:

```

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

```

Sovint el nombre de solucions és tan gran que no se'ns demana calcular el nombre exacte sinó la resposta mòdul m on, per exemple, $m = 10^9 + 7$. Això es pot fer fent que tots els càlculs es fàcil mòdul m . En el codi anterior, n'hi ha prou afegint la línia

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

després de

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

Amb això ja hem discutit totes les idees bàsiques de la programació dinàmica. Com que la programació dinàmica es pot fer servir en moltes situacions diferents, presentarem ara un conjunt de problemes que mostren més exemples sobre les possibilitats de la programació dinàmica.


7.2 Subseqüència creixent més llarga

El nostre primer problema és trobar la **subseqüència creixent més llarga** en un vector v de n elements. Aquesta és una longitud màxima d'una seqüència d'elements del vector, triats d'esquerra a dreta, i que cada element de la seqüència és més gran que l'element anterior. Per exemple, al vector

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

es té que la subseqüència creixent més llarga conté 4 elements:

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



Sigui $\text{longitud}(k)$ la longitud de la subseqüència creixent més llarga que acaba a la posició k . Si calculèssim tots els valors de $\text{longitud}(k)$ on $0 \leq k \leq n-1$, descobriríem quina és la longitud de la subseqüència creixent més llarga. Per

exemple, els valors de la funció per al vector anterior són els següents:

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

Per exemple, $\text{longitud}(6) = 4$, perquè la subseqüència creixent més llarga que acaba a la posició 6 consta de 4 elements.

Per calcular un valor de $\text{longitud}(k)$, hauríem de trobar una posició $i < k$ per a la qual $v[i] < v[k]$ i $\text{longitud}(i)$ és tan gran com sigui possible. D'aquí deduïm que $\text{longitud}(k) = \text{longitud}(i) + 1$, perquè aquesta és una manera òptima d'afegir $v[k]$ a una subseqüència. Tanmateix, si no existeix aquesta posició i , llavors $\text{longitud}(k) = 1$, és a dir, l'única subseqüència possible es aquella que només conté l'element $v[k]$.

Podem fer servir programació dinàmica perquè tots els valors de la funció es poden calcular a partir de valors més petits. En el codi següent emmagatzemem els valors de la funció en el vector `longitud`.

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

Aquest codi funciona en temps $O(n^2)$ perquè consta de dos bucles niats. Tanmateix, també és possible implementar el càlcul anterior de programació dinàmica de manera més eficient en temps $O(n \log n)$. Pots trobar una manera de fer-ho?

7.3 Camins en una quadrícula

El problem següent és trobar un camí que vagi de la cantonada superior esquerra a la cantonada inferior dreta d'una quadrícula $n \times n$, moguent-nos únicament cap avall i cap a la dreta. Cada quadrat conté un nombre enter positiu, i el camí s'ha de construir de tal manera que la suma dels valors al llarg el camí sigui la més gran possible.

La imatge següent mostra un camí òptim en una quadrícula:

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 dels valors del camí és 67, i aquesta és la suma més gran possible per a qualsevol camí des del cantonada superior esquerra a la cantonada inferior dreta.

Suposem que les files i columnes de la quadrícula estan numerades de l'1 al n , i que $\text{valor}[y][x]$ és el valor de la casella (y, x) . Sigui $\text{suma}(y, x)$ la suma màxima de camins que van des de la cantonada superior esquerra a la casella (y, x) . Aleshores, $\text{suma}(n, n)$ ens diu la suma màxima de la cantonada superior esquerra a la cantonada inferior dreta. Per exemple, a la quadrícula anterior, $\text{suma}(5, 5) = 67$.

Les sumes màximes es poden calcular de manera recursiva com segueix:

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

La fórmula recursiva es basa en l'observació que un camí que acaba a la casella (y, x) ha de passar per la casella $(y, x - 1)$ o la casella $(y - 1, x)$:



Per tant, només hem de triar la direcció que maximitzi la suma. Si definim $\text{suma}(y, x) = 0$ per $y = 0$ o $x = 0$ (els camins no poden sortir de la quadrícula), tenim que la fórmula recursiva també funciona quan $y = 1$ o $x = 1$.

Com que la funció suma té dos paràmetres, el vector de la programació dinàmica també té dues dimensions. Per exemple, podem fer servir la matriu

```
vector<vector<int>> suma(N, vector<int>(N));
```

i calcula les sumes de la següent manera:

```
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 complexitat temporal de l'algorisme és $O(n^2)$.

7.4 Problemes de motxilla

El terme **motxilla** (*knapsack*) es refereix a problemes on es dóna un conjunt d'objectes, i volem buscar subconjunts amb algunes propietats. Els problemes de motxilla sovint es poden resoldre fent servir programació dinàmica.

En aquest apartat, ens centrem en el següent problema: donada una llista de pesos $[w_1, w_2, \dots, w_n]$, determinar totes les sumes que es poden construir fent servir els pesos. Per exemple, donats els pesos $[1, 3, 3, 5]$, podem obtenir les següents sumes:

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 aquest cas, totes les sumes entre $0 \dots 12$ són possibles, excepte 2 i 10. Per exemple, la suma 7 és possible perquè podem seleccionar els pesos $[1, 3, 3]$.

Per resoldre el problema, ens centrem en els subproblemes on només fem servir els primers k pesos per construir sumes. Sigui $\text{possible}(x, k) = \text{true}$ si podem construir una suma x fent servir els primers k pesos, i $\text{possible}(x, k) = \text{false}$ en cas contrari. Els valors de la funció es poden calcular recursivament de la següent manera:

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

La fórmula es basa en el fet que podem fer servir o no fer servir el pes w_k a la suma. Si fem servir w_k , la tasca restant és trobar la suma $x - w_k$ fent servir els primers $k - 1$ pesos, i si no fem servir w_k , la tasca restant és formar la suma x fent servir els primers pesos $k - 1$. Els casos bàsics són

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

perquè si no fem servir pesos, només podem formar la suma 0.

La taula següent mostra tots els valors de la funció per als pesos $[1, 3, 3, 5]$ (el símbol "X" indica els valors reals):

$k \backslash x$	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

Després de calcular aquests valors, $\text{possible}(x, n)$ ens diu si podem construir la suma x fent servir tots pesos.

Sigui W la suma total dels pesos. El codi següent es correspon a la funció recursiva anterior i troba la solució en temps $O(nW)$.

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

No obstant això, en aquest cas hi ha una millor implementació que només fa servir un vector unidimensional `possible[x]` que indica si podem construir un subconjunt amb la suma x . El truc és actualitzar el vector de dreta a esquerra per cada nou pes¹:

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

Tingueu en compte que la idea presentada aquí es pot fer servir per molts problemes de motxilla. Per exemple, si ens donen objectes amb pesos i valors, podríem determinar quin subconjunt d'objectes ens donarien valor màxim per a cadascun dels pesos possibles.

7.5 Distància d'edició

La **distància d'edició** o **distància de Levenshtein**² és el nombre mínim d'operacions d'edició que fan falta per transformar una cadena en una altra cadena. Les operacions d'edició permeses són les següents:

- inserir un caràcter (per exemple, $ABC \rightarrow ABCA$)
- eliminar un caràcter (per exemple, $ABC \rightarrow AC$)
- modificar un caràcter (per exemple, $ABC \rightarrow ADC$)

Per exemple, la distància d'edició entre LOVE i MOVIE és 2, perquè primer podem realitzar l'operació $LOVE \rightarrow MOVE$ (modificar) i després l'operació $MOVE \rightarrow MOVIE$ (inserir). Aquest és el nombre més petit possible d'operacions, perquè és evident que una sola operació no és suficient.

Suposem que se'ns dona una cadena x de longitud n i una cadena y de longitud m , i volem calcular la distància d'edició entre x i y . Per resoldre el problema, definim una funció $\text{dist}(a, b)$ que dona la distància d'edició entre prefixos $x[0 \dots a]$

¹N. del T.: Una solució alternativa i general consisteix en fer servir dos vectors unidimensionals, un per la fila actual (k) i un per la fila anterior ($k-1$).

²La distància rep el nom de V. I. Levenshtein, que la va estudiar per la seva relació amb les codificacions binàries [49].

i $y[0 \dots b]$. Així, utilitzant aquesta funció, la distància d'edició entre x i y és igual a $\text{dist}(n-1, m-1)$.

Podem calcular valors de dist com segueix:

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

Aquí $\text{cost}(a, b) = 0$ si $x[a] = y[b]$, i en cas contrari $\text{cost}(a, b) = 1$. La fórmula considera les següents maneres d'editar la cadena x :

- $\text{dist}(a, b-1)$: inserir un caràcter al final de x
- $\text{dist}(a-1, b)$: eliminar l'últim caràcter de x
- $\text{dist}(a-1, b-1)$: modificar, si és necessari, l'últim caràcter de x

En els dos primers casos, fa falta una sola operació d'edició (inserir o eliminar). En el darrer cas, si $x[a] = y[b]$, no fa falta gastar cap operació, però en cas contrari necessitem una operació d'edició (modificar).

La taula següent mostra els valors de dist en el cas exemple:

		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

La cantonada inferior esquerra de la taula ens diu que la distància d'edició entre LOVE i MOVIE és 2. La taula també mostra com construir una seqüència mínima d'operacions d'edició. En aquest trobem el camí següent:

		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

Els últims caràcters de AMOR i MOVIE són iguals, de manera que la distància d'edició entre ells és igual a la distància d'edició entre LOV i MOVI. Podem fer servir una operació d'edició per eliminar l'últim caràcter I de MOVI. Per tant, la distància d'edició és un més gran que la distància d'edició entre LOV i MOV, etc.

7.6 Comptar rajoles

De vegades, els estats d'una solució de programació dinàmica són més complexes que les combinacions fixes de nombres. Per exemple, considerem el problema de calcular el nombre de maneres diferents d'omplir una quadrícula $n \times m$ fent servir fitxes de mida 1×2 i 2×1 . Una solució vàlida per a la quadrícula 4×7 és



i el nombre total de solucions és 781.

El problema es pot resoldre amb programació dinàmica si passem per la quadrícula fila per fila. Cada fila d'una solució es pot representar com una cadena que conté m caràcters del conjunt $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$. Per exemple, la solució anterior consta de quatre files que es corresponen amb les següents cadenes:

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

Sigui $\text{count}(k, x)$ el nombre de maneres de construir una solució per a les files $1 \dots k$ de la graella de manera que la cadena x correspon a la fila k . Aquí és possible utilitzar la programació dinàmica, perquè l'estat d'una fila està restringit només per l'estat de la fila anterior.

Una solució és vàlida si la fila 1 no conté el caràcter \sqcup , la fila n no conté el caràcter \sqcap , i totes les files consecutives són *compatibles*. Per exemple, les files $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup$ i $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$ són compatibles, mentre que les files $\sqcap \sqsubset \sqcap \sqsubset \sqsubset \sqcap$ i $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$ no són compatibles.

Com que una fila consta de m caràcters i n'hi ha quatre opcions per a cada caràcter, el nombre de files diferents és com a màxim 4^m . Per tant, la complexitat temporal de la solució és $O(n4^{2m})$ perquè podem passar pels $O(4^m)$ estats possibles de cada fila i, per a cada estat, hi ha $O(4^m)$ estats possibles de la fila anterior. A la pràctica, és una bona idea girar la quadrícula per a que el costat més curt tingui longitud m , donat que el factor 4^{2m} domina la complexitat temporal.

És possible millorar la solució amb una representació més compacta de les files. Resulta que n'hi ha prou amb saber quines de les columnes de la fila anterior contenen el quadrat superior d'una rajola vertical. Així, podem representar una fila utilitzant només caràcters \sqcap i \square , on \square és una combinació de caràcters \sqcup , \sqsubset i \sqsupset . Fent servir aquesta representació, només n'hi ha 2^m files diferents, i la complexitat temporal és $O(n2^{2m})$.

Com a nota final, també hi ha una fórmula directa sorprenent per calcular el nombre de rajoles³:

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

Aquesta fórmula és molt eficient, perquè calcula el nombre de mosaics en $O(nm)$ temps, però com que la resposta és un producte de nombres reals, fer servir la fórmula requereix emmagatzemar els resultats intermedis amb precisió.

³Sorprenentment, aquesta fórmula va ser descoberta l'any 1961 per dos equips de recerca [43, 67] que funcionaven de manera independent.

Capítol 8

Anàlisi amortitzada

La complexitat temporal d'un algorisme sovint és fàcil d'analitzar simplement mirant l'estructura de l'algorisme: quins bucles conté l'algorisme i quantes vegades s'executen. Tanmateix, de vegades una anàlisi directa no dona una imatge real de l'eficiència de l'algorisme.

L'anàlisi amortitzada es pot fer servir per analitzar algorismes que contenen operacions la complexitat temporal dels quals varia. La idea és estimar el temps total utilitzat per totes aquestes operacions durant l'execució de l'algorisme, en lloc de centrar-se en les operacions individuals.

8.1 Mètode dels dos punters

En el **mètode dels dos punters**, es fan servir dos punters per iterar pels valors d'un vector. Els punters només es poden moure en una direcció, cosa que garanteix que l'algorisme funciona de manera eficient. A continuació discutim dos problemes que es poden resoldre mitjançant el mètode dels dos punters.

Suma de subvector

Com a primer exemple, considerem un problema en què se'ns dona un vector de n enters positius i una suma objectiu x , i volem trobar un subvector la suma del qual és x o informar que no hi ha aquesta subvector.

Per exemple, el vector

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

conté un subvector la suma del qual és 8:

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

Aquest problema es pot resoldre en temps $O(n)$ fent servir el mètode dels dos punters. La idea és mantenir punters que assenyalin el primer i l'últim valor d'un subvector. En cada gir, el punter esquerre es mou un pas cap a la dreta i el punter dret es mou cap a la dreta sempre que la suma del subvector resultant

sigui com a màxim x . Si la suma es converteix exactament en x , s'ha trobat una solució.

Com a exemple, considereu el vector següent i una suma objectiu $x = 8$:

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

El subvector inicial conté els valors 1, 3 i 2 la suma dels quals és 6:

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

Aleshores, el punter esquerre es mou un pas cap a la dreta. El punter dret no es mou, perquè, en cas contrari, la suma del subvector superaria x .

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

De nou, el punter esquerre es mou un pas cap a la dreta, i aquesta vegada el punter dret es mou tres passos cap a la dreta. La suma del subvector és $2 + 5 + 1 = 8$, de manera que s'ha trobat un subvector la suma del qual és x .

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

El temps d'execució de l'algorisme depèn del nombre de passos que es mou el punter dret. Tot i que no hi ha cap límit superior útil sobre quants passos pot moure el punter en un *única* gir, sabem que el punter es mou *un total de* $O(n)$ passos durant l'algorisme, perquè només es mou cap a la dreta.

Com que tant el punter esquerre com el dret es mouen $O(n)$ passos durant l'algorisme, l'algorisme funciona en el temps $O(n)$.

Problema 2SUMA

Un altre problema que es pot resoldre mitjançant el mètode dels dos punters és el següent problema, també conegut com a **problema 2SUMA**: donat un vector de n nombres i una suma objectiu x , trobeu dos valors del vector de manera que suma és x , o informeu que no existeixen aquests valors.

Per resoldre el problema, primer ordenem els valors del vector en ordre creixent. Després d'això, iterem el vector fent servir dos punters. El punter esquerre comença al primer valor i es mou un pas cap a la dreta en cada torn. El punter dret comença a l'últim valor i sempre es mou cap a l'esquerra fins que la suma del valor esquerre i dret és com a màxim x . Si la suma és exactament x , s'ha trobat una solució.

Per exemple, considereu el vector següent i una suma objectiu $x = 12$:

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

Les posicions inicials dels punters són les següents. La suma dels valors és $1 + 10 = 11$ que és més petit que x .



A continuació, el punter esquerre es mou un pas cap a la dreta. El punter dret es mou tres passos cap a l'esquerra i la suma es converteix en $4 + 7 = 11$.



Després d'això, el punter esquerre torna a moure's un pas cap a la dreta. El punter dret no es mou i s'ha trobat una solució $5 + 7 = 12$.



El temps d'execució de l'algorisme és $O(n \log n)$, perquè primer ordena el vector en temps $O(n \log n)$, i després els dos punters mouen $O(n)$ passos.

Tingueu en compte que és possible resoldre el problema d'una altra manera en temps $O(n \log n)$ fent servir la cerca binària. En aquesta solució, iterem a través del vector i per a cada valor del vector intentem trobar un altre valor que produeixi la suma x . Això es pot fer fent n cerques binàries, cadascuna de les quals triga temps $O(\log n)$.

Un problema més difícil és el **problema 3SUMA**, on es demana trobar *tres* valors del vector la suma dels quals és x . Utilitzant la idea de l'algorisme anterior, aquest problema es pot resoldre en temps $O(n^2)$ ¹. Veus com?

8.2 Element menor més propers

L'anàlisi amortitzada s'utilitza sovint per estimar el nombre d'operacions realitzades en una estructura de dades. Les operacions poden estar distribuïdes de manera desigual, de tal forma que la majoria d'operacions tenen lloc en una fase determinada de l'algorisme, però el nombre total d'operacions és limitat.

Per exemple, considereu el problema de trobar per a cada element d'un vector l'**element menor més proper**, és a dir, el primer element menor que l'element original i que el precedeix en el vector. És possible que no existeixi aquest element, i en aquest cas l'algorisme hauria d'informar-ho. A continuació veurem com es pot resoldre el problema de manera eficient mitjançant una estructura de pila.

Recorrem el vector d'esquerra a dreta i mantenim una pila d'elements del vector. Per cada posició del vector, traiem elements de la pila fins que l'element superior sigui més petit que l'element actual o la pila estigui buida. Aleshores,

¹Durant molt de temps, es va pensar que resoldre el problema 3SUMA de manera més eficient que en temps $O(n^2)$ no seria possible. Tanmateix, el 2014, es va veure en [30] que no era així.

informem que l'element superior és l'element menor més proper a l'element actual, o si la pila està buida, l'element no existeix. Finalment, afegim l'element actual a la pila.

Com a exemple, considereu el vector següent:

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

En primer lloc, els elements 1, 3 i 4 s'afegeixen a la pila, perquè cada element és més gran que l'element anterior. Així, l'element menor més proper de 4 és 3 i l'element menor més proper de 3 és 1.

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

1 → 3 → 4

El següent element 2 és menor que els dos elements superiors de la pila. Així, els elements 3 i 4 s'eliminen de la pila i, a continuació, s'afegeix l'element 2 a la pila. El seu element menor més proper és 1:

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

1 → 2

Aleshores, l'element 5 és més gran que l'element 2, de manera que s'afegirà a la pila i el seu element menor més proper és 2:

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

1 → 2 → 5

Després d'això, l'element 5 s'elimina de la pila i els elements 3 i 4 s'afegeixen a la pila:

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

1 → 2 → 3 → 4

Finalment, tots els elements excepte l'1 s'eliminen de la pila i l'últim element 2 s'afegeix a la pila:

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

1 → 2

L'eficiència de l'algorisme depèn del nombre total d'operacions fets a la pila. Si l'element actual és més gran que l'element superior de la pila, s'afegeix directament a la pila, la qual cosa és eficient. Tanmateix, de vegades la pila pot contenir diversos elements més grans i es necessita temps per eliminar-los. Tot i així, cada element s'afegeix *exactament una vegada* a la pila i s'elimina *com a màxim una vegada* de la pila. Així, cada element provoca $O(1)$ operacions de pila, i l'algorisme funciona en temps $O(n)$.

8.3 Mínim de finestra lliscant

Una finestra lliscant (**sliding window**) és un subvector de mida constant que es mou d'esquerra a dreta a través del vector. A cada posició de la finestra, volem calcular una certa informació sobre els elements dins de la finestra. En aquesta secció, ens centrem en el problema de mantenir el **mínim de la finestra**, és a dir, el valor més petit dins de cada finestra.

El mínim de la finestra lliscant es pot calcular fent servir una idea semblant a la dels elements menors més propers. Mantenim una cua on cada element és més gran que l'element anterior, i el primer element sempre es correspon amb l'element mínim de la finestra. Després de cada moviment de la finestra, treiem elements del final de la cua fins que l'últim element de la cua sigui més petit que l'element de la nova finestra, o la cua quedi buida. També eliminem el primer element de la cua si ja no està dins de la finestra. Finalment, afegim l'element de la nova finestra al final de la cua.

Com a exemple, considereu el vector següent:

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

Suposem que la mida de la finestra lliscant és 4. A la primera posició de la finestra, el valor més petit és 1:

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

1	→	4	→	5
---	---	---	---	---

Aleshores, la finestra es mou un pas cap a la dreta. El nou element 3 és més petit que els elements 4 i 5 de la cua, de manera que els elements 4 i 5 s'eliminen de la cua i l'element 3 s'afegeix a la cua. El valor més petit segueix sent 1.

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

1	→	3
---	---	---

Després d'això, la finestra es mou de nou i l'element més petit 1 ja no pertany a la finestra. Així, s'elimina de la cua i el valor més petit és ara 3. També s'afegeix el nou element 4 a la cua.

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

3	→	4
---	---	---

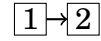
El següent element nou 1 és més petit que tots els elements de la cua. Així, tots els elements s'eliminen de la cua i aquesta només contindrà l'element 1:

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

1

Finalment la finestra arriba a la seva última posició. L'element 2 s'afegeix a la cua, però el valor més petit dins de la finestra segueix sent 1.

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



Com que cada element del vector s'afegeix a la cua exactament una vegada i s'elimina de la cua com a màxim una vegada, l'algorisme funciona en temps $O(n)$.

Capítol 9

Consultes d'interval

En aquest capítol parlem de les estructures de dades que ens permeten processar de manera eficient les consultes d'interval. En una **consulta d'interval**, la nostra tasca és calcular un valor basat en un subvector d'un vector. Les consultes d'interval típiques són:

- $\text{sum}_q(a, b)$: calculate the sum of values in range $[a, b]$
- $\text{min}_q(a, b)$: find the minimum value in range $[a, b]$
- $\text{max}_q(a, b)$: find the maximum value in range $[a, b]$

Per exemple, considereu l'interval $[3, 6]$ al vector següent:

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

En aquest cas, $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ i $\text{max}_q(3, 6) = 6$.

Una manera senzilla de processar les consultes d'interval és fer servir un bucle que recorre tots els valors de matriu de l'interval. Per exemple, la funció següent es pot fer servir per processar consultes de suma en un vector:

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

Aquesta funció funciona en temps $O(n)$, on n és la mida del vector. Així, podem processar q consultes en temps $O(nq)$ fent servir la funció. Tanmateix, si n i q són grans, aquest enfocament és lent. Afortunadament, resulta que hi ha maneres de processar les consultes d'interval de manera molt més eficient.

9.1 Consultes de vector estàtiques

Primer ens centrem en una situació en què el vector és *estàtic*, és a dir, els valors del vector mai canvien entre consultes. En aquest cas, n'hi ha prou amb construir una estructura de dades estàtica que ens indiqui la resposta a qualsevol consulta possible.

Consultes de suma

Podem processar fàcilment les consultes de suma en un vector estàtic mitjançant la construcció d'un vector suma de prefixos (**prefix sum array**). Cada valor del vector suma de prefixos és igual a la suma de valors del vector original fins a aquesta posició, és a dir, el valor a la posició k és $\text{sum}_q(0, k)$. El vector suma de prefixos es pot construir en temps $O(n)$.

Per exemple, considereu el vector següent:

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

El vector suma de prefixos corresponent és:

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

Com que el vector suma de prefixos conté tots els valors de $\text{sum}_q(0, k)$, podem calcular qualsevol valor de $\text{sum}_q(a, b)$ en temps $O(1)$ com segueix:

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

Definit $\text{sum}_q(0, -1) = 0$, la fórmula anterior també es compleix quan $a = 0$.

Per exemple, considereu l'interval $[3, 6]$:

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

En aquest cas $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Aquesta suma es pot calcular a partir de dos valors de la matriu de suma de prefix:

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

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

També és possible generalitzar aquesta idea a dimensions superiors. Per exemple, podem construir una matriu de suma de prefixos bidimensional que es pot utilitzar per calcular la suma de qualsevol submatriu rectangular en temps $O(1)$. Cada suma d'aquesta matriu correspon a una submatriu que comença a la cantonada superior esquerra de la matriu.

The following picture illustrates the idea:

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

La suma de la submatriu gris es pot calcular mitjançant la fórmula

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

on $S(X)$ indica la suma de valors d'una submatriu des de la cantonada superior esquerra fins a la posició de X .

Consultes mínimes

Les consultes de mínim són més difícils de resoldre que les consultes de suma. Tot i així, hi ha un mètode de preprocessament de temps $O(n \log n)$ força senzill després del qual podem respondre qualsevol consulta mínima en $temps O(1)$ ¹. Tingueu en compte que com que les consultes mínimes i màximes es poden processar de manera similar, ens podem centrar en les consultes mínimes.

La idea és precalcular tots els valors de $\min_q(a, b)$ on $b - a + 1$ (la longitud de l'interval) és una potència de dos. Per exemple, per al vector

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

es calculen els valors següents:

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						

¹Aquesta tècnica es va introduir a [7] i de vegades s'anomena mètode del vector dispersa (**sparse array**). També hi ha tècniques més sofisticades [22] on el temps de preprocessament és només $O(n)$, però aquests algorismes no són necessaris en la programació competitiva.

El nombre de valors precalculats és $O(n \log n)$, perquè hi ha $O(\log n)$ longituds d'interval que són potències de dos. Els valors es poden calcular de manera eficient mitjançant la fórmula recursiva

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

on $b - a + 1$ és una potència de dos i $w = (b - a + 1)/2$. Calcular tots aquests valors requereix temps $O(n \log n)$.

Després d'això, qualsevol valor de $\min_q(a, b)$ es pot calcular en temps $O(1)$ com el mínim de dos valors precalculats. Sigui k la potència de dos més gran que no superi $b - a + 1$. Podem calcular el valor de $\min_q(a, b)$ mitjançant la fórmula

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

A la fórmula anterior, l'interval $[a, b]$ es representa com la unió dels intervals $[a, a + k - 1]$ i $[b - k + 1, b]$, tots dos de longitud k .

Com a exemple, considereu l'interval $[1, 6]$:

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

La longitud de l'interval és 6, i la potència més gran de dos que no supera 6 és 4. Així, el rang $[1, 6]$ és la unió dels intervals $[1, 4]$ i $[3, 6]$:

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

Com que $\min_q(1, 4) = 3$ i $\min_q(3, 6) = 1$, concloem que $\min_q(1, 6) = 1$.

9.2 Arbre binari indexat

Un **arbre binari indexat** o un **arbre de Fenwick**² es pot veure com una variant dinàmica d'un vector suma de prefixos. Aquest admet dues operacions de temps $O(\log n)$: processar una consulta de suma d'interval i actualitzar un valor.

L'avantatge d'un arbre binari indexat és que ens permet actualitzar de manera eficient els valors del vector entre les consultes de suma. Això no seria possible si fèssim servir un vector suma de prefix, perquè després de cada actualització, caldria tornar a reconstruir tot el vector suma de prefix en temps $O(n)$.

²L'estructura d'arbre binari indexat va ser presentada per P.M. Fenwick el 1994 [21].

Estructura

Encara que el nom de l'estructura sigui *arbre* binari indexat, normalment es representa com un vector. En aquesta secció suposem que tots els vectors comencen amb index 1 (en lloc de 0), perquè dóna lloc a una implementació més senzilla.

Sigui $p(k)$ la potència de dos més gran que divideix k . Emmagatzemem un arbre binari indexat com un vector arbre de manera que

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

és a dir, cada posició k conté la suma de valors en un interval del vector original la longitud del qual és $p(k)$ i que acaba a la posició k . Per exemple, com que $p(6) = 2$, $\text{tree}[6]$ conté el valor de $\text{sum}_q(5, 6)$.

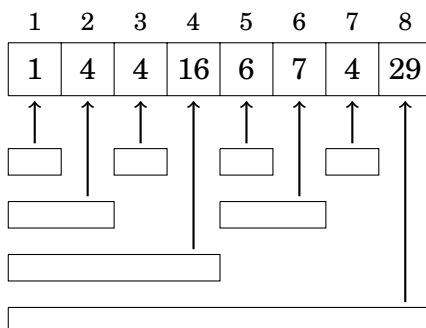
Per exemple, considereu el vector següent:

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

L'arbre binari indexat corresponent és el següent:

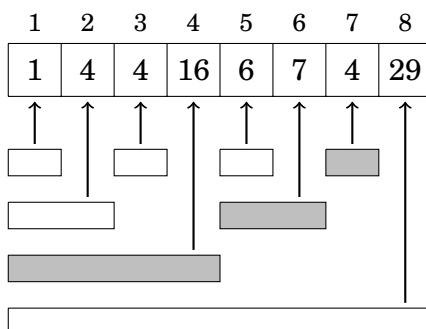
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

La imatge següent mostra més clarament com cada valor de l'arbre binari indexat correspon a un interval del vector original:



Fent servir un arbre binari indexat, qualsevol valor de $\text{sum}_q(1, k)$ es pot calcular en temps $O(\log n)$, perquè un interval $[1, k]$ sempre es pot dividir en $O(\log n)$ intervals les sumes dels quals estan emmagatzemades a l'arbre.

Per exemple, l'interval $[1, 7]$ consta dels intervals següents:



Així, podem calcular la suma corresponent de la següent manera:

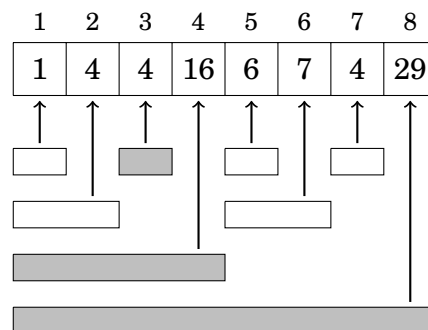
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

Per calcular el valor de $\text{sum}_q(a, b)$ on $a > 1$, podem fem servir el mateix truc que hem fet servir amb els vectors suma de prefixos:

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

Com que podem calcular tant $\text{sum}_q(1, b)$ com $\text{sum}_q(1, a - 1)$ en temps $O(\log n)$, la complexitat total és $O(\log n)$.

Quan actualitzem un valor en el vector original, hem d'actualitzar diversos valors de l'arbre binari indexat. Per exemple, si el valor a la posició 3 canvia, les sumes dels intervals següents canvien:



Com que cada element del vector pertany a $O(\log n)$ intervals de l'arbre binari indexat, n'hi ha prou amb actualitzar $O(\log n)$ valors de l'arbre.

Implementació

Les operacions d'un arbre binari indexat es poden implementar de manera eficient mitjançant operacions de bits. El fet clau és que podem calcular qualsevol valor de $p(k)$ mitjançant la fórmula

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

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

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

La funció següent augmenta en x unitats el valor del vector a la posició k (x pot ser positiu o negatiu):

```

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

```

La complexitat temporal d'ambdues funcions és $O(\log n)$, perquè les funcions accedeixen als $O(\log n)$ valors de l'arbre binari indexat, i cada moviment a la següent posició triga temps $O(1)$.

9.3 Arbre de segments

Un **arbre de segments**³ (*segment tree*) és una estructura de dades que admet dues operacions: processar una consulta d'interval i actualitzar un valor del vector. Els arbres de segments poden suportar consultes de suma, consultes de mínim i màxim i moltes altres consultes perquè ambdues operacions funcionen en $O(\log n)$ temps.

En comparació amb un arbre binari indexat, l'avantatge d'un arbre de segments és que és una estructura de dades més general. Tot i que els arbres indexats binaris només admeten consultes de suma⁴, Els arbres de segments també admeten altres consultes. D'altra banda, un arbre de segments requereix més memòria i és una mica més difícil d'implementar.

Estructura

Un arbre de segments és un arbre binari on els nodes del nivell inferior de l'arbre corresponen als elements del vector i els altres nodes contenen la informació necessària per a processar les consultes d'interval.

En aquesta secció, suposem que la mida del vector és una potència de dos i utilitzem una indexació basada en zero, perquè resulta més convenient. Si la mida del vector no és una potència de dos, sempre podem afegir-hi elements addicionals.

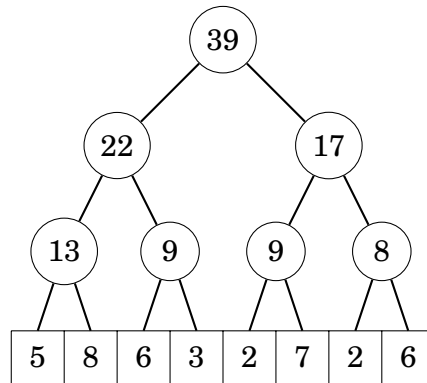
Primer parlarem dels arbres de segments que admeten consultes de suma. Com a exemple, considereu el vector següent:

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

L'arbre de segments corresponent és el següent:

³La implementació de baix a dalt d'aquest capítol correspon a la de [62]. Estructures similars es van fent servir a finals dels 70 per a resoldre problemes geomètrics [9].

⁴De fet, utilitzant *dos* arbres indexats binaris és possible suportar consultes de mínim [16], però és més complicat que utilitzar un arbre de segments.

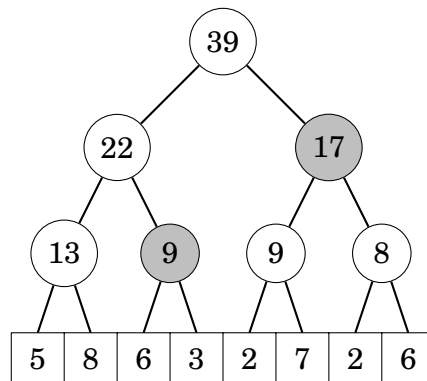


Cada node intern de l'arbre es correspon a un interval del vector la mida del qual és una potència de dos. En l'arbre anterior, el valor de cada node intern és la suma dels valors corresponents del vector i es pot calcular com la suma dels valors del fill esquerre i dret.

Resulta que qualsevol rang $[a, b]$ es pot dividir en $O(\log n)$ els valors dels quals s'emmagatzemen als nodes de l'arbre. Per exemple, considereu l'interval $[2, 7]$:

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 aquest cas, els dos nodes següents es corresponen amb l'interval:

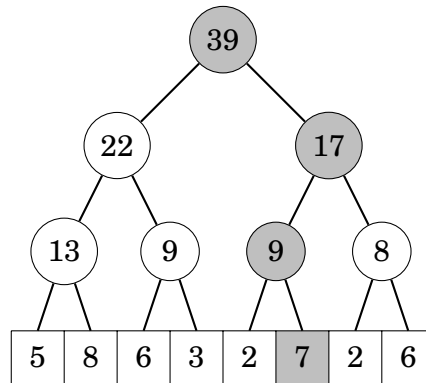


Així, una altra manera de calcular la suma és $9 + 17 = 26$.

Quan la suma es calcula fent servir nodes situats el més alt possible a l'arbre, fan falta com a màxim dos nodes a cada nivell de l'arbre. Per tant, el nombre total de nodes és $O(\log n)$.

Quan actualitzem un element del vector, hem d'actualitzar tots els nodes el valor dels quals depèn de l'element actualitzat. Això es pot fer travessant el camí des de l'element actualitzat fins al node superior i actualitzant els nodes al llarg del camí.

La imatge següent mostra quins nodes d'arbre canvien si l'element 7 del vector canvia:

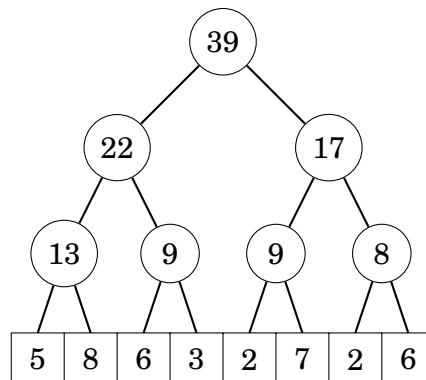


El camí de baix a dalt sempre consta de $O(\log n)$ nodes, de manera que cada actualització té aquest cost.

Implementació

Emmagatzemem un arbre de segments com un vector de $2n$ elements on n és la mida potència de dos del vector original. Els nodes de l'arbre s'emmagatzemen de dalt a baix: `tree[1]` és el node superior, `tree[2]` i `tree[3]` són els seus fills, etcètera. Finalment, els valors de `tree[n]` a `tree[2n - 1]` corresponen als valors del vector original al nivell inferior de l'arbre.

Per exemple, l'arbre de segments



s'emmagatzema de la manera següent:

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

Fent servir aquesta representació, el pare de `tree[k]` és `tree[k/2]`, i els seus fills són `tree[2k]` i `tree[2k + 1]`. Tingueu en compte que això implica que la posició d'un node és parell si és fill esquerre i imparell si és fill dret.

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

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

```

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

```

La funció manté un interval que inicialment és $[a + n, b + n]$. Aleshores, a cada pas, l'interval es mou al nivell superior en l'arbre, i abans d'això, els valors dels nodes que no pertanyen a l'interval superior s'afegeixen a la suma.

La funció següent incrementa en x unitats l'element a la posició k del vector:

```

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

```

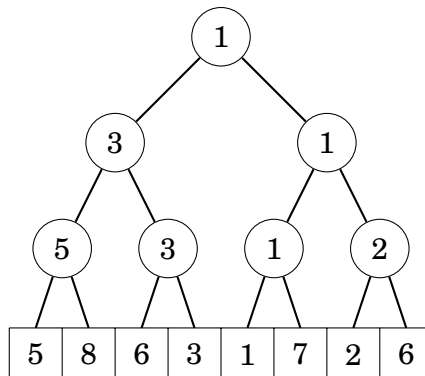
Primer, la funció actualitza el valor al nivell inferior de l'arbre. Després d'això, la funció actualitza els valors de tots els nodes interns de l'arbre, fins que arriba al node superior de l'arbre.

Les dues funcions anteriors funcionen en temps $O(\log n)$, perquè un arbre de segments de n elements consta de $O(\log n)$ nivells, i les funcions puguen l'arbre un nivell a cada pas.

Altres consultes

Els arbres de segments poden suportar totes les consultes d'interval on és possible dividir un interval en dues parts, calcular la resposta per separat per a ambdues parts i després combinar les respostes de manera eficient. Exemples d'aquestes consultes són el mínim i el màxim, el màxim comú divisor i les operacions de bits *and*, *or* i *xor*.

Per exemple, l'arbre de segment següent admet consultes de mínim:

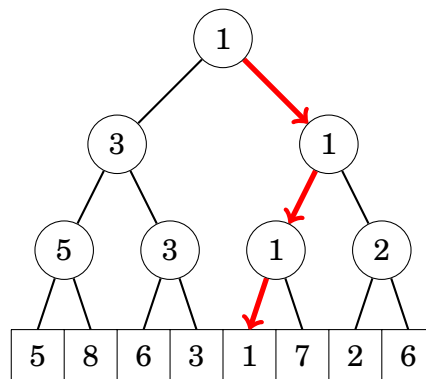


En aquest cas, cada node d'arbre conté el valor més petit de l'interval del vector corresponent. El node superior de l'arbre conté el valor més petit de tot

el vector. Les operacions es poden implementar com abans, però en comptes de sumes, es calculen mínims.

L'estructura d'arbre de segments també ens permet fer servir la cerca binària per a localitzar elements del vector. Per exemple, si l'arbre admet consultes de mínims, podem trobar la posició de l'element més petit en temps $O(\log n)$.

Per exemple, a l'arbre anterior, es pot trobar l'element amb el valor mínim 1 travessant un camí cap avall des del node superior:



9.4 Tècniques addicionals

Compressió de l'índex

Una limitació de les estructures de dades que es construeixen sobre un vector és que els elements s'indexen mitjançant nombres enters consecutius. Les dificultats sorgeixen quan es necessiten índexs grans. Per exemple, si volem fer servir l'índex 10^9 , el vector hauria de contenir 10^9 elements que requeririen massa memòria.

No obstant això, sovint podem ignorar aquesta limitació fent servir compressió de l'índex (**index compression**), on els índexs originals es substitueixen per índexs 1,2,3,, etc. Això es pot fer si coneixem prèviament tots els índexs necessaris durant l'algorisme.

La idea és substituir cada índex original x per $c(x)$ on c és una funció que comprimeix els índexs. Necessitem que l'ordre dels índexs no canviï, de manera que si $a < b$, llavors $c(a) < c(b)$. Això ens permet realitzar consultes còmodament encara que els índexs estiguin comprimits.

Per exemple, si els índexs originals són 555, 10^9 i 8, els nous índexs són:

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

Actualitzacions d'interval

Fins ara, hem implementat estructures de dades que admeten consultes d'interval i actualitzacions de valors únics. Considerem ara una situació oposada, on hauríem d'actualitzar intervals i recuperar valors únics. Ens centrem en una operació que augmenta tots els elements d'un interval $[a, b]$ en x .

Sorprenentment, podem utilitzar les estructures de dades presentades en aquest capítol també en aquesta situació. Per a fer-ho, construïm una **vector de diferències** els valors del qual indiquen les diferències entre valors consecutius del vector original. Així, el vector original és el vector suma de prefixos del vector de diferències. Per exemple, considereu el vector següent:

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

El vector de diferències per al vector anterior és el següent:

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

Per exemple, el valor 2 a la posició 6 del vector original correspon a la suma $3 - 2 + 4 - 3 = 2$ al vector de diferències.

L'avantatge del vector de diferències és que podem actualitzar un interval del vector original canviant només dos elements del vector de diferències. Per exemple, si volem augmentar en 5 els valors del vector original entre les posicions 1 i 4, n'hi ha prou amb augmentar en 5 el valor del vector de diferències a la posició 1 i disminuir en 5 el valor a la posició 5. El resultat és el següent:

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

De manera més general, per augmentar en x els valors de l'interval $[a, b]$, augmentem en x el valor a la posició a i reduïm en x el valor a la posició $b + 1$. Per tant, només cal actualitzar valors únics i processar consultes de suma, de manera que podem utilitzar un arbre binari indexat o un arbre de segments.

Un problema més difícil és donar tant suport a les consultes d'interval com a les actualitzacions d'interval. Al capítol 28 veurem que fins i tot això és possible.

Manipulació de bits

10.1 Representació de bits

101

Per exemple, la representació de bits del nombre int `-43` és

1111111111111111111111111010101.

En una representació sense signe, només es poden fer servir nombres no negatius, però el límit superior dels valors és més gran. Una variable sense signe de n bits pot contenir qualsevol nombre enter entre 0 i $2^n - 1$. Per exemple, en C++, una variable `unsigned int` pot contenir qualsevol nombre enter entre 0 i $2^{32} - 1$.

Hi ha una connexió entre les representacions: un nombre amb signe $-x$ és igual a un nombre sense signe $2^n - x$. Per exemple, el codi següent mostra que el número amb signe $x = -43$ és igual al nombre sense signe $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Si un nombre és més gran que el límit superior de la representació de bits, el nombre es desbordarà. En una representació amb signe, el nombre que segueix $2^{n-1} - 1$ és -2^{n-1} , i en una representació sense signe, el nombre que segueix $2^n - 1$ és 0¹. Per exemple, considereu el codi següent:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Inicialment, el valor de x és $2^{31} - 1$. Aquest és el valor més gran que es pot emmagatzemar en una variable int, de manera que el nombre següent després de $2^{31} - 1$ és -2^{31} .

10.2 Operacions de bits

Operació *and*

L'operació **and** x & y produeix un nombre que té un bit en les posicions on tant x com y tenen un bit. Per exemple, $22 \& 26 = 18$, perquè

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

¹(N. del T.) En C++, està permès fer operacions aritmètiques que causin *overflow* en tipus sense signe, però fer-ho amb tipus amb signe és comportament no definit (*undefined behavior*). No és un problema dels processadors, que poden fer *overflow* de nombres amb signe sense problema, sinó dels compiladors de C++, que optimitzen el codi agressivament sota el supòsit que el vostre programa mai comet *undefined behavior*. No ho feu.

Mitjançant l'operació (*and*), podem comprovar si un nombre x és parell perquè $x \& 1 = 0$ si x és parell, i $x \& 1 = 1$ si x és senar. De manera més general, x és divisible per 2^k exactament quan $x \& (2^k - 1) = 0$.

Operació *or*

L'operació **OR** $x \mid y$ produeix un nombre que té un bit en les posicions on x o y tenen un bit. Per exemple, $22 \mid 26 = 30$, perquè

$$\begin{array}{r} 10110 \quad (22) \\ \mid 11010 \quad (26) \\ \hline = 11110 \quad (30) \end{array}$$

Operació *xor*

L'operació **xor** $x \wedge y$ produeix un nombre que té un bit en les posicions on exactament un de x i y tenen un bit. Per exemple, $22 \wedge 26 = 12$, perquè

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

Operació *not*

L'operació **not** $\sim x$ produeix un nombre on tots els bits de x s'han invertit. En complement a 2 es compleix que $\sim x = -x - 1$. Per exemple, $\sim 29 = -30$.

El resultat de l'operació *not* a nivell de bits depen de la longitud de la representació de bits, perquè l'operació inverteix tots els bits. Per exemple, si els nombres són *ints* de 32 bits, el resultat és el següent:

$$\begin{array}{rcl} x & = & 29 \quad 000000000000000000000000011101 \\ \sim x & = & -30 \quad 111111111111111111111111110010 \end{array}$$

Desplaçament de bits

El desplaçament de bits (*bit shift*) a l'esquerra $x \ll k$ afegeix k bits zero al nombre, i el desplaçament de bits a la dreta $x \gg k$ elimina els k últims bits del nombre. Per exemple, $14 \ll 2 = 56$, perquè 14 i 56 corresponen a 1110 i 111000. De la mateixa manera, $49 \gg 3 = 6$, perquè 49 i 6 corresponen a 110001 i 110.

Tingueu en compte que $x \ll k$ correspon a multiplicar x per 2^k , i $x \gg k$ correspon a dividir x per 2^k arrodonint a un nombre enter.

Aplicacions

Un nombre de la forma $1 \ll k$ té un bit a la posició k i tots els altres bits són zero, de manera que podem fer servir aquests nombres per accedir als bits individuals d'un nombre donat. En particular, el k -èssim bit d'un nombre és 1 exactament quan $x \& (1 \ll k)$ no és zero. El codi següent imprimeix la representació de bits d'un nombre x de tipus *int*:

```
for (int i = 31; i >= 0; i--) {
    if (x & (1 << i)) cout << "1";
    else cout << "0";
}
```

També és possible modificar bits individuals fent servir idees similars. Per exemple, la fórmula $x \mid (1 \ll k)$ posa el k -èssim bit de x a 1, la fórmula $x \& \sim(1 \ll k)$ posa el k -èssim bit de x a 0, i la fórmula $x \wedge (1 \ll k)$ inverteix el k -èssim bit de x .

La fórmula $x \& (x - 1)$ posa l'últim bit de x a zero, i la fórmula $x \& -x$ posa tots els bits a zero, excepte l'últim. La fórmula $x \mid (x - 1)$ inverteix tots els bits després de l'últim bit. Tingueu en compte també que un nombre positiu x és una potència de dos exactament quan $x \& (x - 1) = 0$.

Funcions addicionals

El compilador g++ proporciona les funcions següents per comptar bits:

- `__builtin_clz(x)`: the number of zeros at the beginning of the number
- `__builtin_ctz(x)`: the number of zeros at the end of the number
- `__builtin_popcount(x)`: the number of ones in the number
- `__builtin_parity(x)`: the parity (even or odd) of the number of ones

The functions can be used as follows:

```
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
```

Tot i que les funcions anteriors només admeten nombres de tipus `int`, també hi ha versions `longlong` de les funcions amb el sufix `ll`.

10.3 Representació de conjunts

Cada subconjunt d'un conjunt $\{0, 1, 2, \dots, n - 1\}$ es pot representar com un nombre enter de n bits els bits del qual indiquen quins elements pertanyen al subconjunt. Aquesta és una manera eficient de representar conjunts, perquè cada element només requereix un bit de memòria i les operacions de conjunt es poden implementar com a operacions de bits.

Per exemple, com que `int` és un tipus de 32 bits, un nombre `int` pot representar qualsevol subconjunt del conjunt $\{0, 1, 2, \dots, 31\}$. La representació de bits del conjunt $\{1, 3, 4, 8\}$ és

000000000000000000000000100011010,

que correspon al nombre $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Implementació de conjunts

El codi següent declara una variable `int x` que pot contenir un subconjunt de $\{0, 1, 2, \dots, 31\}$. Després d'això, el codi afegeix els elements 1, 3, 4 i 8 al conjunt i imprimeix la seva mida.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

El codi següent imprimeix tots els elements que pertanyen al conjunt:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

Operacions de conjunts

Les operacions de conjunt es poden implementar de la següent manera com a operacions de bits:

	set syntax	bit syntax
intersection	$a \cap b$	$a \& b$
union	$a \cup b$	$a \mid b$
complement	\bar{a}	$\sim a$
difference	$a \setminus b$	$a \& (\sim b)$

Per exemple, el codi següent construeix primer els conjunts $x = \{1, 3, 4, 8\}$ i $y = \{3, 6, 8, 9\}$, i després construeix el conjunt $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```
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
```

Iteració de subconjunts

El codi següent passa per tots els subconjunts de $\{0, 1, \dots, n-1\}$:

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

El codi següent passa pels subconjunts amb exactament k elements:

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

El codi següent passa pels subconjunts d'un conjunt x :

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

10.4 Optimitzacions de bits

Molts algorismes es poden optimitzar mitjançant operacions de bits. Aquestes optimitzacions no canvien la complexitat temporal de l'algorisme, però poden tenir un gran impacte en el temps real d'execució del codi. En aquesta secció comentem exemples d'aquestes situacions.

Distàncies de Hamming

La **distància de Hamming** $\text{hamming}(a, b)$ entre dues cadenes a i b d'igual longitud és el nombre de posicions on les cadenes difereixen. Per exemple,

$$\text{hamming}(01101, 11001) = 2.$$

Considereu el problema següent: donada una llista de n cadenes de bits, cadascuna de longitud k , calculeu la distància de Hamming mínima entre dues cadenes de la llista. Per exemple, la resposta per a $[00111, 01101, 11110]$ és 2, perquè

- $\text{hamming}(00111, 01101) = 2$,
- $\text{hamming}(00111, 11110) = 3$ i
- $\text{hamming}(01101, 11110) = 3$.

Una manera senzilla de resoldre el problema és passar per tots els parells de cordes i calcular les seves distàncies de Hamming, que dóna lloc a un algorisme de temps $O(n^2k)$. La funció següent es pot utilitzar per calcular distàncies:

```
int hamming(string a, string b) {  
    int d = 0;  
    for (int i = 0; i < k; i++) {  
        if (a[i] != b[i]) d++;  
    }  
    return d;  
}
```

Tanmateix, si k és petit, podem optimitzar el codi emmagatzemant les cadenes de bits com a nombres enters i calculant les distàncies de Hamming mitjançant operacions de bits. En particular, si $k \leq 32$, podem simplement emmagatzemar les cadenes com a valors `int` i utilitzar la funció següent per calcular distàncies:

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

A la funció anterior, l'operació *xor* construeix una cadena de bits que té un bit en posicions on a i b difereixen, i a continuació calculem el nombre de bits mitjançant la funció `__builtin_popcount`.

Per a comparar les implementacions, generem una llista de 10.000 cadenes de bits aleatoris de longitud 30. Amb el primer enfocament, la cerca triga 13.5 segons, però si fem servir l'optimització de bits, només triga 0.5 segons. El codi optimitzat per bits és gairebé 30 vegades més ràpid que el codi original.

Comptar subquadrícules

Com a altre exemple, considereu el problema següent: Donada una quadrícula $n \times n$ cada quadrat de la qual és negre (1) o blanc (0), calculeu el nombre de subquadrícules que tenen totes les cantonades negres. Per exemple, la quadrícula



conté dues subquadrícules d'aquest tipus:



Hi ha un algorisme de temps $O(n^3)$ que resol el problema: passeu per tots els parells de files $O(n^2)$ i per a cada parell (a, b) calculeu el nombre de columnes que contenen un quadrat negre a les dues files en $O(n)$ temps. El codi següent assumeix que `color[y][x]` denota el color de la fila y i la columna x :

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

Aleshores, aquestes columnes representen subquadrícules $\text{count}(\text{count} - 1)/2$ amb cantonades negres, perquè podem triar-ne dues per formar una subquadrícula.

Per optimitzar aquest algorisme, dividim la quadrícula en blocs de columnes de manera que cada bloc consta de N columnes consecutives. Aleshores, cada fila s'emmagatzema com una llista de números de N bits que descriuen els colors dels quadrats. Ara podem processar N columnes simultàniament gràcies a les operacions de bits. Al codi següent, `color[y][k]` representa un bloc de N colors com a bits.

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

L'algorisme resultant funciona en temps $O(n^3/N)$.

Generem una graella aleatòria de 2500×2500 i comparem la implementació original i la implementació optimitzada per bits. El codi original triga 29.6 segons, mentre que la versió optimitzada per bits només triga 3.1 segons amb $N = 32$ (nombres int) i 1.7 segons amb $N = 64$ (nombres longlong).

10.5 Programació dinàmica

Les operacions de bits proporcionen una manera eficient i còmoda d'implementar algorismes de programació dinàmica els estats dels quals contenen subconjunts d'elements, perquè aquests estats es poden emmagatzemar com a nombres enters. A continuació discutim exemples de combinació d'operacions de bits i programació dinàmica.

Selecció òptima

Com a primer exemple, considereu el problema següent: Ens donen els preus de k productes durant n dies i volem comprar cada producte exactament una vegada. Tanmateix, podem comprar com a màxim un producte al dia. Quin és el preu total mínim? Per exemple, considereu l'escenari següent ($k = 3$ i $n = 8$):

	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 aquest escenari, el preu total mínim és de 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

Sigui $\text{price}[x][d]$ el preu del producte x el dia d . Per exemple, a l'escenari anterior $\text{price}[2][3] = 7$. Sigui $\text{total}(S, d)$ el preu total mínim per comprar un subconjunt S de productes en els primers d dies. Amb aquesta funció, la solució al problema és $\text{total}(\{0 \dots k-1\}, n-1)$.

Primer, $\text{total}(\emptyset, d) = 0$, perquè no costa res comprar un conjunt buit, i $\text{total}(\{x\}, 0) = \text{price}[x][0]$, perquè hi ha una manera de comprar un producte el primer dia. Aleshores, es pot utilitzar la recurrència següent:

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

Això vol dir que o bé no comprem cap producte el dia d o bé comprem un producte x que pertany a S . En aquest últim cas, eliminem x de S i afegim el preu de x al preu total.

El següent pas és calcular els valors de la funció mitjançant la programació dinàmica. Per emmagatzemar els valors de la funció, declarem una matriu

```
int total[1<<K][N];
```

on K i N són constants adequadament grans. La primera dimensió de la matriu correspon a una representació de bits d'un subconjunt.

En primer lloc, els casos en què $d = 0$ es poden processar de la següent manera:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Aleshores, la recurrència es tradueix al codi següent:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s & (1<<x)) {
                total[s][d] = min(total[s][d],
                                   total[s^(1<<x)][d-1] + price[x][d]);
            }
        }
    }
}
```

La complexitat temporal de l'algorisme és $O(n2^k k)$.

De les permutacions als subconjunts

Utilitzant la programació dinàmica, sovint és possible canviar una iteració sobre permutacions en una iteració sobre subconjunts². El benefici d'això és que $n!$, el

²Aquesta tècnica va ser introduïda el 1962 per M. Held i R. M. Karp [34].

nombre de permutacions, és molt més gran que 2^n , el nombre de subconjunts. Per exemple, si $n = 20$, llavors $n! \approx 2,4 \cdot 10^{18}$ i $2^n \approx 10^6$. Per tant, per a certs valors de n , podem passar de manera eficient pels subconjunts però no per les permutacions.

Com a exemple, considereu el problema següent: hi ha un ascensor amb un pes màxim x i n persones amb pes conegut que volen anar de la planta baixa a la planta superior. Quin és el nombre mínim de trajectes necessaris si les persones entren a l'ascensor en un ordre òptim?

Per exemple, suposem que $x = 10$, $n = 5$ i els pesos són els següents:

person	weight
0	2
1	3
2	3
3	5
4	6

En aquest cas, el nombre mínim de viatges és 2. Un ordre òptim és $\{0, 2, 3, 1, 4\}$, que divideix les persones en dos viatges: primer $\{0, 2, 3\}$ (pes total 10) i després $\{1, 4\}$ (pes total 9).

El problema es pot resoldre fàcilment en temps $O(n!n)$ provant totes les permutacions possibles de n persones. Tanmateix, podem utilitzar la programació dinàmica per obtenir un algorisme de temps $O(2^n n)$ més eficient. La idea és calcular per a cada subconjunt de persones dos valors: el nombre mínim de viatges necessaris i el pes mínim de les persones que viatgen en l'últim grup.

Sigui $\text{weight}[p]$ el pes de la persona p . Definim dues funcions: $\text{rides}(S)$ és el nombre mínim de viatges per a un subconjunt S , i $\text{last}(S)$ és el pes mínim de l'últim viatge. Per exemple, en l'escenari anterior

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{i} \quad \text{last}(\{1, 3, 4\}) = 5,$$

perquè els viatges òptims són $\{1, 4\}$ i $\{3\}$, i el segon viatge té un pes de 5. Per descomptat, el nostre objectiu final és calcular el valor de $\text{rides}(\{0 \dots n-1\})$.

Podem calcular els valors de les funcions recurrents de manera directa i després aplicar la programació dinàmica. La idea és passar per totes les persones que pertanyen a S i triar de manera òptima l'última persona p que entra a l'ascensor. Cadascuna d'aquestes opcions genera un subproblema per a un subconjunt de persones més petit. Si es dona $\text{last}(S \setminus p) + \text{weight}[p] \leq x$, podem afegir p a l'últim viatge. En cas contrari, haurem de reservar un viatge nou que inicialment només contingui p .

Per implementar la programació dinàmica, declarem un vector

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

que conté per a cada subconjunt S un parell $(\text{rides}(S), \text{últim}(S))$. Establim el valor per al grup buit de la següent manera:

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

Aleshores, podem omplir el vector de la següent manera:

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // add p to an existing ride
                option.second += weight[p];
            } else {
                // reserve a new ride for p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Observeu que el bucle anterior garanteix que, per a dos subconjunts S_1 i S_2 tals que $S_1 \subset S_2$, tractem S_1 abans que S_2 . És a dir, els valors de programació dinàmica es calculen en l'ordre correcte.

Comptar subconjunts

El nostre darrer problema en aquest capítol és el següent: sigui $X = \{0 \dots n-1\}$, i assignem a cada subconjunt $S \subset X$ un enter $\text{value}[S]$. La nostra tasca és calcular per cada S la suma dels valors dels subconjunts de S , és a dir,

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

Per exemple, suposem que $n = 3$ i els valors són els següents:

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

En aquest cas, per exemple,

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

Com que hi ha un total de 2^n subconjunts, una solució possible és passar per tots els parells de subconjunts en $O(2^{2n})$ temps. Tanmateix, utilitzant la

programació dinàmica, podem resoldre el problema en temps $O(2^n n)$. La idea és centrar-se en les sumes on els elements que es poden eliminar de S estan restringits.

Sigui $\text{partial}(S, k)$ la suma de valors dels subconjunts de S amb la restricció que només els elements $0 \dots k$ es poden eliminar de S . Per exemple,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

perquè només podem eliminar els elements $0 \dots 1$. Podem calcular valors de sum fent servir valors de partial , perquè

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Els casos base de la funció són

$$\text{partial}(S, -1) = \text{value}[S],$$

perquè en aquest cas no es pot eliminar cap element de S . Per al cas general podem fer servir la recurrència següent:

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

Aquí ens centrem en l'element k . Si $k \in S$, tenim dues opcions: podem mantenir k a S o eliminar-lo de S .

Hi ha una manera especialment intel·ligent d'implementar el càlcul de sumes. Podem declarar un vector

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

que conté la suma de cada subconjunt. El vector s'inicia de la següent manera:

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

Aleshores, podem omplir el vector de la següent manera:

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

Aquest codi calcula els valors de $\text{partial}(S, k)$ per a $k = 0 \dots n - 1$ al vector sum . Com que $\text{partial}(S, k)$ sempre es basa en $\text{partial}(S, k - 1)$, podem reutilitzar el vector sum , la qual cosa dóna una implementació molt eficient.

Part II

Algorismes de grafs

Capítol 11

Introducció als grafs

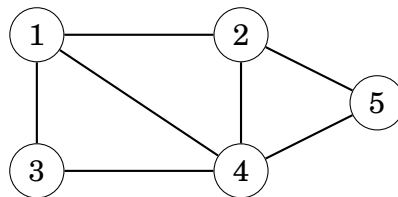
Molts problemes de programació es poden resoldre modelant el problema com un problema de grafs i fent servir l'algorisme de grafs adequat. Un exemple típic de grafs és la xarxa de carreteres i ciutats d'un país. De vegades, però, el graf està amagat dins del problema i pot ser difícil detectar-lo.

Aquesta part del llibre tracta els algorismes de grafs, especialment centrant-se en temes importants en la programació competitiva. En aquest capítol, repassem conceptes relacionats amb els grafs i estudiem diferents maneres de representar els grafs en algorismes.

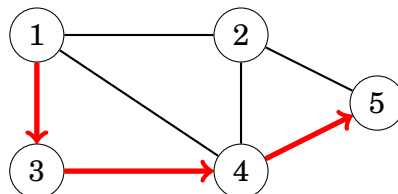
11.1 Vocabulari de grafs

Un **graf** consta de **nodes** i **arestes**. En aquest llibre, la variable n indica el nombre de nodes en un graf i la variable m indica el nombre d'arestes. Els nodes es numeren fent servir nombres $1, 2, \dots, n$.

Per exemple, el graf següent consta de 5 nodes i 7 arestes:



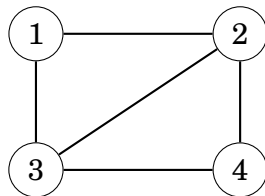
Un **camí** condueix des del node a al node b a través de les arestes del graf. La **longitud** d'un camí és el nombre d'arestes que té. Per exemple, el graf anterior conté un camí $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ de longitud 3 des del node 1 fins al node 5:



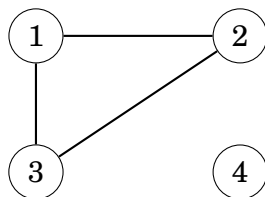
Un camí és un **cicle** si el primer i l'últim node són el mateix. Per exemple, el graf anterior conté un cicle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. Un camí és **simple** si cada node apareix com a màxim una vegada al camí.

Connectivitat

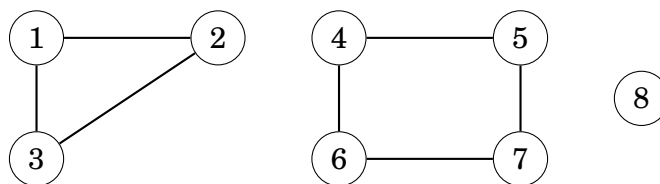
Un graf és **connex** si hi ha un camí entre dos nodes qualsevol. Per exemple, el graf següent és connex:



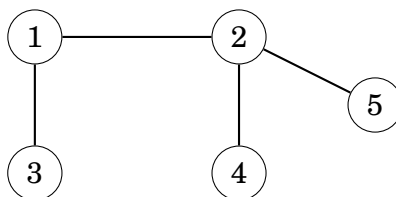
El graf següent no és connex, perquè no és possible anar del node 4 a cap altre node:



Les parts connexes d'un graf s'anomenen **components connexes**. Per exemple, el graf següent conté tres components connexes: {1, 2, 3}, {4, 5, 6, 7} i {8}.

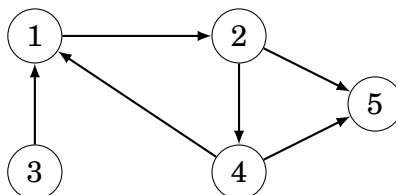


Un **arbre** és un graf connex que consta de n nodes i $n - 1$ arestes. Entre dos nodes qualsevol de l'arbre hi ha un camí únic. Per exemple, el graf següent és un arbre:



Arestes dirigides

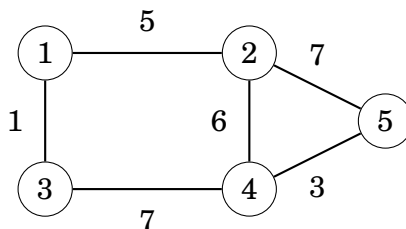
Un graf és **dirigit** si les arestes només es poden recórrer en una direcció. Per exemple, el graf següent és un graf dirigit:



El graf anterior conté un camí $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ des del node 3 fins al node 5, però no hi ha cap camí des del node 5 fins al node 3.

Arestes amb pesos

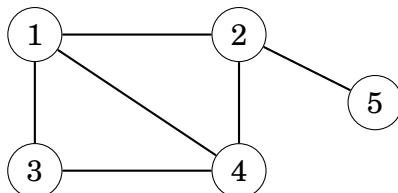
Un graf té **pesos** si a cada aresta se li assigna un **pes**. Els pesos s'interpreten sovint com a longituds de l'aresta. Per exemple, el graf següent és un graf amb pesos:



La longitud d'un camí d'un graf amb pesos és la suma dels pesos de les arestes del camí. Per exemple, al graf anterior, la longitud del camí $1 \rightarrow 2 \rightarrow 5$ és 12 i la longitud del camí $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ és 11. Aquest darrer camí és el camí **més curt** des del node 1 fins al node 5.

Veïns i graus

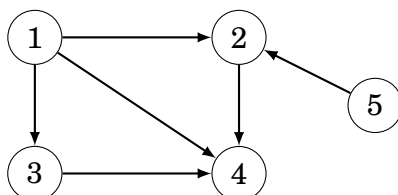
Dos nodes són **veïns** o **adjacents** si hi ha una aresta entre ells. El **grau** d'un node és el nombre d'arestes incidents al node, que coincideix amb el nombre de nodes veïns si el graf és simple. Per exemple, al graf següent, els veïns del node 2 són 1, 4 i 5, de manera que el seu grau és 3.



La suma de graus d'un graf és sempre $2m$, on m és el nombre d'arestes, perquè cada aresta té dos extrems, i per tant incrementa el grau de dos nodes en una unitat. Per aquest motiu, la suma dels graus sempre és un nombre parell.

Un graf és **regular** si el grau de cada node és una constant d . Un graf és **complet** si el graf conté una aresta per cada parell de nodes i , per tant, cada node té grau $n - 1$.

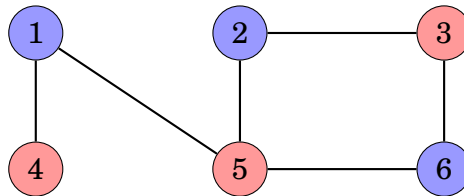
En un graf dirigit, el **grau d'entrada** d'un node és el nombre d'arestes que apunten al node, i el **grau de sortida** d'un node és el nombre d'arestes que surten del node. Per exemple, al graf següent, el grau d'entrada del node 2 és 2 i el grau de sortida del node 2 és 1.



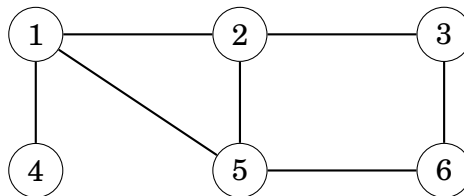
Coloracions

Una **coloració** d'un graf és una assignació de nodes a colors de manera que no hi hagi dos nodes adjacents amb el mateix color.

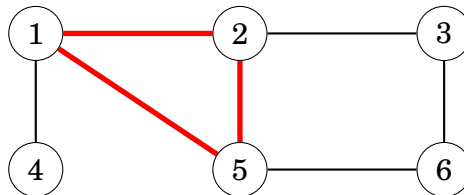
Un graf és **bipartit** si és possible acolorir-lo amb dos colors. Es pot demostrar que un graf és bipartit exactament quan no conté cap cicle amb un nombre senar d'arestes. Per exemple, el graf $[[[11]]]$ és bipartit, perquè es pot acolorir de la següent manera:



Tanmateix, el graf

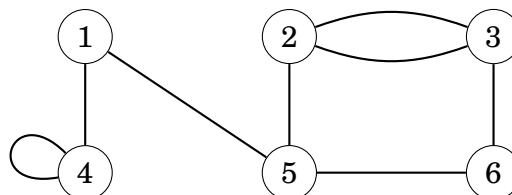


no és bipartit, perquè no és possible acolorir el següent cicle de tres nodes amb dos colors:



Grafs simples

Un graf és **simple** si cap aresta comença i acaba al mateix node, i no hi ha múltiples arestes entre dos nodes. Sovint assumim que els grafs són simples. Per exemple, el graf següent *no* és simple:



11.2 Representació de grafs

Hi ha diverses maneres de representar grafs en algorismes. L'elecció d'una estructura de dades depèn de la mida del graf i de la forma en què l'algorisme el processa. A continuació mostrarem tres representacions comunes.

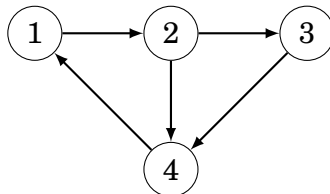
Llista d'adjacència

Per a representar un graf com a llista d'adjacència, assignem a cada node x del graf una **llista d'adjacència** que consta dels nodes als quals hi ha una aresta des de x . Les llistes d'adjacència són la manera més popular de representar grafs, i la majoria dels algorismes es poden implementar eficientment amb llistes d'adjacència.

Una manera convenient d'emmagatzemar les llistes d'adjacència és declarar un *array* de vectors de la manera següent:

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

La constant N s'escull de manera que es pugui emmagatzemar totes les llistes d'adjacència. Per exemple, el graf



es pot emmagatzemar de la següent manera:

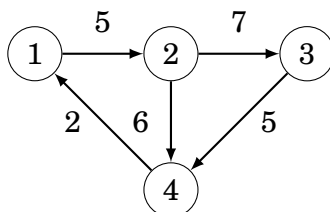
```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Si el graf no està dirigit, podem fer servir una representació semblant, però afegim cada aresta en ambdues direccions.

Per a un graf amb pesos, l'estructura es pot ampliar de la següent manera:

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

En aquest cas, la llista d'adjacència del node a conté el parell (b, w) sempre quan hi ha una aresta des del node a fins al node b amb pes w . Per exemple, el graf



es pot emmagatzemar de la següent manera:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

L'avantatge d'utilitzar llistes d'adjacència és que podem trobar de manera eficient els nodes als quals ens podem moure des d'un node determinat a través d'una aresta. Per exemple, el següent bucle passa per tots els nodes als quals ens podem moure des del node s :

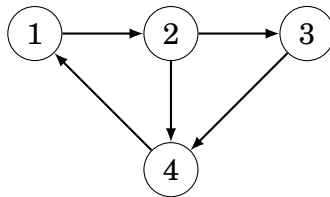
```
for (auto u : adj[s]) {
    // fer coses amb el node 'u'
}
```

Matriu d'adjacència

Una **matriu d'adjacència** és una matriu bidimensional que indica quines arestes pertanyen al graf. Amb una matriu d'adjacència podem comprovar de manera eficient si hi ha una aresta entre dos nodes. La matriu es pot emmagatzemar com un *array* de vectors

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

on cada valor $adj[a][b]$ indica si el graf conté una aresta des del node a fins al node b . Si l'aresta s'inclou al graf, llavors $adj[a][b] = 1$, i en cas contrari $adj[a][b] = 0$. Per exemple, el graf



es pot representar de la següent manera:

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

Si el graf té pesos, la representació amb matriu d'adjacència s'extén de manera que la matriu contingui el pes a l'aresta si l'aresta existeix. Utilitzant aquesta representació, el graf



es correspon amb la matriu següent:

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

L'inconvenient de la representació amb matrius d'adjacència és que la matriu conté n^2 elements, i normalment la majoria d'ells són zero. Per aquest motiu, la representació no es pot fer servir si el graf és gran.

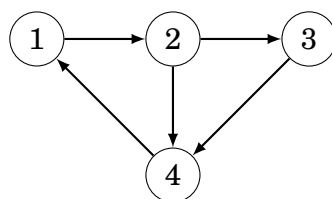
Llista d'arestes

Una **llista d'arestes** conté totes les arestes d'un graf en un cert ordre. Aquesta és una manera convenient de representar un graf si l'algorisme processa totes les arestes del graf i no cal trobar les arestes que comencen en un node determinat.

La llista d'arestes es pot emmagatzemar en un vector

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

on cada parell (a,b) indica que hi ha una aresta des del node a fins al node b . Així, el graf



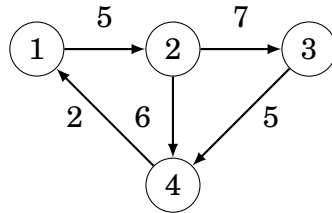
es pot representar de la següent manera:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Si el graf té pesos, l'estructura es pot ampliar de la següent manera:

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

Cada element d'aquesta llista té la forma (a,b,w) , el que significa que hi ha una aresta des del node a fins al node b amb pes w . Per exemple, el graf



es pot representar com segueix¹:

```
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});
```

¹En alguns compiladors antics, és necessari fer servir la funció `make_tuple` en lloc de les claus (per exemple, `make_tuple(1,2,5)` en lloc de `{1,2,5}`).

Capítol 12

Recorreguts en grafs

Aquest capítol tracta dos algorismes fonamentals de grafs: la cerca en profunditat i la cerca en amplada. Ambdós algorismes reben un node inicial del graf i visiten tots els nodes als quals es pot accedir des del node inicial. La diferència entre els algorismes és l'ordre en què visiten els nodes.

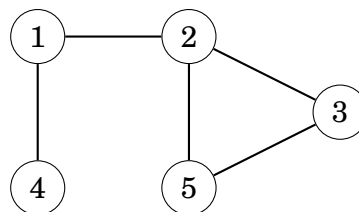
12.1 Cerca en profunditat

Cerca en profunditat (*depth-first search*, DFS) és una tècnica senzilla per a recórrer grafs. L'algorisme comença en un node inicial, i continua per tots els altres nodes als quals es pot accedir des del node inicial fent servir les arestes del graf.

La cerca en profunditat sempre recorre un únic camí del graf mentre trobi nodes nous per explorar. Després d'això, torna als nodes anteriors i comença a explorar altres parts del graf. L'algorisme té control dels nodes que ja han estat visitats, de manera que processa cada node només una vegada.

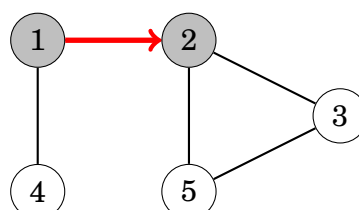
Exemple

Posem com a exemple la cerca en profunditat del graf següent:

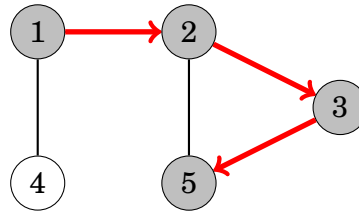


Podem començar la cerca en qualsevol node del graf, per exemple, el node 1.

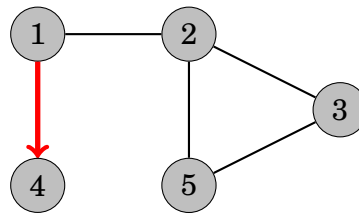
La cerca passa primer pel node 2:



Després d'això, visita els nodes 3 i 5:



Els veïns del node 5 són el 2 i el 3, però la cerca ja ha visitat ambdós, de manera que tornem als nodes anteriors. També hem vist els veïns dels nodes 3 i 2, per la qual cosa el següent moviment és del node 1 al node 4:



Després d'això, la cerca finalitza perquè ha visitat tots els nodes.

La complexitat temporal de la cerca en profunditat és $O(n + m)$ on n és el nombre de nodes i m és el nombre d'arestes, perquè l'algorisme processa cada node i aresta una sola vegada.

Implementació

La cerca en profunditat es pot implementar còmodament mitjançant recursivitat. La següent funció `dfs` comença una cerca en profunditat en un node determinat. La funció suposa que el graf s'emmagatzema com a llistes d'adjacència en un vector

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

i també manté un vector

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

per conèixer els nodes visitats. Inicialment, cada valor del vector és `false`, i quan la cerca arriba al node s , el valor de `visited[s]` es converteix en `true`. La funció es pot implementar de la següent manera:

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

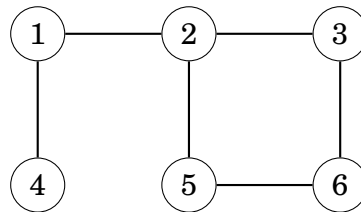
12.2 Cerca en amplada

La **cerca en amplada** (*breadth-first search*, BFS) visita els nodes en ordre creixent a la seva distància des del node inicial. Així, podem calcular la distància des del node inicial fins a la resta de nodes mitjançant la cerca en amplada. Tanmateix, la cerca en amplada és més difícil d'implementar que la cerca en profunditat.

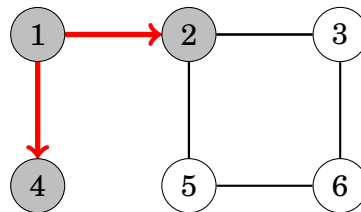
La cerca en amplada passa per tots els nodes d'un nivell abans de considerar els nodes del nivell següent. Primer, la cerca explora tots els nodes a distància 1 del node inicial, després els nodes a distància 2, i així successivament. Aquest procés continua fins que s'han visitat tots els nodes.

Exemple

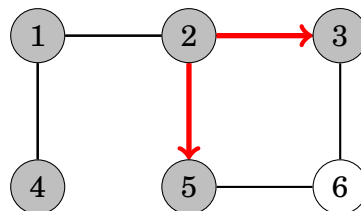
Considerem com la cerca en amplada processa el graf següent:



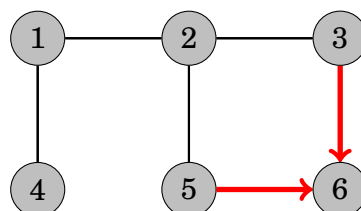
Suposem que la cerca comença al node 1. En primer lloc, processem tots els nodes als quals es pot arribar des del node 1 mitjançant una única aresta:



Després d'això, passem als nodes 3 i 5:



Finalment, visitem el node 6:



Ara hem calculat les distàncies des del node inicial fins a tots els nodes del graf. Les distàncies són les següents:

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

A l'igual que en la cerca en profunditat, la complexitat temporal de la cerca en amplada és $O(n + m)$, on n és el nombre de nodes i m és el nombre d'arestes.

Implementació

La cerca en profunditat és més difícil d'implementar que la cerca en profunditat, perquè l'algoritme visita els nodes de parts diferents del graf. Una implementació típica es basa en una cua que conté els nodes. En cada pas processem un node de la cua.

El següent codi suposa que el graf s'emmagatzema com a llistes d'adjacència i manté les estructures de dades següents:

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

La cua q conté els nodes que s'han de processar en ordre de distància creixent. Els nous nodes sempre s'afegeixen al final de la cua, i el node al començament de la cua és el següent node a processar. El vector `visited` indica quins nodes ja s'han visitat, i el vector `distance` contindrà les distàncies des del node inicial a tots els nodes del graf.

La cerca es pot implementar de la següent manera, començant pel 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);  
    }  
}
```

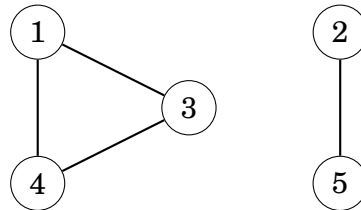
12.3 Aplicacions

Fent servir els algorismes de recorreguts en grafs podem comprovar moltes propietats dels mateixos. Podem fer servir tant la cerca en profunditat com la cerca en amplada, però a la pràctica, la cerca en profunditat és una millor opció perquè és més fàcil d'implementar. En les aplicacions següents assumirem que el graf no està orientat.

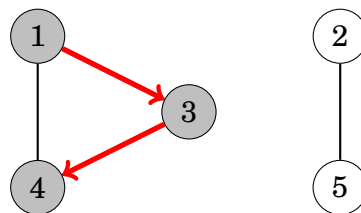
Comprovació de connectivitat

Un graf és connex si hi ha un camí entre dos nodes qualsevol del graf. Així, podem comprovar si un graf és connex començant per un node arbitrari i esbrinant si podem arribar a tots els altres nodes.

Per exemple, en el graf



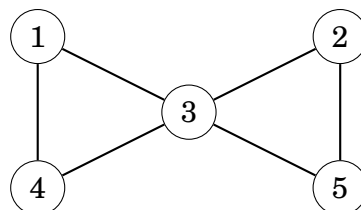
es pot veure que una cerca en profunditat des del node 1 visita els nodes següents:



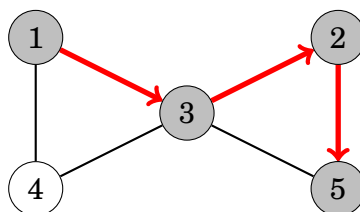
Com que la cerca no ha visitat tots els nodes, en conclouem que el graf no és connex. De manera semblant, podem trobar totes les components connexes d'un graf iterant pels nodes i començant una nova cerca en profunditat cada cop que trobem un node que no pertanyi a cap component connexa..

Trobar cicles

Un graf conté un cicle si al recórrer el graf trobem un node amb un veí que ja ha estat visitat, i que no sigui el node anterior en el camí actual. Per exemple, el graf



conté dos cicles i podem trobar un d'ells de la següent manera:



Després de passar del node 2 al node 5 observem que el veí 3 del node 5 ja ha estat visitat. Així, el graf conté un cycle que passa pel node 3, per exemple, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

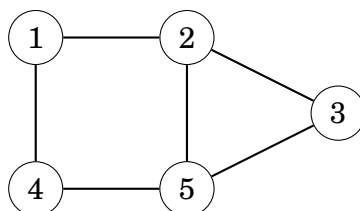
Una altra manera d'esbrinar si un graf conté un cycle és calcular el nombre de nodes i arestes de cada component connexa. Si un component conté c nodes i cap cycle, ha de contenir exactament $c - 1$ arestes (ha de ser un arbre). Si hi ha c o més arestes, la component sens dubte conté un cycle.

Comprovació de bipartitisme

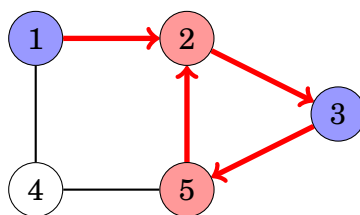
Un graf és bipartit si els seus nodes es poden acolorir amb dos colors de manera que no hi hagi nodes adjacents amb el mateix color. Comprovar si un graf és bipartit mitjançant algorismes de recorregut de grafs és sorprenentment fàcil.

La idea és pintar el node inicial de blau, tots els seus veïns de vermell, tots els seus veïns de blau, etc. Si en algun moment de la cerca observem que dos nodes adjacents tenen el mateix color, això vol dir que el graf no és bipartit. En cas contrari, el graf és bipartit i hem trobat una coloració.

Per exemple, el graf



no és bipartit, perquè si fem una cerca des del node 1 veiem el següent:



Observem que el color dels nodes 2 i 5 és vermell, tot i que són adjacents. Per tant, el graf no pot ser bipartit.

Aquest algorisme sempre funciona, perquè quan només hi ha dos colors disponibles, el color del node inicial d'un component determina els colors de tots els altres nodes del component, sense importar si acolorim el node inicial vermell o blau.

Tingueu en compte que en el cas general, és difícil esbrinar si els nodes d'un graf es poden acolorir amb k colors de manera que cap node adjacent tingui el

mateix color. Fins i tot quan $k = 3$, és sap que el problema és NP-difícil i no es coneix cap algorisme eficient.

Capítol 13

Camins més curts

Trobar el camí més curt entre dos nodes d'un graf és un problema important que té moltes aplicacions pràctiques. Per exemple, donada una xarxa de carreteres, calcula la longitud més curta possible d'una ruta entre dues ciutats, tenint en compte les longituds de les carreteres.

En un graf sense pesos, la longitud d'un camí és igual al nombre d'arestes, i podem fer servir la cerca en amplada per trobar el camí més curt. Tanmateix, en aquest capítol ens centrem en els grafs amb pesos on necessitem algorismes més sofisticats per trobar els camins mínims.

13.1 Algorisme de Bellman–Ford

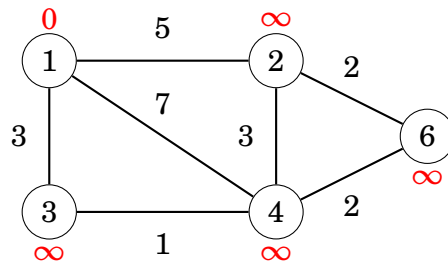
L'**algorisme de Bellman–Ford**¹ troba els camins més curts des d'un node inicial a tots els nodes del graf. L'algorisme pot processar tot tipus de grafs, sempre que el graf no contingui un cicle de longitud negativa. L'algorisme és capaç de detectar si el graf té cicles de longitud negativa.

L'algorisme manté un vector amb les millors distàncies conegudes des del node inicial fins a tots els nodes del graf. Inicialment, la distància al node inicial és 0 i la distància a tots els altres nodes és infinita. L'algorisme redueix les distàncies trobant arestes que escurcen els camins fins que no és possible reduir cap distància.

Exemple

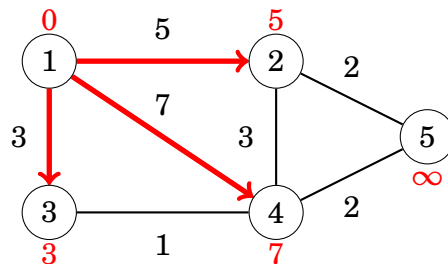
Mostrem com funciona l'algorisme de Bellman–Ford en el graf següent:

¹L'algorisme rep el nom de R.E. Bellman i L.R. Ford que el van publicar de manera independentment els anys 1958 i el 1956 [5, 24].

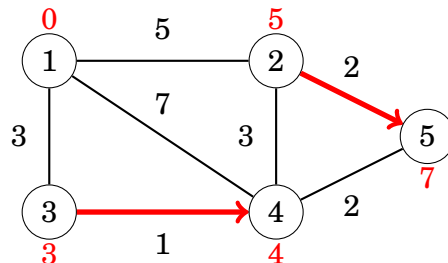


A cada node del graf li assignem una distància. Inicialment, la distància al node inicial és 0 i la distància a tots els altres nodes és infinita.

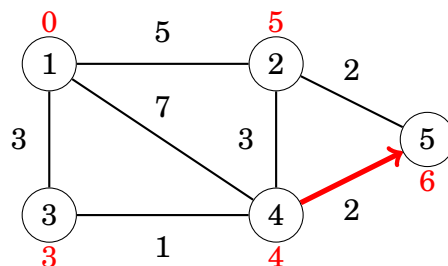
L'algorisme cerca arestes que redueixin distàncies. En primer lloc, totes les arestes del node 1 redueixen les distàncies:



Després d'això, les arestes $2 \rightarrow 5$ i $3 \rightarrow 4$ redueixen les distàncies:

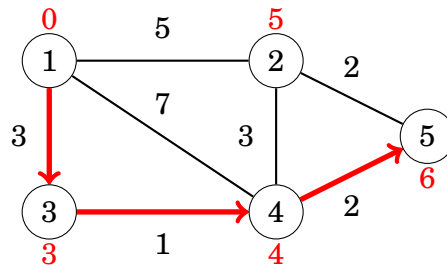


Finalment, hi ha un canvi més:



Després d'això, cap aresta pot reduir cap distància. Això vol dir que les distàncies són definitives i hem calculat correctament les distàncies més curtes des del node inicial fins a tots els nodes del graf.

Per exemple, la distància més curta del node 1 al node 5 és 3 i es correspon amb al camí següent:



Implementació

La següent implementació de l'algorisme de Bellman-Ford determina les distàncies més curtes des d'un node x a tots els nodes del graf. El codi suposa que el graf s'emmagatzema com una llista de arestes `edges` que consta de tuples de la forma (a, b, w) , el que significa que hi ha una aresta des del node a fins al node b amb pes w .

L'algorisme consta de $n - 1$ rondes, i a cada ronda l'algoritme passa per totes les arestes del graf i intenta reduir les distàncies. L'algorisme construeix una vector `distance` que conté les millors distàncies conegudes de x a tots els nodes del graf. La constant `INF` denota una distància infinita.

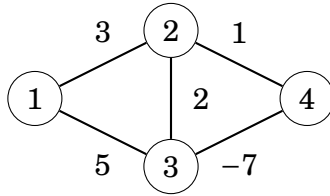
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

La complexitat temporal de l'algorisme és $O(nm)$, perquè l'algoritme consta de $n - 1$ rondes i itera per totes les m arestes durant una ronda. Si no hi ha cicles negatius al graf, totes les distàncies són finals després de $n - 1$ rondes, perquè els camins mínims no poden repetir nodes, i per tant contenen com a molt $n - 1$ arestes.

A la pràctica, sovint no és necessari fer $n - 1$ rondes per a trobar les distàncies definitives. Podem fer l'algorisme més eficient si l'aturem quan no reduim cap distància durant una ronda.

Cicles negatius

L'algorisme de Bellman-Ford també es pot fer servir per comprovar si el graf conté un cicle de longitud negativa. Per exemple, el graf



conté un cycle negatiu $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ de longitud -4 .

Si el graf conté un cycle negatiu, podem escurçar infinites vegades qualsevol camí que contingui el cycle repetint el cycle una i altra vegada. Per tant, el concepte de camí més curt no té sentit en aquesta situació.

Podem detectar cicles negatius executant l'algorisme de Bellman-Ford exactament n rondes. Si l'última ronda redueix alguna distància, el graf conté un cycle negatiu. Observeu que aquest algorisme es pot fer servir per cercar un cycle negatiu a tot el graf independentment del node inicial.

Algorisme SPFA

L'algorisme **SPFA** ("Shortest Path Faster Algorithm") [20] és una variant de l'algorisme de Bellman-Ford, que sovint és més eficient que l'algorisme original. L'algoritme SPFA no passa per totes les arestes a cada ronda, sinó que tria les arestes que s'han d'examinar d'una manera més intel·ligent.

L'algorisme manté una cua de nodes que es poden fer servir per reduir les distàncies. En primer lloc, l'algoritme afegeix el node inicial x a la cua. A continuació, l'algorisme considera el primer node a de la cua, i comprova si les arestes de la forma $a \rightarrow b$ redueixen la distància, i si és el cas, afegeix el node b a la cua.

L'eficiència de l'algorisme SPFA depèn de l'estructura del graf: l'algorisme és sovint eficient, però la seva complexitat temporal en el pitjor dels casos segueix sent $O(nm)$ i és possible crear entrades que fan que l'algorisme sigui tan lent com l'algorisme original de Bellman-Ford.

13.2 Algorisme de Dijkstra

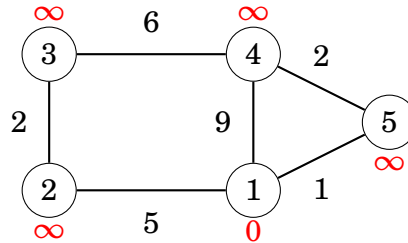
Algorisme de Dijkstra² troba els camins més curts des del node inicial fins a tots els nodes del graf, a l'igual que l'algorisme de Bellman-Ford. L'avantatge de l'algorisme de Dijkstra és que és més eficient i es pot fer servir per processar grafs grans. Tanmateix, l'algorisme requereix que no hi hagi arestes de pes negatius al graf.

Igual que l'algorisme de Bellman-Ford, l'algoritme de Dijkstra manté les distàncies als nodes i les redueix durant la cerca. L'algorisme de Dijkstra és eficient, perquè només processa cada aresta del graf una vegada, fent servir el fet que no hi ha arestes negatives.

²E. W. Dijkstra va publicar l'algorisme el 1959 [14]; tanmateix, el seu article original no esmenta com implementar l'algorisme de manera eficient.

Exemple

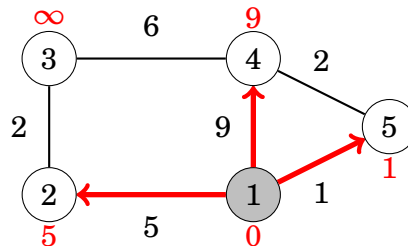
Considerem com funciona l'algorisme de Dijkstra al graf següent quan el node inicial és el node 1:



A l'igual que en l'algorisme de Bellman-Ford, inicialment la distància al node inicial és 0 i la distància a tots els altres nodes és infinita.

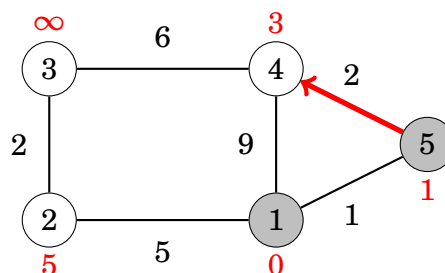
A cada pas, l'algorisme de Dijkstra selecciona aquell node que encara no s'ha processat i que està a distància mínima. El primer d'aquests nodes és el node 1 a distància 0.

Quan es selecciona un node, l'algorisme passa per totes les arestes que surten del node i redueix les distàncies:

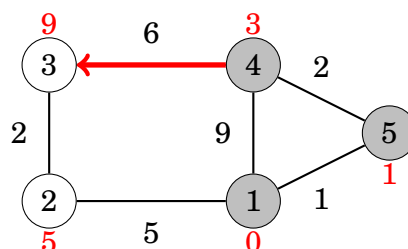


En aquest cas, les arestes del node 1 redueixen les distàncies als nodes 2, 4 i 5, que passen a ser ara 5, 9 i 1.

El següent node a processar és el node 5 a distància 1. Això redueix la distància al node 4 de 9 a 3:

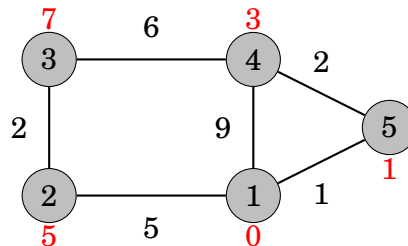


Després d'això, el següent node és el node 4, i fa que la distància al node 3 sigui 9:



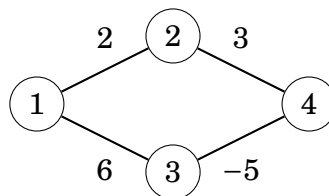
Una propietat notable de l'algorisme de Dijkstra és que sempre que treballem amb un node trobem la seva distància final. Per exemple, en aquest punt de l'algorisme, les distàncies 0, 1 i 3 són les distàncies finals als nodes 1, 5 i 4.

Després d'això, l'algorisme processa els dos nodes restants i les distàncies resultants són les següents:



Arestes negatives

L'eficiència de l'algorisme de Dijkstra es basa en el fet que el graf no conté arestes negatives. Si hi ha una aresta negativa l'algorisme pot donar resultats incorrectes. Per exemple:



El camí més curt des del node 1 al node 4 és $1 \rightarrow 3 \rightarrow 4$ i la seva longitud és 1. Tanmateix, l'algorisme de Dijkstra troba el camí $1 \rightarrow 2 \rightarrow 4$ seguint les arestes de pes mínim. L'algorisme no té en compte que en l'altre camí el pes -5 serveix per compensar el pes 6.

Implementació

La següent implementació de l'algorisme de Dijkstra calcula les distàncies mínimes des d'un node x a els altres nodes del graf. El graf s'emmagatzema com a llistes d'adjacència de manera que $\text{adj}[a]$ conte un parell (b, w) quan hi ha una aresta del node a fins al node b amb pes w .

Una implementació eficient de l'algorisme de Dijkstra requereix trobar de manera eficient el node de distància mínima que no s'ha processat encara. L'estructura de dades adequada és una cua de prioritats que conté els nodes ordenats per distància. Amb una cua de prioritats trobem el següent node a processar en temps logarítmic.

Al codi següent, la cua de prioritat q conté els parells de la forma $(-d, x)$ per a indicar que la distància al node x és d . El vector `distance` conté la distància a cada node, i el vector `processed` indica si s'ha processat un node. Inicialment la distància és 0 a x i ∞ a tots els altres 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});
        }
    }
}
}

```

Tingueu en compte que la cua de prioritat conté distàncies *negatives* als nodes. La raó d'això és que la versió predeterminada de la cua de prioritats C++ troba els elements màxims, mentre que volem trobar els elements mínims. Mitjançant l'ús de distàncies negatives, podem utilitzar directament la cua de prioritat per defecte³. Tingueu en compte també que pot haver-hi diverses instàncies del mateix node a la cua de prioritats; tanmateix, només processarem la instància amb distància mínima.

La complexitat temporal de la implementació anterior és $O(n + m \log m)$, perquè l'algoritme passa per tots els nodes del graf i afegeix, per cada aresta, com a màxim una distància a la cua de prioritats.

13.3 Algorisme de Floyd-Warshall

L'algorisme de **Floyd-Warshall**⁴ proporciona una manera alternativa d'abordar el problema de trobar els camins més curts. Aquest, a diferència dels altres algorismes d'aquest capítol, troba tots els camins més curts entre tots els nodes en una sola execució.

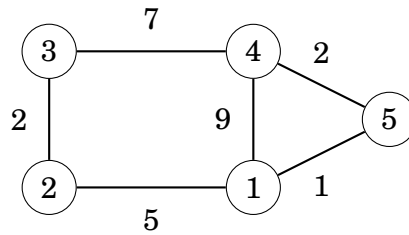
L'algorisme manté una matriu que conté les distàncies entre els nodes. Al principi, les distàncies es calculen únicament amb arestes directes entre els nodes, i a continuació, l'algorisme redueix les distàncies utilitzant nodes intermedis en camins.

Exemple

Considerem com funciona l'algorisme de Floyd-Warshall al graf següent:

³Per descomptat, també podríem declarar la cua de prioritat com al capítol 4.5 i fer servir distàncies positives, però la implementació seria una mica més llarga.

⁴L'algorisme rep el nom de R.W. Floyd i S. Warshall que el van publicar de manera independent el 1962 [23, 70].



Inicialment, la distància de cada node a si mateix és 0, i la distància entre els nodes a i b és x si hi ha una aresta entre els nodes a i b amb pes x . Totes les altres distàncies són infinites.

En aquest graf, la matriu inicial és la següent:

	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

L'algorisme consta de rondes consecutives. A cada ronda, l'algoritme selecciona un nou node que pot actuar com a node intermedi en els camins a partir d'ara, i les distàncies es redueixen mitjançant aquest node.

A la primera ronda, el node 1 és el nou node intermedi. Hi ha un nou camí entre els nodes 2 i 4 de longitud 14, perquè el node 1 els connecta. També hi ha un nou camí entre els nodes 2 i 5 amb longitud 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

A la segona ronda, el node 2 és el nou node intermedi. Això crea nous camins entre els nodes 1 i 3 i entre els nodes 3 i 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

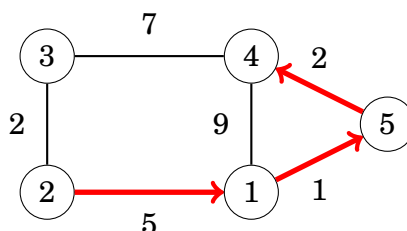
A la tercera ronda, el node 3 és el nou node intermedi. Hi ha un nou camí entre els nodes 2 i 4:

	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

L'algorisme continua així, fins que tots els nodes han estat designats nodes intermedis. Un cop acabat l'algorisme, la matriu conté les distàncies mínimes entre dos nodes qualsevol:

	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

Per exemple, la matriu ens indica que la distància més curta entre els nodes 2 i 4 és 8. Això correspon al camí següent:



Implementació

L'avantatge de l'algorisme de Floyd-Warshall és que és fàcil d'implementar. El codi següent construeix una matriu de distàncies on $distance[a][b]$ és la distància més curta entre els nodes a i b . Primer, l'algorisme inicialitza $distance$ fent servir la matriu d'adjacència adj del graf:

```
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;
    }
}
```

Després d'això, les distàncies més curtes es poden trobar de la següent manera:

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

```
        for (int j = 1; j <= n; j++) {  
            distance[i][j] = min(distance[i][j],  
                                distance[i][k]+distance[k][j]);  
        }  
    }  
}
```

La complexitat temporal de l'algorisme és $O(n^3)$, perquè conté tres bucles niats que passen per tots els nodes del graf.

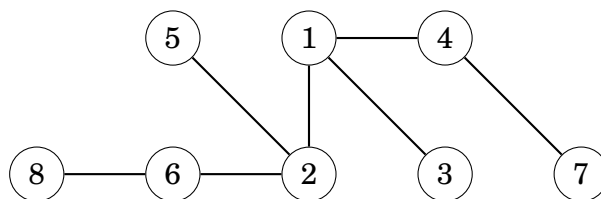
Com que la implementació de l'algorisme de Floyd-Warshall és senzilla, aquest algoritme pot ser una bona opció encara que només sigui necessari trobar un únic camí mínim del graf. Tanmateix, l'algorisme només pot fer-se servir quan el graf és tan petit que la complexitat cúbica resultant és acceptable.

Capítol 14

Algorismes d'arbres

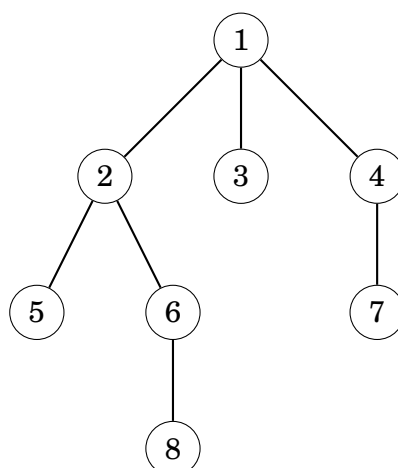
Un **arbre** és un graf connex i acíclic que consta de n nodes i $n - 1$ arestes. Quan eliminem qualsevol aresta d'un arbre el dividim en dos components connexes, i quan afegim qualsevol aresta a un arbre creem un cicle. A més, sempre hi ha un camí únic entre dos nodes qualsevol d'un arbre.

Per exemple, l'arbre següent consta de 8 nodes i 7 arestes:



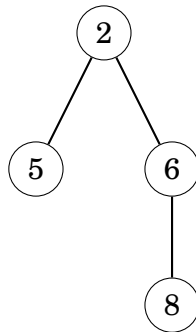
Les **fulles** d'un arbre són els nodes de grau 1, és a dir, amb només un veí. Per exemple, les fulles de l'arbre anterior són els nodes 3, 5, 7 i 8.

En un arbre **arrelat**, un dels nodes s'anomena l'**arrel** de l'arbre, i tots els altres nodes es col·loquen sota l'arrel. Per exemple, a l'arbre següent, el node 1 és el node arrel.



En un arbre arrelat, els **fills** d'un node són els seus veïns inferiors, i el **pare** d'un node és el seu veí superior. Cada node té exactament un pare, excepte l'arrel que no té cap pare. Per exemple, a l'arbre anterior, els fills del node 2 són els nodes 5 i 6, i el seu pare és el node 1.

L'estructura d'un arbre arrelat és *recursiva*: cada node de l'arbre actua com a arrel d'un **subarbre** que conté el node en si i tots els nodes que es troben als subarbres dels seus fills. Per exemple, a l'arbre anterior, el subarbre del node 2 consta dels nodes 2, 5, 6 i 8:



14.1 Recorregut d'arbres

Els algorismes generals de recorreguts de grafs es poden utilitzar per recórrer els nodes d'un arbre. Tanmateix, el recorregut d'un arbre és més fàcil d'implementar que el d'un graf general, perquè no hi ha cicles a l'arbre i no és possible arribar a un node des de múltiples direccions.

La manera típica de recórrer un arbre és iniciar una cerca en profunditat en un node arbitrari. Es pot utilitzar la funció recursiva següent:

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

La funció té dos paràmetres: el node actual s i el node anterior e . El propòsit del paràmetre e és assegurar-se que la cerca només es mou als nodes que encara no s'han visitat.

El següent codi inicia la cerca al node x :

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

En la primera crida $e = 0$, perquè no hi ha cap node anterior, i es permet avançar en qualsevol direcció de l'arbre.

Programació dinàmica

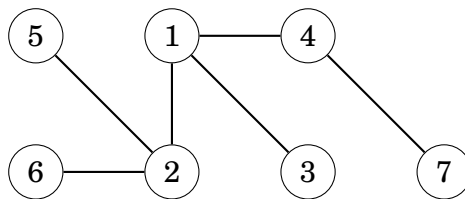
La programació dinàmica es pot fer servir per calcular informació quan recorrem un arbre. Per exemple, podem calcular en temps $O(n)$ la mida del subarbre de cada node, o la longitud del camí més llarg des de cada node a una fulla.

Com a exemple, calculem per a cada node s un valor $\text{count}[s]$: la mida del seu subarbre. El subarbre conté el node i tots els nodes dels subarbres dels seus fills, de manera que podem calcular aquesta mida recursivament amb el codi següent:

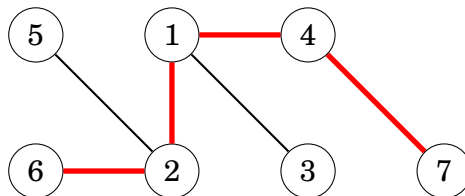
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Diàmetre

El **diàmetre** d'un arbre és la longitud màxima d'un camí entre dos nodes. Per exemple, considereu l'arbre següent:



El diàmetre d'aquest arbre és 4, i es correspon al següent camí:



Tingueu en compte que poden haver-hi diversos camins de longitud màxima. Al camí anterior, podríem substituir el node 6 pel node 5 per obtenir un altre camí de longitud 4.

A continuació mostrem dos algorismes de temps $O(n)$ que calculen el diàmetre d'un arbre. El primer algorisme es basa en la programació dinàmica, i el segon algorisme utilitza dues cerques en profunditat.

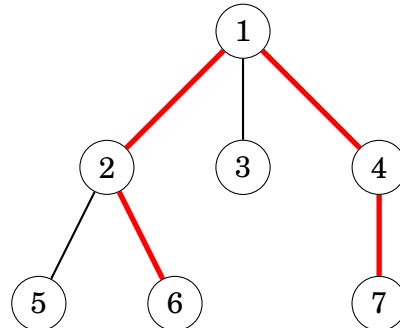
Algorisme 1

Una manera general d'abordar molts problemes dels arbres és arrelar primer l'arbre de manera arbitrària. Després d'això, podem intentar resoldre el problema per separat per a cada subarbre. El nostre primer algorisme per calcular el diàmetre es basa en aquesta idea.

Una observació important és que cada camí d'un arbre arrelat té un *punt més alt*: el node més alt que pertany al camí. Així, podem trobar per a cada node x

el camí més llarg que té x com a node més alt del camí. Un d'aquests camins correspon al diàmetre de l'arbre.

Per exemple, a l'arbre següent, el node 1 és el punt més alt del camí que correspon al diàmetre:



Calculem per a cada node x dos valors:

- $\text{toLeaf}(x)$: 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

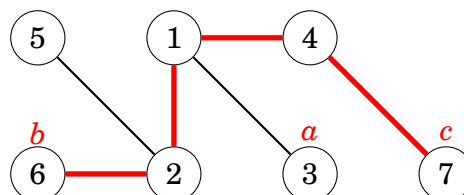
Per exemple, a l'arbre anterior, $\text{toLeaf}(1) = 2$, perquè hi ha un camí $1 \rightarrow 2 \rightarrow 6$, i $\text{maxLength}(1) = 4$, perquè hi ha un camí $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$. En aquest cas, $\text{maxLength}(1)$ és igual al diàmetre.

La programació dinàmica es pot utilitzar per calcular els valors anteriors per a tots els nodes en temps $O(n)$. Primer, per calcular $\text{toLeaf}(x)$, considerem els fills c de x i afegim un al màxim dels valors $\text{toLeaf}(c)$. A continuació, per a calcular $\text{maxLength}(x)$ escollim dos fills diferents a i b de manera que la suma $\text{toLeaf}(a) + \text{toLeaf}(b)$ sigui màxima, i afegim dos a aquesta suma.

Algorisme 2

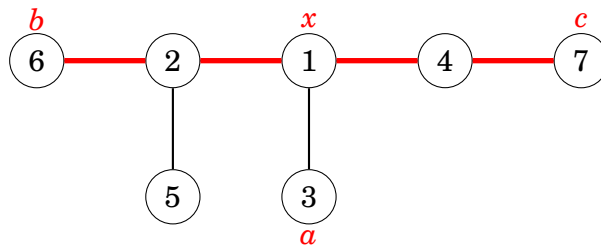
Una altra manera eficient de calcular el diàmetre d'un arbre es fer dues cerques en profunditat. Primer, triem un node arbitrari a de l'arbre i trobem el node b més llunyà de a . Aleshores, trobem el node més llunyà c de b . El diàmetre de l'arbre és la distància entre b i c .

Al graf següent, a , b i c podrien ser:



Aquest és un mètode elegant, però per què funciona?

És útil dibuixar l'arbre de manera que el camí que correspon al diàmetre sigui horitzontal i tots els altres nodes en penguin:

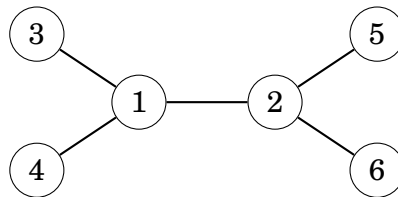


El node x indica el lloc on el camí des del node a s'uneix al camí que correspon al diàmetre. El node més allunyat de a és el node b , el node c o algun altre node que estigui almenys tan lluny del node x , i que, per tant, hagués pogut reemplaçar a b o c com a punt final del camí que es correspon al diàmetre.

14.3 Tots els camins més llargs

El problema següent és calcular per a cada node de l'arbre la longitud màxima d'un camí que comença al node. Això és una generalització del problema del diàmetre de l'arbre, perquè la major d'aquestes longituds és el diàmetre de l'arbre. Aquest problema també es pot resoldre en temps $O(n)$.

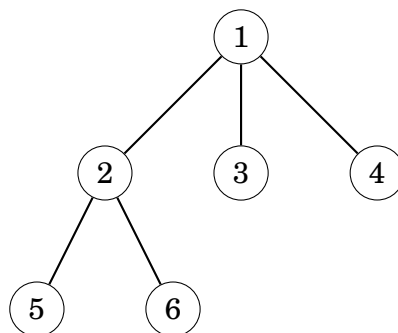
Com a exemple, considereu l'arbre següent:



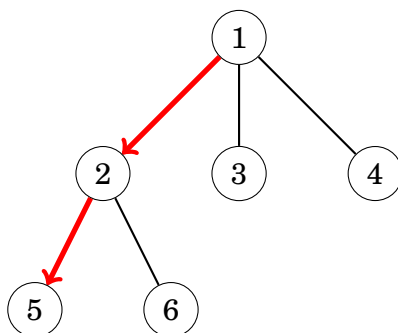
Sigui $\text{maxLength}(x)$ la longitud màxima d'un camí que comença al node x . Per exemple, a l'arbre anterior, $\text{maxLength}(4) = 3$, perquè hi ha un camí $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Aquí teniu una taula completa dels valors:

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

A l'igual que abans, un bon punt de partida per resoldre el problema és arrelar l'arbre de manera arbitrària:

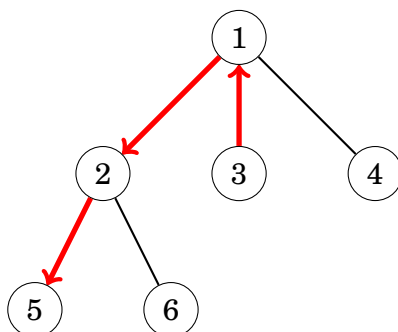


La primera part del problema és calcular per a cada node x la longitud màxima d'un camí que passa per un fill de x . Per exemple, el camí més llarg des del node 1 passa pel seu fill 2:

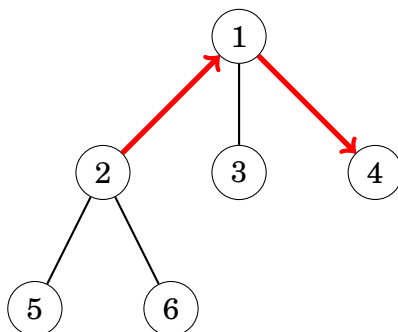


Aquesta part és fàcil de resoldre en temps $O(n)$ fent servir programació dinàmica, com abans.

La segona part del problema és calcular per a cada node x la longitud màxima d'un camí a través del seu pare p . Per exemple, el camí més llarg des del node 3 passa pel seu pare 1:



A primera vista, sembla que hauríem de triar el camí més llarg que surt de p . Tanmateix, això *no* sempre funciona, perquè el camí més llarg des de p pot passar per x . Aquí teniu un exemple d'aquesta situació:



Tot i així, podem resoldre la segona part en temps $O(n)$ emmagatzemant *dues* longituds màximes per a cada 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

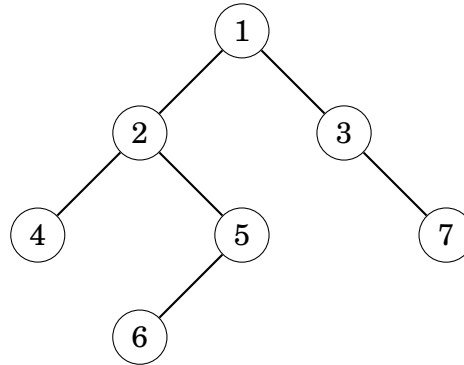
Per exemple, al graf anterior, $\text{maxLength}_1(1) = 2$ utilitzant el camí $1 \rightarrow 2 \rightarrow 5$, i $\text{maxLength}_2(1) = 1$ utilitzant el camí $1 \rightarrow 3$.

Finalment, si el camí que correspon a $\text{maxLength}_1(p)$ passa per x , arribem a la conclusió que la longitud màxima és $\text{maxLength}_2(p) + 1$, i en cas contrari la longitud màxima és $\text{maxLength}_1(p) + 1$.

14.4 Arbres binaris

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:



Els nodes d'un arbre binari tenen tres ordenacions naturals que es corresponen a maneres distintes de recórrer l'arbre recursivament:

- **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

Per a l'arbre anterior, l'ordenació dels nodes de l'arbre en pre-ordre és [1, 2, 4, 5, 6, 3, 7], en in-ordre és [4, 2, 6, 5, 1, 3, 7] i en post-ordre és [4, 6, 5, 2, 7, 3, 1].

Si coneixem l'ordenació en pre-ordre i in-ordre d'un arbre, podem reconstruir l'estructura exacta de l'arbre. Per exemple, l'arbre anterior és l'únic arbre possible amb pre-ordre [1, 2, 4, 5, 6, 3, 7] i in-ordre [4, 2, 6, 5, 1, 3, 7]. De manera similar, si tenim l'ordenació en post-ordre i in-ordre també podem determinar l'estructura d'un arbre.

Tanmateix, la situació és diferent si només coneixem l'ordenació en pre-ordre i post-ordre d'un arbre. En aquest cas, pot haver-hi més d'un arbre que coincideixi amb les ordenacions. Per exemple, els dos arbres



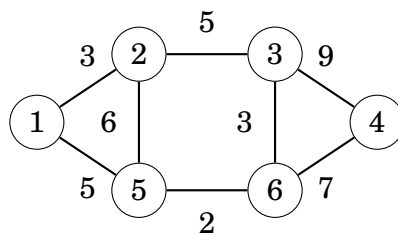
tenen pre-ordre [1, 2] i post-ordre [2, 1], però les estructures dels arbres són diferents.

Capítol 15

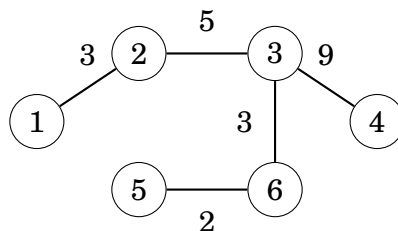
Arbres d'expansió

Un **arbre d'expansió** (*spanning tree*) d'un graf és un arbre que conté tots els nodes del graf i algunes de les arestes. Com que és un arbre, els arbres d'expansió són connexos, acíclics i existeix un sol camí entre dos nodes qualsevol. Normalment hi ha diverses maneres de construir un arbre d'expansió.

Per exemple, considereu el graf següent:

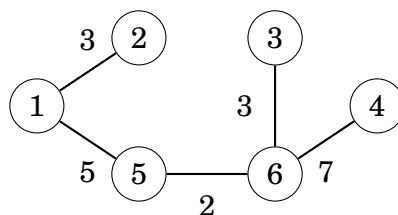


Un possible arbre d'expansió és el següent:

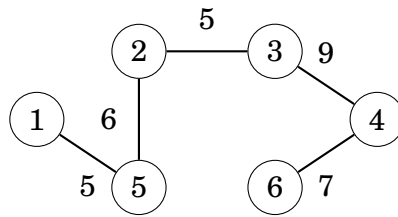


El pes d'un arbre d'expansió és la suma dels pesos de les seves arestes. Per exemple, el pes de l'arbre d'expansió anterior és $3 + 5 + 9 + 3 + 2 = 22$.

Un **arbre d'expansió mínim** és aquell que té pes mínim. En l'exemple anterior, el pes mínim és 20, i aquest arbre es pot construir com segueix:



De la mateixa manera, un **arbre d'expansió màxim** és aquell que té pes màxim. En l'exemple anterior és 32:



Tingueu en compte que un graf pot tenir diversos arbres d'expansió mínims i màxims, de manera que els arbres no són únics.

Resulta que es poden fer servir diversos algorismes greedy per construir arbres d'expansió mínim i màxim. En aquest capítol, discutim dos algorismes que processen les arestes del graf ordenades segons els seus pesos. Ens centrem en trobar arbres d'expansió mínim, però els mateixos algorismes poden trobar arbres d'expansió màxim processant les arestes en ordre invers.

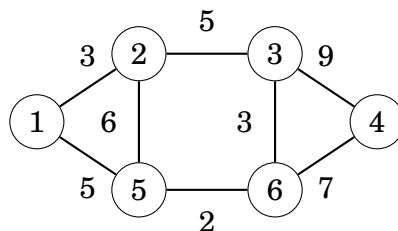
15.1 Algorisme de Kruskal

En l'**algorisme de Kruskal**¹, omplirem un arbre d'expansió aresta a aresta. L'algorisme recorre totes les arestes ordenades pels seus pesos, i afegeix l'aresta si no crea cap cicle amb les arestes ja seleccionades.

L'algorisme manté les components connexes de l'arbre d'expansió que estem omplint. Inicialment, cadascun dels n nodes del graf pertany a una component connexa distinta. Només afegim arestes quan unim dues components distintes, ja que altrament tindríem un cicle. L'algorisme acaba quan, després d'afegir $n - 1$ arestes, tenim una sola component connexa resultant. Quan això passa hem trobat un arbre d'expansió mínim.

Exemple

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



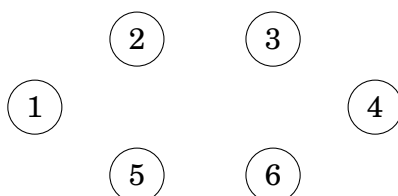
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

¹L'algorisme va ser publicat l'any 1956 per J. B. Kruskal [48].

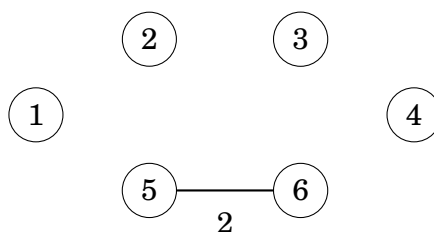
edge	weight
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Després d'això, l'algorisme recorre la llista i afegeix cada aresta a l'arbre si aquesta uneix dues components separades.

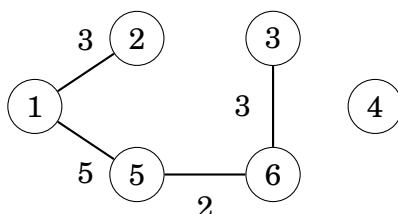
Inicialment, cada node té la seva pròpia component connexa:



La primera aresta que s'afegeix a l'arbre és la aresta 5-6 que crea la component {5,6} resultant d'unir les components {5} i {6}:



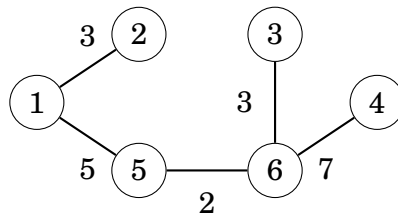
Després d'això, les arestes 1-2, 3-6 i 1-5 s'afegeixen de la mateixa manera:



Després d'aquests passos, la majoria de components s'han unit i hi ha dues components a l'arbre: {1,2,3,5,6} i {4}.

L'aresta següent de la llista és l'aresta 2-3, però no s'inclou a l'arbre perquè els nodes 2 i 3 ja estan a la mateixa component. El mateix passa amb l'aresta 2-5.

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

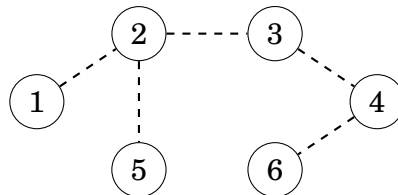


Després d'això, l'algorisme no afegeix cap aresta nova, perquè el graf està connectat i hi ha un camí entre dos nodes qualsevol. El graf resultant és un arbre d'expansió mínim amb pes $2 + 3 + 3 + 5 + 7 = 20$.

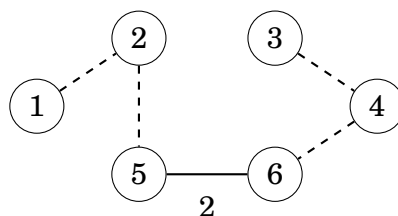
Per què funciona això?

Es bó preguntar-se per què funciona l'algorisme de Kruskal. Com és que aquesta estratègia greedy sempre funciona?

Vegem què passa si l'aresta de mínim pes del graf *no* s'inclogués a l'arbre d'expansió. Per exemple, suposem que l'arbre d'expansió del graf anterior no inclou l'aresta de pes mínim 5–6. No coneixem l'estructura exacta de l'arbre d'expansió, però sens dubte contindrà algunes arestes. Imaginem-nos un arbre com aquest:



Però no és possible que l'arbre anterior sigui un arbre d'expansió mínim. La raó d'això és que si afegim a aquest arbre l'aresta de pes mínim 5–6 estem creant un cicle, i per tant podríem qualsevol altra aresta del cicle i trobaríem un nou arbre d'expansió amb pes *menor*:



Per aquesta raó, sabem que sempre és òptim incloure qualsevol aresta de pes mínim a l'arbre d'expansió mínim. Amb un argument similar, podem demostrar que afegir l'aresta següent en ordre de pes també és òptim, i així successivament. Per tant, l'algorisme de Kruskal funciona correctament i sempre troba un arbre d'expansió mínim.

Implementació

Quan s'implementa l'algorisme de Kruskal, és convenient fer servir la representació de llista d'arestes del graf. La primera fase de l'algorisme ordena les arestes de la llista en temps $O(m \log m)$ temps. La segona fase de l'algorisme construeix l'arbre d'abast mínim de la següent manera:

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

El bucle passa per les arestes de la llista i processa arestes de la forma $a-b$. Es necessiten dues funcions: la funció `same` determina si a i b estan en la mateixa component connexa, i la funció `unite` uneix les components connexes que contenen a i b .

El problema és com implementar de manera eficient les funcions `same` i `unite`. Una possibilitat és implementar la funció `same` com a recorregut de grafs i comprovar si podem passar del node a al node b . Tanmateix, la complexitat temporal d'aquesta funció seria $O(n+m)$ i l'algorisme resultant seria lent, perquè es cridaria la funció `same` per a cada aresta del graf.

Resoldrem el problema fent servir una estructura *union-find* (o també coneguda com *merge-find-set* o *MF set*) que implementa ambdues funcions en temps $O(\log n)$. Així, la complexitat temporal de l'algorisme de Kruskal és $O(m \log n)$ després d'ordenar la llista de arestes.

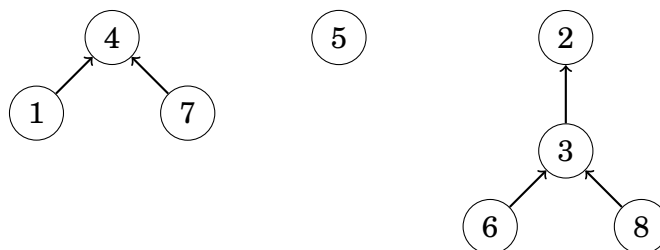
15.2 Estructura *union-find*

Una **estructura *union-find*** manté una col·lecció de conjunts. Els conjunts són disjunts, de manera que cap element pertany a més d'un conjunt. Aquesta estructura admet dues operacions de temps $O(\log n)$: l'operació `unite` uneix dos conjunts, i l'operació `find` troba el representant del conjunt que conté un element donat².

Estructura

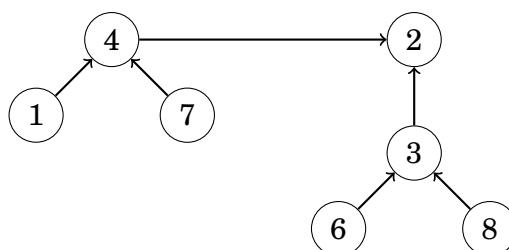
En una estructura *union-find*, un element de cada conjunt és el representant del conjunt, i hi ha una cadena des de qualsevol altre element del conjunt fins al seu representant. Per exemple, suposem que els conjunts són $\{1, 4, 7\}$, $\{5\}$ i $\{2, 3, 6, 8\}$:

²L'estructura presentada aquí va ser introduït el 1971 per JD Hopcroft i JD Ullman [38]. Més tard, el 1975, R. E. Tarjan va estudiar una variant més sofisticada de l'estructura [64] que es discuteix en molts llibres d'algorísmica.



En aquest cas els representants dels conjunts són 4, 5 i 2. Podem trobar el representant d'un element seguint la cadena que comença en l'element donat. Per exemple, l'element 2 és el representant del conjunt que conté l'element 6, perquè seguim la cadena $6 \rightarrow 3 \rightarrow 2$. Dos elements pertanyen al mateix conjunt exactament quan els seus representants són els mateixos.

Podem unir dos conjunts connectant el representant d'un conjunt amb el representant de l'altre conjunt. Per exemple, els conjunts $\{1, 4, 7\}$ i $\{2, 3, 6, 8\}$ es poden unir de la següent manera:



El conjunt resultant conté els elements $\{1, 2, 3, 4, 6, 7, 8\}$. A partir d'aquí, l'element 2 és el representant de tot el conjunt i l'antic representant 4 apunta a l'element 2.

L'eficiència de l'estructura d'unió-troba depèn de com s'uneixen els conjunts. Resulta que podem seguir una estratègia senzilla: fent la connexió des del representant del conjunt *més petit* al representant del conjunt *més gran* (o, si els conjunts són de la mateixa mida, podem fer una elecció arbitrària). Amb aquesta estratègia, es pot veure que la longitud de qualsevol cadena és $O(\log n)$, de manera que es pot trobar el representant de qualsevol element de manera eficient seguint la cadena corresponent.

Implementació

L'estructura *union-find* s'implementa amb vectors. En la següent implementació, el vector `link` conté per a cada element l'element següent de la cadena, o el propi element si és un representant del conjunt, i el vector `size` indica per a cada representant la mida del seu conjunt corresponent.

Inicialment, cada element pertany a un conjunt distint:

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

La funció `find` retorna el representant d'un element x , que es troba seguint la cadena que comença a x .

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

La funció `same` verifica si els elements a i b pertanyen al mateix conjunt. Això es pot fer fàcilment utilitzant la funció `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;
}
```

La complexitat temporal de la funció `find` és $O(\log n)$ perquè la longitud de cada cadena és $O(\log n)$. En aquest cas, les funcions `same` i `unite` també funcionen en temps $O(\log n)$. La funció `unite` assegura que la longitud de cada cadena és $O(\log n)$ perquè connecta el conjunt més petit al conjunt més gran.

15.3 Algorisme de Prim

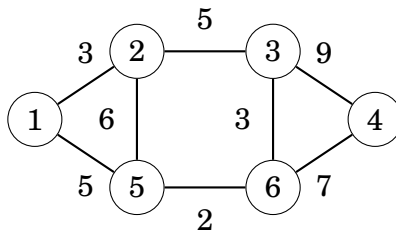
L'**algorisme de Prim**³ és un mètode alternatiu per trobar un arbre d'expansió mínim. L'algorisme afegeix primer un node arbitrari a l'arbre. Després d'això, l'algorisme sempre tria una aresta de pes mínim que afegeix un nou node a l'arbre. Al final, s'han afegit tots els nodes a l'arbre i s'ha trobat un arbre d'expansió mínim.

L'algorisme de Prim s'assembla a l'algorisme de Dijkstra. La diferència és que l'algorisme de Dijkstra sempre selecciona una aresta la distància de la qual des del node inicial és mínima, però l'algorisme de Prim simplement selecciona la aresta de pes mínim que afegeix un nou node a l'arbre.

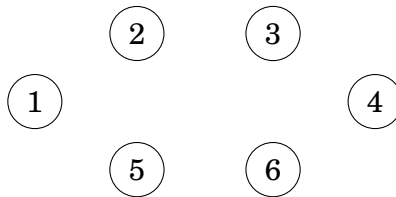
Exemple

Considerem com funciona l'algorisme de Prim en el graf següent:

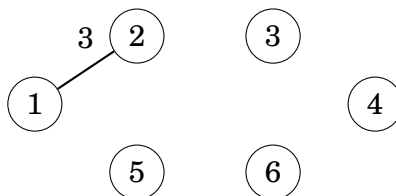
³L'algorisme rep el nom de R. C. Prim que el va publicar l'any 1957 [54]. No obstant això, el mateix algorisme ja va ser descobert l'any 1930 per V. Jarník.



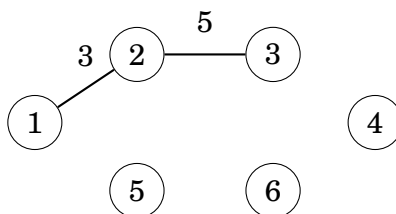
Inicialment, no hi ha arestes entre els nodes:



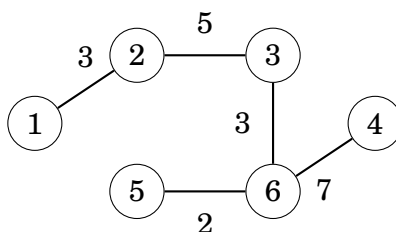
El node inicial és un node arbitrari, així que escollim el node 1. Primer, afegim el node 2 que està connectat per una aresta de pes 3:



Després d'això, hi ha dos arestes amb pes 5, de manera que podem afegir el node 3 o el node 5 a l'arbre. Afegim primer el node 3:



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



Implementació

A l'igual que l'algorisme de Dijkstra, l'algorisme de Prim s'implementa de manera eficient amb una cua de prioritats. La cua de prioritats ha de contenir tots els

nodes que es poden connectar al component actual mitjançant una única aresta, en ordre creixent dels pesos de les arestes corresponents.

La complexitat temporal de l'algorisme de Prim és $O(n + m \log m)$ que és igual a la complexitat temporal de l'algorisme de Dijkstra. A la pràctica, els algorismes de Prim i Kruskal són tots dos eficients, i l'elecció de l'algorisme és qüestió de gustos. La majoria dels programadors competitius fan servir l'algorisme de Kruskal.

Part III

Temes avançats

Bibliografia

- [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.

