

Anàlisi i Disseny d'Algorismes (ADA)

Col·lecció de Problemes

9 de febrer de 2009




The abacist versus the algorist.
(From Gregor Reisch, *Margarita Philosophica*, Strassbourg, 1504.)


María Teresa Abad
Albert Atserias
Maria Josep Blesa
Rafel Cases
Amalia Duch
Carles Franquesa
Guillem Godoy

Conrado Martínez
Edelmira Pasarella
Jordi Petit
Salvador Roura
Maria José Serna
Gabriel Valiente
Àlvar Vinacua
Cristina Zoltán

Índex

1	Anàlisi d'algorismes	1
2	Estructures de dades	13
2.1	Taules de dispersió	13
2.2	Arbres binaris de cerca	14
2.3	Arbres AVL	17
2.4	Altres diccionaris	18
2.5	Heaps	19
2.6	Particions	26
3	Grafs	29
4	Algorismes de dividir i vèncer	39
5	Algorismes de programació dinàmica	49
6	Algorismes voraços	57
7	Algorismes de cerca exhaustiva	65
8	Introducció a la NP-completesa	75
9	Solucions	83


Nota: Els problemes marcats amb ☆ són més difícils (més símbols indiquen més dificultat). Els problemes marcats amb un  apareixen resolts al final de la col·lecció i la solució possiblement es presenti a classe, però, intenteu resoldre'ls vosaltres mateixos abans de mirar les solucions!

El document en PDF té diversos elements de navegació: es pot clicar en la taula de continguts per a accedir directament al corresponent capítol, es pot clicar sobre el símbol  per a accedir directament a la corresponent solució, etc.


Si no es diu el contrari i us cal, considereu que tots els logaritmes són en base 2.

1

Anàlisi d'algorismes

1. Utilitzeu paper milimètric, un full de càlcul o un programa de dibuix per traçar les funcions $\log x$, x , $x \log x$, x^2 , x^3 i 2^x per a $x \in \{0, \dots, 100\}$ i per a $x \in \{0, \dots, 10000\}$.
2. Contesteu les preguntes següents sobre logaritmes:
 - Quants bits calen per representar un nombre natural entre 0 i $n - 1$?
 - Començant per $x = 1$, quants cops cal doblar x fins que sigui més gran o igual que n ?
 - Començant per $x = n$, quants cops cal dividir x per dos fins que sigui menor o igual que 1? I si es divideix per tres?
3. Demostreu les igualtats següents per inducció:
 - $\sum_{i=1}^n i = n(n+1)/2$.
 - $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$.
4. Demostreu que $\sum_{i=0}^n 2^i = 2^{n+1} - 1$.
-  5. Demostreu que $\sum_{i=1}^k i2^{-i} = 2 - k2^{-k} - 2^{1-k}$.
6. Demostreu les propietats asimptòtiques següents:
 - Tot polinomi $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 n^0$ amb $a_k > 0$ pertany a $\Theta(n^k)$.
 - Per a qualssevol bases $a, b > 1$, es té que $\log_a n = \Theta(\log_b n)$.

☆☆ 7. Demostreu que $\log(n!) = \Theta(n \log n)$.

☆  8. El número harmònic n -èsim és $H_n = \sum_{i=1}^n 1/i$. Mostreu que $H_n = \Theta(\log n)$ aproximant per integrals contínues.


9. Demostreu que $\sum_{i=1}^n H_i = \Theta(n \log n)$.

10. Gordon Moore, cofundador d'Intel, va fer cap al 1965 una cèlebre observació que avui en dia es coneix com a “Llei de Moore”. Moore predeia un creixement exponencial en el nombre de transistors en els circuits integrats i en la seva velocitat. Essencialment, Moore va predir que la potència dels processadors doblaria cada 18 mesos.


A la novel·la “Turing (A novel about computation)” de Christos Papadimitriou (MIT Press 2003) l'única indicació de l'any en què succeix, és el fet que Ethel, la protagonista, té un ordinador amb un processador de 9 GHz. Podrieu situar aproximadament en quin any té lloc la novel·la?

11. Existeixen diferents versions sobre l'origen del joc d'escacs. Totes elles el situen a l'Índia i relaten, de forma més o menys semblant, la demanda feta per un savi a un rei per tal de completar amb grans de blat un tauler d'escacs amb la condició que cada casella continguéss el doble de grans que l'anterior. A la primera casella va demanar que hi col·loquessin un gra, a la segona dos grans, a la tercera quatre grans i així fins a la darrera. Calculeu el nombre total de grans de blat requerit.



Els agricultors compten el gra en *bushels* (1 bushel equival a uns 35.238 litres). Sabent que 1 bushel conté aproximadament uns 5 milions de grans de blat i que (en l'actualitat) la producció mundial mitjana de blat se situa al voltant dels 2.000.000.000 bushels, quantes collites caldrien per satisfer la demanda del savi?

 12. Per a cada funció $f(n)$ i temps t de la taula següent, determineu la talla més gran que es pot resoldre en temps t d'un problema que necessita un temps de $f(n)$ μ s per executar-se sobre una entrada de talla n .

	1 segon	1 minut	1 hora	1 dia	1 mes	1 any	1 segle
$\log n$							
\sqrt{n}							
n							
n^2							
n^3							
2^n							

 13. La tecnologia actual permet resoldre, en un cert temps T , un problema de talla fins a $n = 10^8$ amb un algorisme de complexitat $\Theta(n)$. Supposeu que la constant amagada en la notació asimptòtica és la mateixa per a tots els algorismes. Quina és la talla n de la instància més gran que es pot resoldre en temps T amb un algorisme de complexitat $\Theta(n\sqrt{n})$ i per què?

Suposeu que amb la tecnologia d'aquí a deu anys, els processadors siguin 100 cops més ràpids. Quina serà llavors la talla n de la instància més gran que es podrà resoldre en temps T amb un algorisme de complexitat $\Theta(n\sqrt{n})$?


-  14. Considereu les funcions següents: $5n \ln n$, $4n\sqrt{n}$, $\log_\pi(n^n)$, $n/\log_2 n$, $n/\ln n$, $3 \ln(7^n)$. Ordeneu-les, de menor a major, segons el seu creixement asimptòtic. Si dues o més funcions tenen el mateix grau de creixement, indiqueu-ho.
15. Agrupeu les funcions següents de manera que $f(n)$ i $g(n)$ pertanyin al mateix grup si i només si $f(n) \in \Theta(g(n))$, i enumereu els grups segons el seu ordre de creixement: n , \sqrt{n} , $n^{1.5}$, n^2 , $n \log n$, $n \log \log n$, $(\log \log n)^2$, $n \log^2 n$, $n^3/(n-1)$, $n^{\log n}$, $n \log(n^2)$, $21n$, 2^n , $2^{\sqrt{n}}$, $2^{\sqrt{\log n}}$, $2^{n/2}$, 37 , 2^{2^n} , $2/n$ i $n^2 \log n$.
-  16. De cadascun dels fragments de codi següents, analitzeu el seu cost.

```
// Fragment 1
int s = 0;
for (int i = 0; i < n; ++i) {
    ++s;
}

// Fragment 2
int s = 0;
for (int i = 0; i < n; i += 2) {
    ++s;
}

// Fragment 3
int s = 0;
for (int i = 0; i < n; ++i) {
    ++s;
}
for (int j = 0; j < n; ++j) {
    ++s;
}


// Fragment 4
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        ++s;
    }
}

// Fragment 5 
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        ++s;
    }
}
```

```
// Fragment 6
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = i; j < n; ++j) {
        ++s;
    }
}

// Fragment 7
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < n; ++k) {
            ++s;
        }
    }
}

// Fragment 8
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        for (int k = 0; k < j; ++k) {
            ++s;
        }
    }
}

// Fragment 9 
int s = 0;
for (int i = 1; i <= n; i *= 2) {
    ++s;
}

// Fragment 10
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i * i; ++j) {
        for (int k = 0; k < n; ++k) {
            ++s;
        }
    }
}

// Fragment 11 ☆
int s = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i * i; ++j) {
        if (j % i == 0) {
            for (int k = 0; k < n; ++k) {
                ++s;
            }
        }
    }
}
```

17. Digueu quin és el cost de les instruccions (1), (2) i (3) del programa següent.


```
int funcio1 (vector<int>& v) { return v[0]; }
int funcio2 (vector<int> v) { return v[0]; }

int f (int n) {
    vector<int> v(n,0);           // (1)
    int a = funcio1(v);           // (2)
    int b = funcio2(v);           // (3)
    return a+b;
}
```

18. Escriviu un algorisme de cerca lineal per determinar si un element apareix en una taula de n posicions. Analitzeu el seu cost.
19. Utilitzeu la notació asimptòtica més indicada (\mathcal{O} , Ω , Θ) per indicar el cost en temps de la cerca seqüencial.

- En el cas pitjor,
- En el cas millor,
- En el cas mitjà.

20. Considereu una taula T amb n elements.

- Escriviu un algorisme que usi exactament $n - 1$ comparacions per trobar l'element mínim de T .
- Escriviu un algorisme que usi exactament $n - 1$ comparacions per trobar l'element màxim de T .
- ☆ Escriviu un algorisme que trobi els elements mínim i màxim de T amb només unes $\frac{3}{2}n$ comparacions.

21. Quin és el cost en temps i en espai dels algorismes “escolars” per

- Sumar dos naturals de n dígit,
- Multiplicar dos naturals de n dígit.



22. Escriviu l'algorisme clàssic per sumar dues matrius $n \times n$. Expliqueu perquè aquest algorisme de cost $\mathcal{O}(n^2)$ és lineal.

Demostreu que qualsevol algorisme per sumar dues matrius $n \times n$ requereix $\Omega(n^2)$ passos.

23. Escriviu l'algorisme clàssic per multiplicar dues matrius $n \times n$. Quin és el seu cost en funció de n ? Quin és el seu cost en funció de la talla de les entrades?

Demostreu que qualsevol algorisme per multiplicar dues matrius $n \times n$ requereix $\Omega(n^2)$ passos.



eratòstenes

24. Escriviu un algorisme per determinar si un número natural n és primer. Quin és el seu cost en funció de n ?
25. Escriviu i analitzeu l'algorisme corresponent al garbell d'Eratòstenes per determinar tots els nombres primers entre 2 i n .
26. Escriviu un algorisme recursiu per calcular el factorial d'un nombre natural n . Quin és el seu cost en funció de n ? Passeu l'algorisme a iteratiu.



27. Resoleu les recurrències substractores següents:

- $T(n) = T(n - 1) + \Theta(1)$,
- $T(n) = T(n - 2) + \Theta(1)$,
- $T(n) = T(n - 1) + \Theta(n)$,
- $T(n) = 2T(n - 1) + \Theta(1)$.



28. Resoleu les recurrències divisores següents:

- $T(n) = 2T(n/2) + \Theta(1)$,
- $T(n) = 2T(n/2) + \Theta(n)$,
- $T(n) = 2T(n/2) + \Theta(n^2)$,
- $T(n) = 2T(n/2) + \mathcal{O}(1)$,
- $T(n) = 2T(n/2) + \mathcal{O}(n)$,
- $T(n) = 2T(n/2) + \mathcal{O}(n^2)$,
- $T(n) = 4T(n/2) + n$,
- $T(n) = 4T(n/2) + n^2$,
- $T(n) = 4T(n/2) + n^3$,
- $T(n) = 9T(n/3) + 3n + 2$,
- $T(n) = T(9n/10) + \Theta(n)$,
- $T(n) = 2T(n/4) + \sqrt{n}$,

- ☆☆ 29. Resoleu la recurrència $T(n) = T(\sqrt{n}) + 1$ tot utilitzant un canvi de variables.

30. Ordeneu la taula $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ utilitzant els algorismes d'ordenació elementals (selecció, inserció i bombolla).
31. Digueu quins algorismes d'ordenació elementals són estables. Pels que ho són, justifiqueu per què ho són, pel que no ho són, doneu un contraexemple.
32. Digueu quan triguen els algorismes d'ordenació elementals quan l'entrada es troba...
 - Permutada a l'atzar,
 - Ordenada,

- Ordenada del revés,
- Amb tots els elements iguals.

33. Cap a l'any 1202 Fibonacci de Pisa investigava com de ràpid es podien reproduir els conills en condicions ideals: Supposeu que un camp hi ha una parella de conills acabats de néixer, l'un mascle, l'altre femella. Els conills es poden aparellar a l'edat d'un mes, de forma que al final del segon més la femella dóna llum a una nova parella de conills. Supposeu que els conills no moren mai i que les femelles produeixen una nova parella (un mascle i una femella) cada parell de mesos. Quants conills hi haurà després d'un any?



Fibonacci

Els nombres de Fibonacci venen definits per la recurrència següent:

$$F_n = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2. \end{cases}$$

Escriviu un algorisme recursiu per calcular el nombre de Fibonacci n -èsim. Implementeu-lo i executeu-lo per mesurar quan triga per calcular F_{24} .

Quin és el cost asimptòtic de l'algorisme? Per què és lent?

Escriviu un algorisme iteratiu que tingui cost lineal en funció de n .

34. Tenim una taula t amb n elements i desitgem saber si k_n altres elements són dins de la taula t o no. Hi ha, com a mínim, dues maneres possibles de procedir:

Opció 1: Per a cadascun dels k_n elements, fem una cerca lineal a la taula t .

Opció 2: Ordenem primer la taula t amb un algorisme $\Theta(n \log n)$ i, després, per a cadascun dels k_n elements, fem una cerca dicotòmica a la taula t .

Digueu quin ha de ser el valor mínim de k_n (utilitzant notació asimptòtica) perquè la segona opció sigui més ràpida o igual que la primera.

35. Analitzeu el cost de les funcions recursives següents per calcular x^n per a $n \geq 0$.

```
double potencia_1 (double x, int n) {
    if (n==0) {
        return 1;
    } else {
        return x*potencia_1(x,n-1);
    }
}
```

```
double potencia_2 (double x, int n) {
    if (n==0) {
        return 1;
    } else if (n%2==0) {
        int y = potencia_2(x,n/2);

```

```

        return y*y;
    } else {
        int y = potencia_2(x,n/2);
        return y*y*x;
    } }

double potencia_3 (double x, int n) {
    if (n==0) {
        return 1;
    } else if (n%2==0) {
        return potencia_3(x,n/2) * potencia_3(x,n/2);
    } else {
        return potencia_3(x,n/2) * potencia_3(x,n/2) * x;
    } }

```

36. Considereu les dues funcions següents per trobar el màxim d'una taula de n reals (amb $n \geq 1$):


```


// Crida inicial: maxS(t,n-1)
double maxS(const vector<double>& t, int i) {
    if (i==0) {
        return t[0];
    } else {
        double m = maxS(t,i-1);
        return t[i]<m ? m : t[i];
    } }

// Crida inicial: maxD(t,0,n-1)
double maxD(const vector<double>& t, int i, int j) {
    if (i==j) {
        return t[i];
    } else {
        int k = (i+j)/2;
        double m1 = maxD(t,i,k);
        double m2 = maxD(t,k+1,j);
        return m1>m2 ? m1 : m2;
    } }

```

Calculeu el cost dels dos algorismes en funció de la talla de la taula n . Quin d'ells escolliríeu? Per què?

-  37. Un algorisme A té un cost donat per la recurrència $T_A(n) = 7T_A(n/2) + n^2$. Un algorisme competidor B té cost $T_B(n) = xT_B(n/4) + n^2$. Quin és l'enter x més gran per al qual B és asimptòticament millor que A ?

-  38. Calculeu el cost de l'algorisme següent:

```


double F (vector<double>& v, int i, int j) {
    int aux = (j-i+1)/4;

```

```

    if (aux>0) {
        int m1 = i-1+aux;
        int m2 = m1+aux;
        int m3 = m2+aux;
        return F(v,i,m1) + F(v,m1+1,m2) +
               F(v,m2+1,m3) + F(v,m3+1,j);
    } else {
        return i+j-1;
    }
}

```

 39. Considereu les dues funcions següents:

```

double A (vector<double>& v, int i, int j) {
    if (i<j) {
        int x = f(v,i,j);
        int m = (i+j)/2;
        return A(v,i,m-1) + A(v,m,j) + A(v,i+1,m) + x;
    } else {
        return v[i];
    }
}

double B (vector<double>& v, int i, int j) {
    if (i<j) {
        int x = g(v,i,j);
        int m1 = i+(j-i+1)/3;
        int m2 = i+(j-i+1)*2/3;
        return B(v,i,m1-1) + B(v,m1,m2-1) + B(v,m2,j) + x;
    } else {
        return v[i];
    }
}

```

Suposant que el cost de la funció f és $\Theta(1)$ i que el cost de g és proporcional a la grandària del vector a processar (és a dir, $\Theta(j-i+1)$), quin és el cost asimptòtic de A i de B en funció de n ?

Quina de les dues funcions escollirieu si fan el mateix?

40. Considereu una taula bidimensional de talla $n \times n$ on cada columna té els elements ordenats en ordre estrictament creixent de dalt a baix i cada fila té els elements ordenats en ordre estrictament creixent d'esquerra a dreta. Doneu un algorisme $\mathcal{O}(n)$ que decideixi si un número x donat és o no a la matriu.
41. Considereu el fragment de codi C++ següent:

```

double* T = new double[1];
int mida = 1, n = 0;
double x;
cin >> x;
while (x!=0) {
    if (n==mida) {

```

```

        double* Taux = new double[2*mida];
        for (int i=0; i<mida; ++i) {
            Taux[i] = T[i];
        }
        delete[] T;
        T = Taux;
        mida = 2*mida;
    }
    T[n++] = x;
    cin >> x;
}

```

Expliqueu breument què fa aquest codi.

Suposant que les instruccions `new[]` i `delete[]` tenen cost constant, digueu raonadament quin és el cost d'aquest codi.

42. Considereu el codi següent:

```

const int M = 1000;
struct cadena {
    int n;           // 0 ≤ n ≤ M
    char T[M];       // la cadena es guarda a
                    // les n primeres posicions de T
};

int comptar_as (cadena& c) {
    int as = 0;
    for (int i=0; i<c.n; ++i) {
        if (c.T[i]=='A') {
            ++as;
        }
    }
    return as;
}

```

Detecteu on és l'error en el raonament següent: “Com que la taula T té una capacitat màxima M , el cost en cas pitjor de `comptar_as` és $\mathcal{O}(M)$, ja que el programa recorre $c.n$ posicions de la taula i $c.n \leq M$. Com que M és una constant, el cost en cas pitjor és senzillament $\mathcal{O}(1)$.”

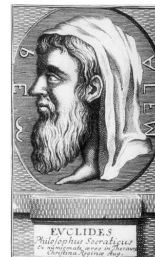
☆☆ 43. Considereu l'algorisme d'Euclides per calcular el màxim comú divisor de dos naturals:

```

// Precondicio: 0 < j ≤ i
int mcd (int i, int j) {
    int r = i % j;
    if (r==0) {
        return j;
    } else {
        return mcd(j,r);
    }
}

```

Demostreu que el cost de la funció `mcd(i, j)` és $\mathcal{O}(\log i)$. *Ajut:* Primer, desplegueu un cop el codi recursiu, és a dir, substituïu la invocació de la crida recursiva `return mcd(j, r)` pel text de `mcd` amb els paràmetres adients. A continuació, mostreu que en el codi reescrit, en fer la crida recursiva, de la forma `mcd(m, n)`, es compleix $m \leq i/2$. Finalment, expresseu el cost de l'algorisme amb una recurrència i resoleu-la.




Euclides


Nota històrica: Euclides va donar el seu algorisme utilitzant restes i no pas mòduls.

44. Malgrat anar tot el dia adormit, l'Omer ha escrit un algorisme *correcte* que retorna la mediana d'una taula de n elements. Vosaltres no heu vist l'algorisme de l'Omer (de fet, probablement ni tans sols coneixeu l'Omer encara que el seu nom us soni per ser un estudiant de la FIB que va quedar en 15è lloc al concurs mundial de programació de l'ACM del 2004). Tanmateix, podeu assegurar que només una de les afirmacions següents és certa. Digueu raonadament quina.

- (a) El cost de l'algorisme de l'Omer és per força $\Theta(n)$.
- (b) El cost de l'algorisme de l'Omer és per força $\Omega(n)$.
- (c) El cost de l'algorisme de l'Omer en el cas mitjà és per força $\mathcal{O}(\log n)$.
- (d) El cost de l'algorisme de l'Omer en el cas pitjor és per força $\Theta(n \log n)$.

- ☆☆ 45. Considereu una matriu M de talla $n \times n$ en què cada columna està ordenada de manera estrictament creixent de dalt a baix, i cada fila està ordenada de manera estrictament creixent d'esquerra a dreta. Escriviu en C++ o Java una funció que, donat un element x , determini en temps $\Theta(n)$ quants elements de la matriu M són estrictament menors que x .

-  46. Tenim un algorisme A amb cost en cas pitjor $\Theta(\sqrt{n})$ i un altre B amb cost en cas pitjor $\Theta(2^{\sqrt{\log_2 n}})$. Quin del dos algorismes és més eficient? O tenen la mateixa eficiència asimptòtica? Justifica la teva resposta.

-  47. El cost $T(n)$ d'un algorisme recursiu satisfà $T(n) = xT(n/4) + \Theta(\sqrt{n})$, amb $x > 0$. Quin és el cost $T(n)$?

Pista: Hi ha tres situacions possibles, segons l'interval en el qual cau la x .

Estructures de dades

2.1 Taules de dispersió



1. Considereu una taula de dispersió d'encadenament separat amb $M = 10$ posicions per a claus enteres i funció de dispersió $h(x) = x \bmod M$.

Començant amb una taula buida, mostreu com queda després d'inserir les claus 3441, 3412, 498, 1983, 4893, 3874, 3722, 3313, 4830, 2001, 3202, 365, 128181, 7812, 1299, 999 i 18267.



Mostreu com queda la taula de dispersió anterior quan s'aplica una redispersió per enmagatzemar 20 posicions.

2. Construïu la taula de dispersió d'encadenament separat per a les claus 30, 20, 56, 75, 31 i 19, amb 11 posicions i funció de dispersió $h(x) = x \bmod 11$.

Calculeu el nombre esperat de comparacions en una cerca amb èxit en aquesta taula.

Calculeu el nombre màxim de comparacions en una cerca amb èxit en aquesta taula.



3. Considereu una taula de dispersió oberta amb 20 posicions. Suposant que la taula és buida, mostreu com queda després d'inserir les claus 3441, 3412, 498, 1983, 4893, 3874, 3722, 3313, 4830, 2001, 3202, 365, 128181, 7812, 1299, 999 i 18267 utilitzant exploració lineal i prenent com a funció de dispersió $h(x) = x \bmod 10$.

4. Construïu la taula de dispersió oberta amb 11 posicions per a les claus 30, 20, 56, 75, 31 i 19 i funció de dispersió $h(x) = x \bmod 11$.

Calculeu el nombre esperat de comparacions en una cerca amb èxit en aquesta taula.

Calculeu el nombre màxim de comparacions en una cerca amb èxit en aquesta taula.

5. Dissenyeu i analitzeu un algorisme que utilitzi taules de dispersió per comprovar que tots els elements d'una llista són diferents.
6. Un programa en BASIC consisteix d'una sèrie d'instruccions numerades en ordre creixent. El control de flux es gestiona a través de les instruccions `GOTO x` i `GOSUB x`, on `x` és un número d'instrucció. Per exemple, el programa següent calcula el factorial d'un número:

```
50 INPUT N
60 LET F = 1
61 LET I = 1
73 IF I=N THEN GOTO 99
76 LET F = F*I
80 LET I = I+1
81 GOTO 73
99 PRINT F
```

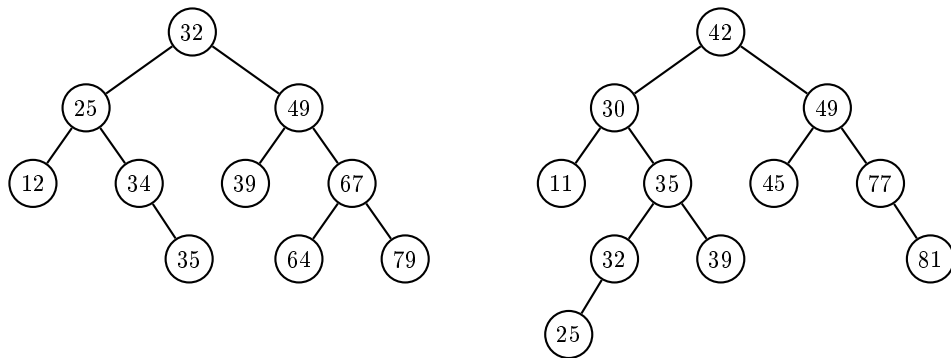
La instrucció `RENUM` renumera les instruccions del programa de forma que les línies vagin de 10 en 10. Per exemple, després de renumerar el programa anterior queda:

```
10 INPUT N
20 LET F = 1
30 LET I = 1
40 IF I=N THEN GOTO 80
50 LET F = F*I
60 LET I = I+1
70 GOTO 40
80 PRINT F
```

Descriviu com implementar la instrucció `RENUM` en temps lineal en el cas mitjà.

2.2 Arbres binaris de cerca

7. Digueu si els arbres següents són arbres binaris de cerca o no i perquè.

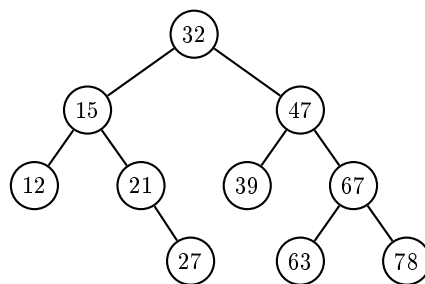


8. Partint d'un arbre binari de cerca buit, inseriu amb l'algorisme clàssic, l'una rera l'altra, la seqüència de claus 32, 15, 47, 67, 78, 39, 63, 21, 12, 27.

9. Partint d'un arbre binari de cerca buit, inseriu amb l'algorisme clàssic, l'una rera l'altra, la seqüència de claus 12, 15, 21, 27, 32, 39, 47, 63, 67, 78.



10. Partint de l'arbre binari de cerca següent, elimineu les claus 63, 21, 15, 32 l'una rera l'altra. Expliqueu quin algorisme heu usat per eliminar les claus.



11. Expliqueu si és cert o no que en un arbre binari de cerca no buit, l'element màxim pot tenir fill esquerre però no pot tenir fill dret.

12. Demostreu que el recorregut en inordre d'un arbre binari de cerca visita els elements en ordre creixent.

- Per als problemes següents, utilitzeu aquesta definició de tipus per als arbres binaris de cerca:

```
struct node {
    Elem x;                // Informació en el node
    node* fe;              // Punter al fill esquerre
    node* fd;              // Punter al fill dret

    node(Elem _x, node* _fe, node* _fdr)
        : x(_x), fe(_fe), fd(_fd) {}
}
```

```

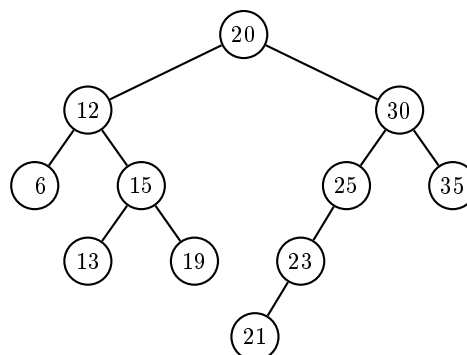
~node () {
    delete fe; delete fd;
}

};

typedef node* abc; // Un ABC es denota per un punter a la seva a.
                  // (null si és buit)

```

13. Implementeu una operació `abc crear ()` que retorni un arbre binari de cerca buit. Quin és el seu cost?
14. Implementeu una operació `bool hi_es (abc a, Elem x)` que indiqui si l'element `x` és a l'arbre binari de cerca `a`. Quin és el seu cost?
15. Implementeu una operació `void afegir (abc& a, Elem x)` que afegixi l'element `x` a l'arbre binari de cerca `a`. Quin és el seu cost?
16. Implementeu una operació `void treure (abc& a, Elem x)` que tregui l'element `x` de l'arbre binari de cerca `a`. Quin és el seu cost?
17. Implementeu una operació `Elem minim (abc a)` que retorni l'element més petit de l'arbre binari de cerca no buit `a`. Quin és el seu cost?
18. Implementeu una operació `Elem maxim (abc a)` que retorni l'element més gran de l'arbre binari de cerca no buit `a`. Quin és el seu cost?
19. La fusió de dos arbres binaris de cerca és un arbre binari de cerca que conté tots els elements de `a1` i `a2`. Implementeu una operació `abc fusio (abc& a1, abc& a2)` que retorni la fusió dels arbres binaris de cerca `a1` i `a2` (els dos arbres donats han de quedar buits després de la crida). Quin és el seu cost?
20. Dissenyeu una funció `list<Elem> entre (abc a, Elem x1, Elem x2)` que, donat un arbre binari de cerca `a` i dos elements `x1` i `x2` amb $x1 \leq x2$, retorni una llista amb tots els elements de `a` que es trobin entre `x1` i `x2` en ordre decreixent.
Per exemple, donat l'arbre

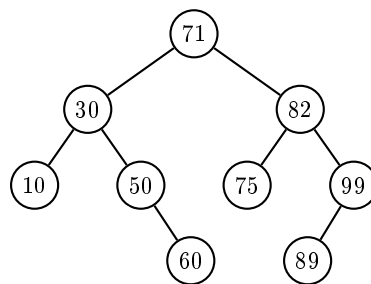


i els valors 18 i 26, cal retornar la llista $\langle 25, 23, 21, 20, 19 \rangle$.

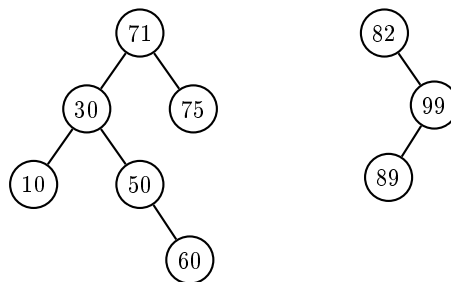
Calculeu el cost del vostre algorisme en el cas pitjor.



21. Implementeu una funció `int menors (abc a, Elem x)` que retorni el nombre d'elements de l'arbre binari de cerca `a` que són estrictament inferiors a `x`. Quin és el seu cost?
22. Dibuixeu l'arbre binari de cerca el recorregut en preordre del qual és 8, 5, 2, 1, 4, 3, 6, 7, 11, 9, 13, 12.
23. Implementeu la funció `abc reconstrueix (vector<Elem> pre)` que reconstrueix un arbre binari de cerca `a` a partir del seu recorregut en preordre emmagatzemat en un vector `pre`. Quin és el seu cost?
24. Implementeu una funció `void separa (abc& a, Elem x, abc& le, abc& gt)` que donat un arbre binari de cerca `a`, retorni dos arbres binaris de cerca `le` i `gt` on `le` conté tots els elements de `a` que són més petits o iguals que `x` i `gt` conté tots els elements de `a` que són més grans que `x`. L'arbre original `a` ha de quedar destruït. Per exemple, donat l'arbre binari de cerca següent



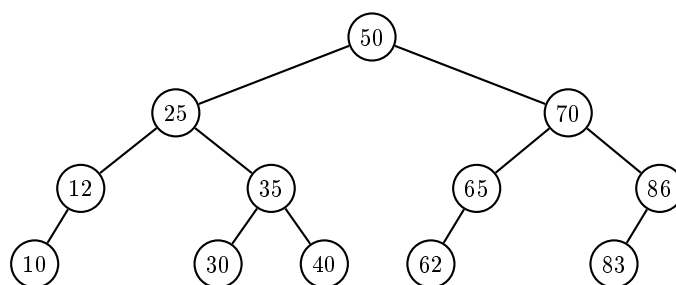
i l'element $x = 75$, els arbres `le` i `gt` són els següents:



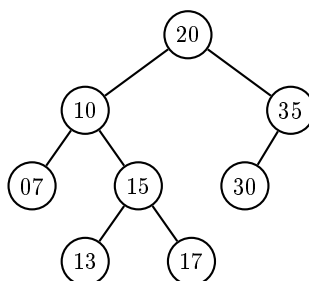
2.3 Arbres AVL



25. Doneu els tres arbres AVL resultants d'afegir les claus 31, 32 i 33 l'una després de l'altra en l'arbre AVL següent:



26. Doneu els quatre arbres AVL resultants d'afegir les claus 18 i 12 i d'esborrar les claus 7 i 30 a l'arbre AVL següent (apliqueu cada operació a l'arbre obtingut anteriorment):



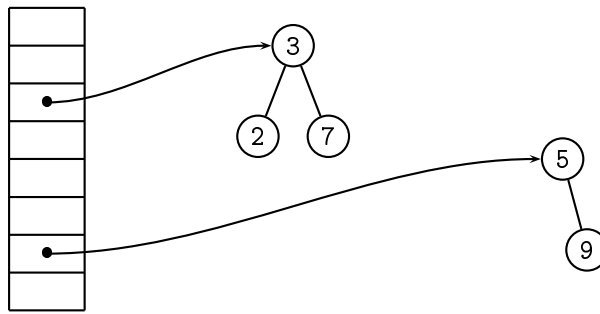
27. Dibuixeu un arbre AVL d'alçada màxima que tingui les claus 1, 2, ..., 12.

2.4 Altres diccionaris

28. Supposeu que es vol emmagatzemar un nombre d'elements proper a N , on N és un valor gran conegut a priori, de manera que es puguin fer tant eficientment com sigui possible les operacions següents: buscar un element, inserir un element, esborrar un element, saber el nombre total d'elements. Quina estructura de dades usàrieu?

Si les operacions demanades fossin: buscar un element, inserir un element, esborrar un element, obtenir el mínim element, i obtenir el màxim element, quina estructura de dades faríeu servir?

29. Supposeu que implementem una taula de dispersió amb encadenament separat, però enlloc de posar les claus que col·lisionen en una llista encadenada, les posem en un arbre AVL. Per exemple, si les claus 2, 3 i 7 i les claus 5 i 9 col·lisionessin, llavors es tindria la situació següent:

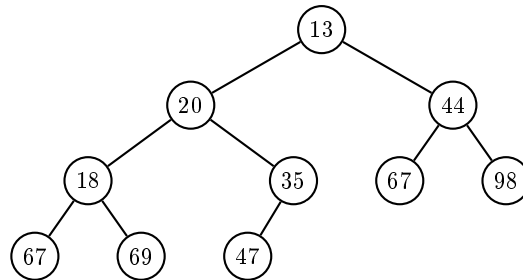
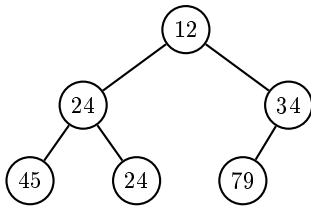


Suposeu que la taula té M posicions i n claus emmagatzemades, amb $M = \Theta(n)$. Considereu el cost de buscar una clau.

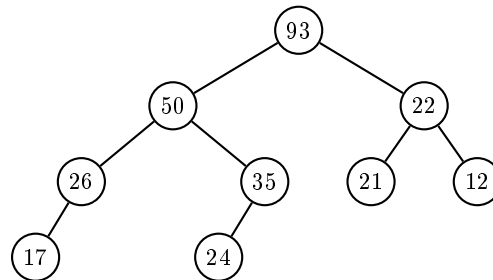
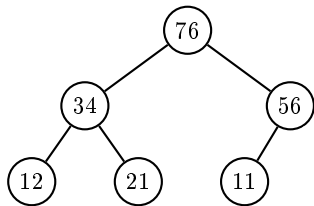
- Quin és el cas pitjor?
- Quant costa buscar una clau en el cas pitjor?
- Quant costa buscar una clau en el cas mitjà?

2.5 Heaps

30. Digueu si els arbres binaris següents són min-heaps o no i perquè.

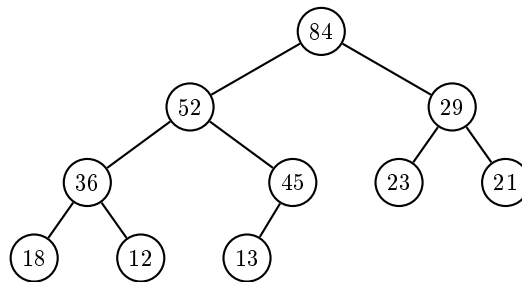


31. Digueu si els arbres binaris següents són max-heaps o no i perquè.



32. Partint d'un min-heap buit, inseriu successivament les claus 45, 67, 23, 46, 89, 65, 12, 34, 98, 76.

33. Partint d'un max-heap buit, inseriu successivament les claus 45, 67, 23, 46, 89, 65, 12, 34, 98, 76.
34. Partint del max-heap següent, elimineu successivament l'element màxim fins que quedi buit.



35. Donat el min-heap següent implementat en un taula,

7	11	9	23	41	27	12	29
---	----	---	----	----	----	----	----

dibuixeu l'arbre representat per la taula i, a continuació, dibuixeu l'evolució dels continguts del taula i de l'arbre representat, en aplicar successivament les operacions d'inserir l'element 3 i eliminar el mínim.

36. Considereu la definició de tipus següent per a arbres binaris:

```

struct node {
    node* fe;
    node* fd;
    int x;
};

typedef arbin node*;
  
```

Implementeu una funció `bool min_heap (arbin a)` que indiqui si l'arbre binari `a` és un min-heap. Quin és el seu cost?

37. Implementeu una funció `bool max_heap (arbin a)` que indiqui si l'arbre binari `a` és un max-heap. Quin és el seu cost?
38. Convertiu la taula següent en un min-heap tot aplicant l'algorisme de construcció de heaps de dalt cap a baix.

45	53	27	21	11	97	34	78
----	----	----	----	----	----	----	----

39. Convertiu la taula següent en un min-heap tot aplicant l'algorisme de construcció de heaps de baix cap a dalt.

45	53	27	21	11	97	34	78
----	----	----	----	----	----	----	----



40. Valideu o desmentiu les afirmacions següents:

- (a) “Una taula amb els elements ordenats de menor a major és un min-heap.”
- (b) “Inserir en un max-heap amb n elements té cost $\Theta(\log n)$ en el cas pitjor.”
- (c) “Trobar l’element màxim en un min-heap té cost $\mathcal{O}(\sqrt{n})$.”
- ★ “Eliminar el màxim d’una cua de prioritats de n elements amb prioritats diferents en un max-heap en taula té cost $\Theta(1)$ en el cas millor.”

41. En un heap k -ari amb $k \geq 2$, tot node té com a màxim k fills (els heaps habituals són doncs heaps 2-aris). Doneu una representació espacialment eficient d’un heap k -ari en una taula, tot descrivint la implementació de les operacions que permeten anar d’un element al seu pare i els seus k fills.

42. Implementeu la classe `CuaDePrio` següent per a cues de prioritats d’elements `Elem` tot utilitzant un min-heap en una taula. Tracteu els errors possibles.

```
template <typename Elem> class CuaDePrio {
public:
    CuaDePrio (nat M = 100);           // Crea una CuaDePrio buida per
M elements.
    CuaDePrio (CuaDePrio& q)           // Constructor de còpia
    ~CuaDePrio ();                     // Destructor
    CuaDePrio& operator= (CuaDePrio& q); // Operador d'assignació
    void afegir (Elem x);               // Afegeix un element x
    Elem treure_min ();                 // Treu i retorna l'element mínim
    Elem minim ();                      // Retorna un element amb prioritat mínima
    int talla ();                       // Retorna la talla de la cua
    bool buida ();                      // Indica si la cua és buida.
};
```

43. A continuació es dona una possible representació de la classe `CuaDePrio` del problema anterior amb la implementació del mètode `afegir()`.

```
template <typename Elem> class CuaDePrio {
private:
    nat M;           // Capacitat
    nat n;           // Nombre d'elements
    Elem* t;         // Taula on es guarda el heap (la posició 0 no s'utilitza)

    void surar (nat i) {
        if (i!=1 and t[i/2]>t[i]) {
            int x = t[i];  t[i] = t[i/2];  t[i/2] = x; // Δ
            surar(i/2);
        }
    }
```

```

    }    }

public:

    CuaDePrio (nat M_ = 100) : M(M_), n (0),
    t(new Elem[M_ + 1]) {
    }

    void afegir (Elem x) {
        if (n == M) throw ErrorImpl("CuaDePrio_lplena.");
        t[++n] = x;
        surar(n);
    }
};

```

Apliqueu al codi anterior les millores següents l'una rera l'altra:

- (a) Passeu a iteratiu el codi recursiu final per surar elements.
- (b) Elimineu la seqüència d'intercanvis de la línia Δ .
- (c) Utilitzeu la posició 0 de la taula com a sentinella per eliminar la condició $i!=1$ de la cerca.

44. Considereu el codi següent que llegeix n enters i els reescriu en ordre decreixent:

```

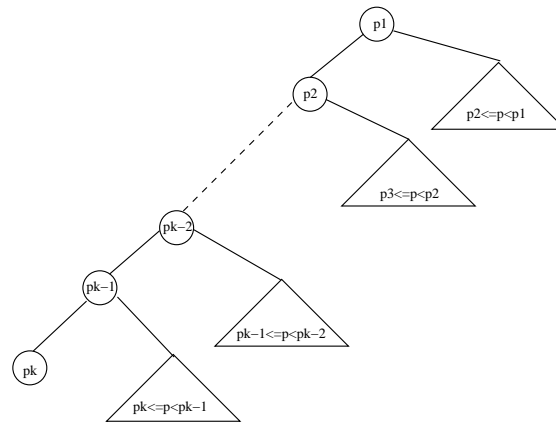
priority_queue<int> CP;
int x;
while (cin >> x) {
    CP.push(x);
}
while (not CP.empty()) {
    cout << CP.top() << endl;
    CP.pop();
}

```

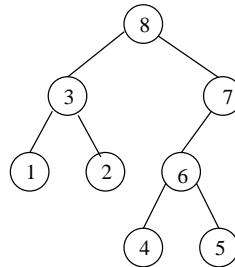
Digueu tant acuradament com pugueu quin és el cost en espai i quin és el cost en temps en el cas pitjor d'aquest algorisme.



45. Un *arbre de prioritats* és una estructura de dades que es pot fer servir per implementar cues de prioritat. Per simplificar, suposarem que totes les prioritats són diferents. Un arbre de prioritats és un arbre binari tal que per tot node, les prioritats dels nodes del seu fill dret estan entre la seva prioritat i la prioritat de l'arrel del seu fill esquerre. Observeu que si un node no té fill esquerre, llavors tampoc pot tenir fill dret.



La figura següent mostra un arbre de prioritats de 8 nodes format per la inserció successiva de les prioritats 3, 1, 8, 6, 4, 2, 7 i 5.



Donada la definició següent en C++

```
struct node {
    int prio;
    node* left;
    node* right;
};
typedef node* arbreprio;
```

(a) Implementeu en C++ una funció

arbreprio insert(arbreprio a, int p)

que donat un arbre de prioritats a i una prioritat p retorni l'arbre de prioritats resultant d'inserir un node amb prioritat p a l'arbre a .

(b) Escriuiu la recurrència per al cost en cas pitjor de la funció **insert** en funció del nombre n de nodes de l'arbre on es fa la inserció. Resol la recurrència. Justifica les teves respostes.

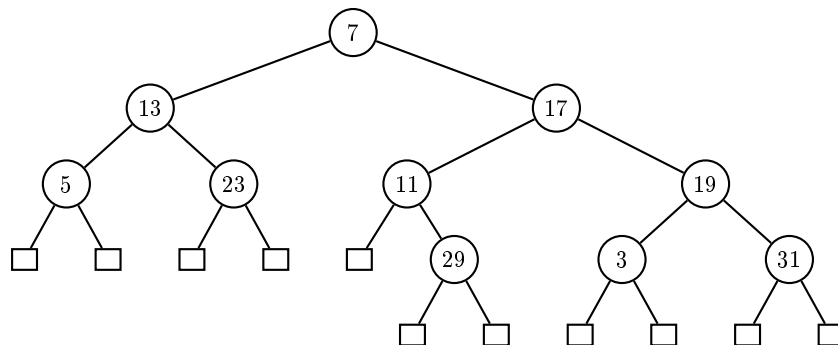
46. Dissenyau una estructura de dades per al conjunt dels enters amb les operacions i els costos temporals en cas pitjor següents. Podeu suposar que es coneix una fita superior M del nombre d'elements del conjunt.

Operació	Descripció	Cas mitjà	Cas pitjor
<code>S.add(x)</code>	Insereix l'element x en el conjunt S		$\Theta(\log n)$
<code>S.del(x)</code>	Esborrar l'element x del conjunt S		$\Theta(\log n)$
<code>S.member(x)</code>	Indica si l'element x és al conjunt S	$\Theta(1)$	$\Theta(\log n)$
<code>S.min</code>	Retorna l'element mínim del conjunt S		$\Theta(1)$
<code>S.max</code>	Retorna l'element màxim del conjunt S		$\Theta(1)$
<code>S.pred(x)</code>	Donat un element $x \in S$, retorna l'element màxim del conjunt $\{y \in S : y < x\}$	$\Theta(1)$	$\Theta(\log n)$
<code>S.succ(x)</code>	Donat un element $x \in S$, retorna l'element mínim del conjunt $\{y \in S : y > x\}$	$\Theta(1)$	$\Theta(\log n)$
<code>S.count(x, y)</code>	Retorna el nombre d'elements del conjunt $\{z \in S : x \leq z \leq y\}$		$\Theta(\log n)$

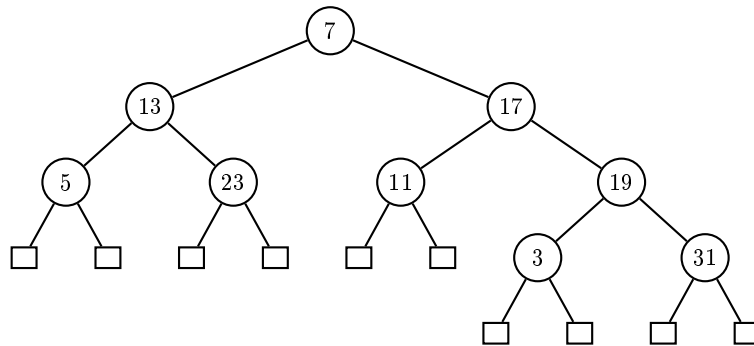
47. L'Édgar González (Ex-becari d'EDA, classificat 15è millor programador del món al Concurs de Programació ACM-IBM 2004) és un *freaky* a qui li agrada usar l'expressió “arbre caducifoli”.

Nosaltres definirem un arbre caducifoli com un arbre binari amb tots els nivells plens (excepte potser l'últim nivell, on els elements es troben el màxim a la dreta possible). Els arbres caducifolis tenen un mètode públic `cau_fulla()`, el qual esborra l'única fulla que fa que l'arbre segueixi sent caducifoli.

Per exemple, si apliquem `cau_fulla()` en aquest arbre



obtidrem l'arbre



Si continuéssim fent caure fulles de l'arbre, ho farien en l'ordre 3, 31, 5, 23, etc.

La representació interna d'un arbre és la següent:

```

class ArbreCaducifoli {
    struct node {                // Cada node té:
        int elem;                //      un element de tipus enter,
        node* esq, dre;          //      punters cap als dos fills.
    };

    node* arrel;                 // Arrel de l'arbre.
    int n;                       // Nombre d'elements.


public:
    void cau_fulla();
    ...
}
  
```

Implementeu en C++ el mètode `cau_fulla()`, suposant com a precondition que l'arbre no és buit. És imprescindible que el vostre mètode tingui cost proporcional a l'alçada de l'arbre.


Els únics atributs de la classe són el punter `arrel` i el comptador `n`; no se'n pot afegir cap altre. Tampoc podeu modificar la definició de `node`. Denoteu el punter nul amb `null`. Podeu implementar i usar mètodes privats si us calen.

48. Implementeu una funció `void ordenar (vector<Elem>& t)` que ordeni una taula `t` amb l'algorisme *heapsort* (els elements són a les posicions 0 a `t.size() - 1`).
49. Quin és el cost en temps i espai de l'algorisme *heapsort*?
50. És l'algorisme *heapsort* estable?
51. La STL requereix que l'operació `sort()` ha de tenir cost $(n \log n)$ en el cas pitjor i ha de ser *in-situ*. En canvi, l'operació `stable_sort()` ha de tenir cost $\mathcal{O}(n \log n)$ en el cas pitjor i ha de ser estable. Quins algorismes d'ordenació podeu utilitzar per implementar cadascuna d'aquestes operacions?

Busqueu quin algorisme d'ordenació utilitza la implementació de l'algorisme `sort` i de `stable_sort` de la STL al vostre ordinador.

52. Expliqueu per què no se solen implementar les cues de prioritat amb arbres equilibrats.
- ☆☆  53. Dissenyeu un algorisme de cost $\Theta(k_n \log n + n)$ que, donades n taules ordenades creixentment amb n elements cadascuna i un cert k_n , trobi el k_n -èsim element més petit global.
- ☆ 54. L'algorisme `partial_sort` de la STL rep un vector amb n elements i un valor m , $1 \leq m \leq n$, i reordena el vector de manera que els seus m primers elements són els m elements més petits del vector original, en ordre creixent.
- (a) Expliqueu com es pot implementar aquesta funció de manera que el seu cost en cas pitjor sigui $\mathcal{O}(n + m \log n)$.
 - (b) I com es podria implementar per a que el seu cost sigui $\mathcal{O}(n \log m)$?
 - (c) Quan convé usar la primera alternativa i quan la segona?

2.6 Particions

55. Definim $\log^* n$ com el nombre de vegades que cal aplicar l'operació de logaritme a un nombre n fins que sigui menor o igual que 1. Per exemple, $\log^* 16 = 3$ perquè $\log \log \log 16 = 1$.
- Per a quin número n tenim $\log^* n = 4$?
 - I per a quin número n tenim $\log^* n = 5$?
 - Quin és el límit de $\log^* n$ quan n tendeix a infinit?
 - Busqueu una aproximació al nombre d'àtoms de l'univers; quin és el seu \log^* ?
-  56. Considereu la següent seqüència d'instruccions aplicades sobre *mf-sets*:
- ```
crear(15), unir(1,2), unir(3,4), unir(3,5), unir(1,7), unir(3,6), unir(8,9),
unir(1,8), unir(3,10), unir(3,11), unir(3,12), unir(7,9), unir(3,13), unir(15,3),
unir(14,15).
```
- Simuleu l'algorisme bàsic.
  - Simuleu l'algorisme amb unió per talla.
  - Simuleu l'algorisme amb unió per alçada.
  - Simuleu l'algorisme amb unió per talla i compressió de camins.
  - Simuleu l'algorisme amb unió per alçada i compressió de camins.



```
}; // (nombre de cadenes)
```

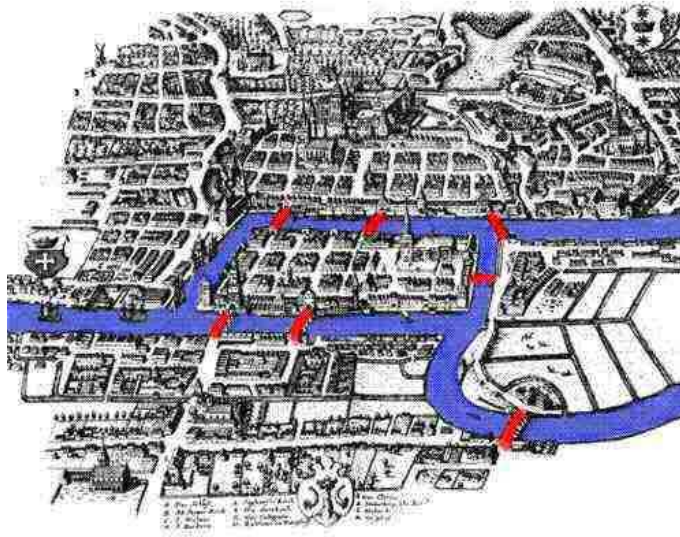
Observeu que cal afegir les claus `x` i `y` cada cop que aquestes no siguin dins de la partició.



# 3

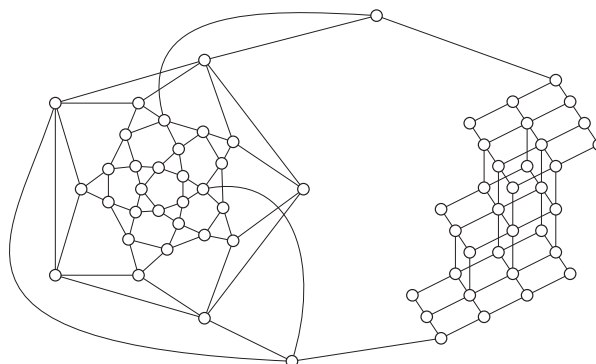
## Grafs

1. Es diu que Leonard Euler considerà el primer problema de grafs l'any 1735 a Königsberg (avui Kaliningrad), una ciutat que comptava amb 7 ponts com es veuen al mapa.



La gent de Königsberg volia saber si era possible o no fer un tomb per la vila de manera que es passés exactament un cop per cadascun dels ponts. Ajudeu els vilatans de Königsberg.

2. Digueu si el graf següent és Eulerià.



3. Dibuixeu el graf  $G = (V, E)$  donat per:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{\{1, 2\}, \{1, 4\}, \{3, 2\}, \{4, 5\}, \{5, 1\}, \{5, 2\}\}$

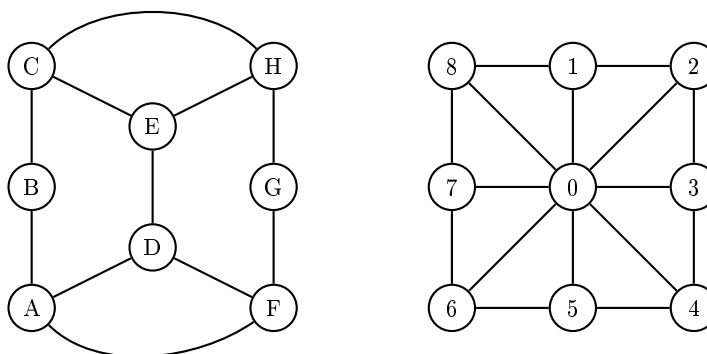
És connex? Si no ho és, quants components connexos té?

4. Dibuixeu el graf dirigit  $G = (V, E)$  donat per:

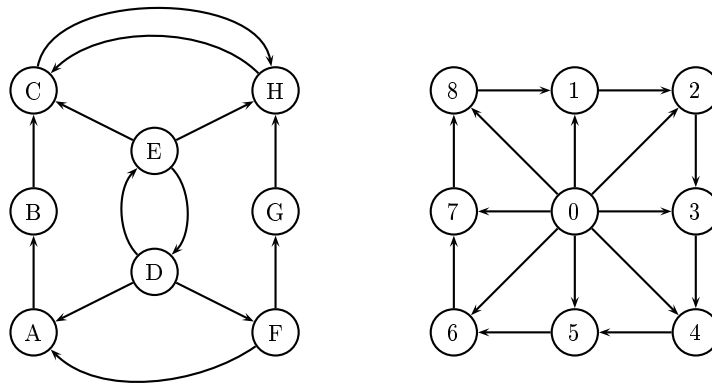
- $V = \{1, 2, 3, 4, 5\}$
- $E = \{(1, 2), (1, 4), (3, 2), (4, 5), (5, 1), (5, 2)\}$


És fortament connex? És feblement connex?

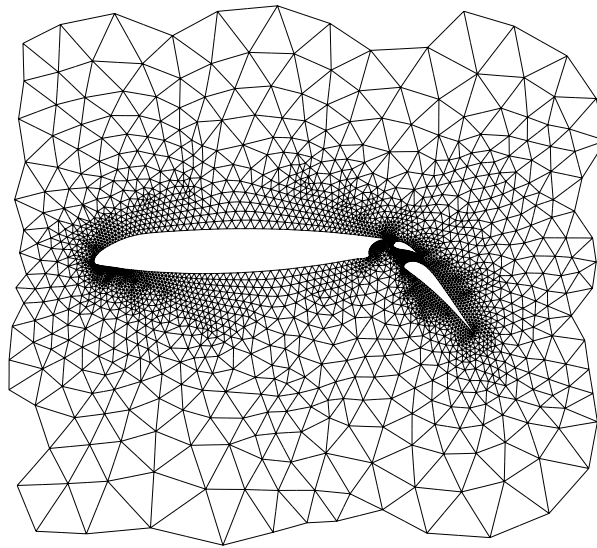
5. Mostreu com es representen els dos grafs següents utilitzant una matriu d'adjacència i llistes d'adjacència. Compteu el grau de cada vèrtex. Calculeu el diàmetre de cada graf.



6. Mostreu com es representen els grafs dirigits següents utilitzant una matriu d'adjacència i llistes d'adjacència. Compteu el grau d'entrada i de sortida de cada vèrtex. Digueu si els grafs són fortament o feblement connexos.

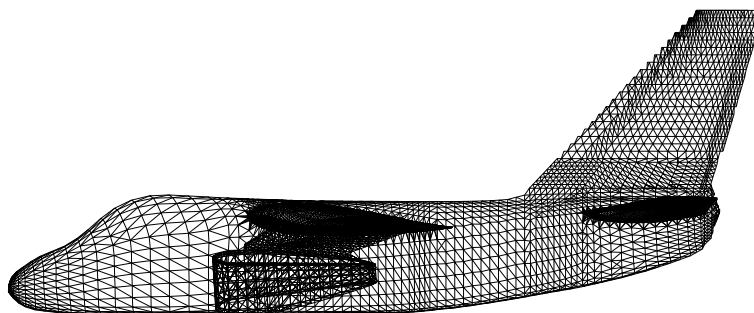


-  7. Demostreu que en un graf no dirigit  $G = (V, E)$  es té  $\sum_{u \in V} \text{grau}(u) = 2|E|$ .
8. Considereu un graf no dirigit amb cinc vèrtexs, tots de grau tres. Si existeix tal graf, feu-ne un dibuix. Si no n'existeix cap, doneu una explicació enraonada.
9. El graf de la figura següent té  $n = 4253$  vèrtexs i  $m = 12289$  arestes.

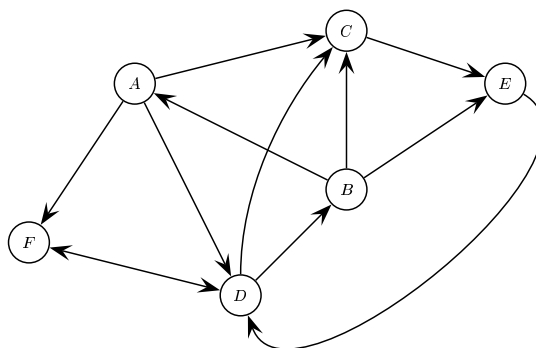


Suposeu que al vostre ordinador cada booleà ocupi un byte, cada enter quatre i cada punter quatre. Digueu quanta memòria cal per emmagatzemar aquest graf utilitzant una representació amb matriu d'adjacència i utilitzant una representació amb llistes d'adjacència.

10. Repetiu el problema anterior per al graf següent, que té  $n = 156317$  vèrtexs i  $m = 1059331$  arestes.



11. Llisteu totes les possibles seqüències de visita dels vèrtexs d'aquest graf dirigit, tot aplicant un recorregut en profunditat que comenci en el vèrtex  $E$ .

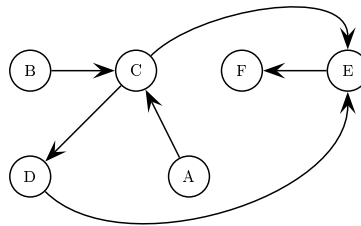


12. Repetiu el problema anterior tot aplicant un recorregut en amplitada que comenci en el vèrtex  $B$ .
13. (a) Dissenyeu i analitzeu un algorisme per calcular el grau d'un vèrtex donat en un graf no dirigit representat amb una matriu d'adjacència.
- (b) Dissenyeu i analitzeu un algorisme per calcular el grau d'un vèrtex donat en un graf no dirigit representat amb llistes d'adjacència.
- (c) Dissenyeu i analitzeu un algorisme per calcular el grau d'entrada i de sortida d'un vèrtex donat en un graf dirigit representat amb una matriu d'adjacència.
- (d) Dissenyeu i analitzeu un algorisme per calcular el grau d'entrada i de sortida d'un vèrtex donat en un graf no dirigit representat amb llistes d'adjacència.
- (e) Dissenyeu i analitzeu un algorisme per calcular el grau d'entrada i de sortida de tots els vèrtexs en un graf dirigit representat amb una matriu d'adjacència.
- (f) Dissenyeu i analitzeu un algorisme per calcular el grau d'entrada i de sortida de tots els vèrtexs en un graf no dirigit representat amb llistes d'adjacència.
- (g) Dissenyeu i analitzeu un algorisme per calcular el grau d'entrada i de sortida de tots els vèrtexs en un graf dirigit representat amb llistes d'adjacència.

14. Considereu la funció següent sobre un graf no dirigit  $G = (V, E)$  amb  $n = |V|$  vèrtexs i  $m = |E|$  arestes.

```
int misteri(const graph& G) {
 int compt = 0;
 forall_vertex(u, G)
 forall_adj_vertex(w, G[u])
 ++compt;
 return compt;
}
```

- Expliqueu breument què retorna la funció.
  - Quin cost té la funció, suposant que el graf  $G$  està implementat amb llistes d'adjacència?
15. Doneu, en ordre lexicogràfic i una a sota l'altra, totes les possibles ordenacions topològiques del graf dirigit acíclic següent:



16. Si al graf anterior li afegíssim un vèrtex aïllat, quantes ordenacions topològiques tindria? (No les llisteu, només digueu quantes i expliqueu perquè.)



17. La transposició d'un graf dirigit  $G = (V, E)$  consisteix en obtenir un nou graf dirigit  $G^T = (V, E^T)$  on  $E^T = \{(u, v) \mid (v, u) \in E\}$ . Dissenyeu i analitzeu un algorisme que transposi un graf dirigit representat amb una matriu d'adjacència.


Feu el mateix amb un graf dirigit representat amb llistes d'adjacència.

18. El quadrat d'un graf  $G = (V, E)$  és un altre graf  $G^Q = (V, E^Q)$  on  $E^Q = \{\{u, v\} \mid \exists w \in V \mid \{u, w\} \in E \wedge \{w, v\} \in E\}$ . Dissenyeu i analitzeu un algorisme que, donat un graf representat amb una matriu d'adjacència, calculi el seu quadrat.

Feu el mateix amb un graf representat amb llistes d'adjacència.




19. En un graf no dirigit, un quadre és un cicle de llargada 4. Dissenyeu un algorisme que, donat un graf no dirigit, digui si aquest conté algun quadre. Analitzeu la seva eficiència en diferents representacions.


- ☆☆  20. En una festa, un convidat es diu que és una *celebritat* si tothom el coneix, però ell no coneix a ningú (tret d'ell mateix). Les relacions de coneixença donen lloc a un graf dirigit: cada convidat és un vèrtex, i hi ha un arc entre  $u$  i  $v$  si  $u$  coneix a  $v$ .

Doneu un algorisme que, donat un graf dirigit representat amb una matriu d'adjacència, indica si hi ha o no cap celebritat. En el cas que hi sigui, cal dir qui és. El vostre algorisme ha de funcionar en temps  $\mathcal{O}(n)$ , on  $n$  és el nombre de vèrtexs.

21. Dissenyeu i analitzeu un algorisme que calculi el nombre de components connexos d'un graf no dirigit.

-  22. Dissenyeu i analitzeu un algorisme que en temps  $\mathcal{O}(|V|)$  determini si un graf no dirigit conté cicles o no.

23. Dissenyeu i analitzeu un algorisme que determini si un graf dirigit conté cicles o no.

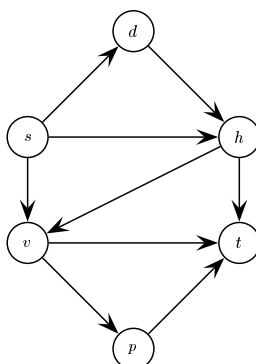
-  24. Una *ordenació topològica inversa* d'un graf dirigit acíclic és una seqüència formada per tots els vèrtexs del graf tal que si  $(v, w)$  és una aresta del graf, llavors  $w$  apareix abans que  $v$  a la seqüència. A partir del recorregut en profunditat, dissenyeu i analitzeu un algorisme que obtingui una ordenació topològica inversa d'un graf dirigit i acíclic.

25. Dissenyeu un algorisme de cost  $\mathcal{O}(|V|)$  que determini si un graf no dirigit  $G = (V, E)$  és un arbre o no.

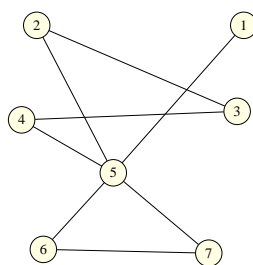
26. Dissenyeu un algorisme de cost  $\mathcal{O}(|V|)$  que determini si un graf dirigit  $G = (V, E)$  és un arbre arrelat o no.

27. Sigui  $G = (V, E)$  un graf dirigit amb  $m = |E|$ ,  $n = |V|$  i  $m > n$ . Fixats dos vèrtexs del graf, per exemple  $s$  i  $t$ , cal construir un conjunt de camins disjunts de  $s$  a  $t$ . Els camins del conjunt no comparteixen cap vèrtex (excepte  $s$  i  $t$ ) i es demana que el conjunt sigui *maximal*. En aquest cas *maximal* vol dir que si afegim qualsevol altre camí al conjunt, els camins deixen de ser disjunts. Dissenyeu un algorisme que resolgui el problema en temps  $\mathcal{O}(m)$ .

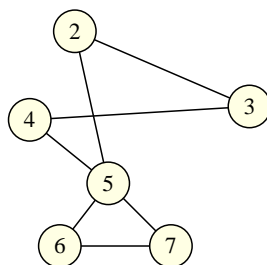
Per exemple, al graf següent, alguns dels conjunts maximals possibles són:  
 $C_1 = \{\langle s, d, h, t \rangle, \langle s, v, p, t \rangle\}$ ,  $C_2 = \{\langle s, h, v, t \rangle\}$ ,  $C_3 = \{\langle s, h, v, p, t \rangle\}$  i  
 $C_4 = \{\langle s, h, t \rangle, \langle s, v, t \rangle\}$ .



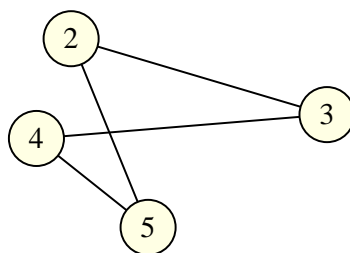
- ☆☆ 28. Sigui  $G = (V, E)$  un graf dirigit acíclic i sigui  $k$  la llargada del camí més llarg en  $G$ . Dissenyeu i analitzeu un algorisme que particioni el conjunt  $V$  en, com a màxim,  $k + 1$  grups de vèrtexs de manera que per a tot parell de vèrtexs  $u, w$  diferents i pertanyent al mateix grup no hi hagi camí de  $u$  a  $w$  ni de  $w$  a  $u$ . Una bona solució té cost lineal.
29. Una expressió aritmètica pot representar-se com un graf dirigit acíclic. Expliqueu com avaluar-la.
30. Demostreu que tot graf no dirigit i connex té, com a mínim, un vèrtex  $u$  tal que si eliminem el vèrtex  $u$  i totes les seves arestes incidents, el graf resultant continua sent connex. Dissenyeu i analitzeu un algorisme que trobi un vèrtex d'aquestes característiques.
- ☆☆ 31. Sigui  $G = (V, E)$  un graf no dirigit complet. Sigui  $G' = (V, E')$  un graf dirigit en el que  $E'$  conté les mateixes arestes que  $E$  però assignant una orientació arbitrària a cadascuna d'elles. Demostreu que  $G'$  conté com a mínim un camí Hamiltonià. Dissenyeu i analitzeu un algorisme que calculi tal camí.
- ☆☆ 32. Es diu que un vèrtex d'un graf connex és un punt d'articulació del mateix si en suprimir aquest vèrtex i totes les aristes que hi incideixen, el graf resultant deixa de ser connex. Per exemple, un graf en forma d'anell no té cap punt d'articulació mentre que tot node que no sigui fulla d'un arbre és punt d'articulació. Dissenyeu un algorisme que donat un graf connex no dirigit  $G = (V, E)$ , indiqui quins vèrtexos del graf són punts d'articulació. Calculeu el seu cost. Indiqueu quina implementació del graf és la que proporciona un algorisme més eficient.
33. Donat un graf  $G = (V, E)$  no dirigit amb  $n = |V|$ , dissenyeu un algorisme que donada una constant natural  $k \geq 0$  calculi el *subgraf induït màxim*  $H = (U, F)$ , si existeix. El subgraf induït màxim  $H$  ha de complir que tots els seus vèrtexos tinguin grau més gran o igual que  $k$ . Feu-ne una anàlisi de l'eficiència.
- Ajut: Comenceu l'anàlisi resolent aquests tres casos:  $n \leq k$ ,  $n = k + 1$  y  $n > k + 1$ . Per exemple, donat el graf següent:



el subgraf induït màxim quan  $k = 2$  és



Aquest també és un subgraf induït de grau 2 però no és màxim:



- ☆☆ 34. Sigui  $G = (V, E)$  un graf no dirigit i connex. Dissenyeu i implementeu un algorisme eficient per determinar un *conjunt base dels cicles simples* del graf. Cal proporcionar les arestes que formen cadascun dels cicles simples. Noteu que una aresta pot pertànyer a més d'un cicle simple. Per facilitar l'escriptura de l'algorisme, la solució es retornarà en una llista els elements de la qual seran els cicles simples trobats, on cada cicle serà la llista d'arestes que el componen.

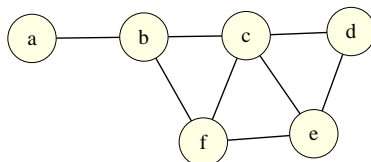
Definicions necessàries:

**Cicle simple:** És una seqüència  $S$  de vèrtexos, tots ells diferents excepte els extrems que són iguals, tal que  $S = \langle v_1, v_2, \dots, v_i, \dots, v_r \rangle$ ,  $|S| \geq 4$  i compleix que per a tot  $i$ ,  $1 \leq i < r$ ,  $(v_i, v_{i+1}) \in E$ .



**Conjunt base:** Un conjunt de cicles simples és base si la uni ó de les arestes que conté coincideix exactament amb el conjunt de totes les arestes de  $G$  que formen part d'algun cicle.

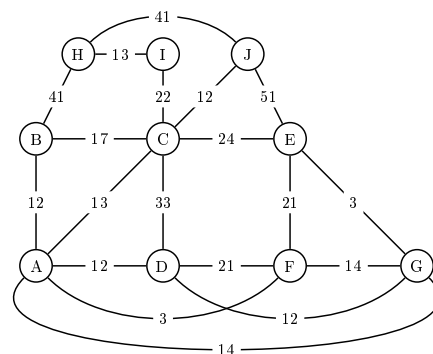
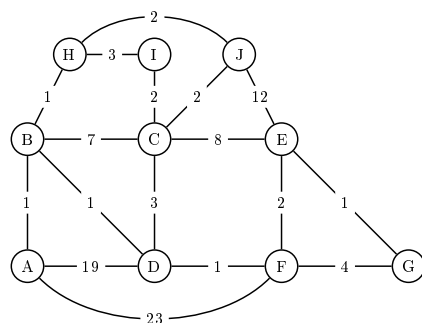
Per exemple, donat el graf de la figura següent,



un possible conjunt base seria

$$B = \{ \langle (b, c), (c, d), (d, e), (e, f), (f, b) \rangle, \\ \langle (c, d), (d, e), (e, f), (f, c) \rangle, \\ \langle (c, d), (d, e), (e, c) \rangle \}.$$

- ☆☆ 35. Donat un graf dirigit  $G = (V, E)$  i dos dels seus vèrtexos  $p$  i  $q$ , dissenyeu un algorisme que retorni una llista amb tots els camins elementals que van de  $p$  a  $q$  en  $G$ . Una bona versió recorre cada aresta un sol cop. Raoneu la seva correctesa i analitzeu-ne el cost. Recordeu que un camí elemental és aquell en què no es repeteixen vèrtexos.
- ☆☆ 36. Dissenyau un algorisme eficient per trobar el que es denomina *enllaços crítics* en una xarxa d'ordinadors, és a dir, aquelles connexions entre nodes que en cas d'avaría deixarien parts de la xarxa desconnectades de la resta. No hi ha més d'un enllaç entre dos nodes qualssevol de la xarxa, però si cap enllaç no falla, la xarxa és connexa. Calculeu l'eficiència de l'algorisme en termes del número de nodes  $n$  i el número d'enllaços  $m$  entre nodes. No es considerarà vàlida una solució consistent en llevar una aresta, comprovar si el graf continua essent connex o no, repetir el mateix amb una altra aresta, i amb una altra, i amb una altra, ...
- ☆☆ 37. Com pot variar el número de components fortament connexos d'un graf dirigit en afegir-hi un nou arc? Justifiqueu la vostra resposta. Convé emprar en el raonament el denominat graf de components  $G^{SCC}$ . Aquest graf té un vèrtex per cada component fortament connex del graf original  $G$ . Existeix un arc entre dos vèrtexos de  $G^{SCC}$ , si existeix un arc en  $G$  que uneix dos vèrtexos dels corresponents components fortament connexos.
38. Dissenyau un algorisme que, donat un graf no dirigit  $G = (V, E)$  i un subconjunt  $V' \subseteq V$  dels seus vèrtexs, trobi el mínim de les distàncies entre qualssevol dos vèrtexs en  $V'$  (si no hi ha cap camí entre els vèrtexs de  $V'$ , retorneu  $+\infty$ ). Per exemple, en el graf següent, caldria retornar 2 quan  $V' = \{A, G, I\}$ .



Suposeu que el graf es troba representat per llistes d'adjacència. El vostre algorisme ha de tenir cost  $(|V|(|V|+|E|))$ . Formuleu el vostre algorisme amb un grau de detall prou gran perquè tots els seus elements quedin ben clars. Argumenteu la correctesa de la vostra solució.

39. Dissenyeu un algorisme que, donat un graf no dirigit  $G = (V, E)$  i un vèrtex  $u \in V$ , calculi la llargada del cycle més curt que passi per  $u$  (si no n'hi ha cap, retorneu  $+\infty$ ).

*Ajut:* Penseu en esborrar  $u$  de  $G$  i utilitzar l'algorisme del problema anterior tot escollint un conjunt  $V'$  adequat.

40. Escriu un algorisme en pseudocodi o C++ que donat un graf acíclic dirigit (DAG)  $G = \langle V, E \rangle$  calculi quantes *arrels* i quants *fulles* conté el DAG  $G$ . Un vèrtex  $v$  és una *arrel* del DAG si el seu grau d'entrada és 0, i és un *fulla* si el seu grau de sortida és 0. El graf està implementat amb llistes d'adjacència: per a cada vèrtex  $v$  tenim una llista enllaçada amb els seus successors.

Calcula el cost de l'algorisme desenvolupat en funció de  $n = |V|$  i  $m = |E|$ , i justifica la seva correctesa.

41. Es disposa d'un tipus **grafc** que representa grafos no dirigits  $G = \langle V, E \rangle$  ón cada aresta  $e = (u, v) \in E(G)$  té un color, que podem consultar mitjançant la funció  $G.\text{color}(u, v)$ . Aquesta operació té cost  $\Theta(1)$ . Escriviu un algorisme en pseudocodi o C++ que donat un d'aquests grafos  $G$ , retorni un vector de booleans  $H$  tal que  $H[i]$  és cert si i només si totes les arestes de la component connexa  $i$ -èssima són del mateix color. Addicionalment, l'algorisme haurà de retornar un vector d'enters  $CC$  tal que  $CC[u]$  és el número de la component connexa a la qual pertany el vèrtex  $u$ . La numeració de les components connexes del graf va de 0 endavant, però tret d'això, és totalment arbitrària, a elecció de l'algorisme.

Justifiqueu la correctesa del vostre algorisme i calculeu la seva eficiència en funció de  $n = |V|$  i  $m = |E|$ .

---

## Algorismes de dividir i vèncer

1. Escriviu un algorisme de cost  $\Theta(\log n)$  que, donada una taula ordenada  $T$  amb  $n$  elements i un element  $x$ , retorni  $-1$  si  $x$  no és en  $T$  o un índex  $i$  tal que  $T[i] = x$  altrament. Si heu escrit el vostre algorisme recursivament, passeu-lo a iteratiu.



2. Considereu el problema de, donada una taula ordenada  $T$  amb  $n$  elements i un element  $x$ , retorna  $-1$  si  $x$  no és en  $T$  o un índex  $i$  tal que  $T[i] = x$  altrament. Digueu què en penseu dels tres fragments de codi següents (el fragment 1 es troba en una web de programadors, el fragment 2 és una adaptació del codi del llibre *Modern software development using Java* de Tymann i Schneider i el fragment 3 és una adaptació del codi del llibre *Developing Java software* de Winder i Roberts):

```
// Fragment 1
int lookup1 (vector<int>& T, int x) {
 int l = 0;
 int r = T.size() - 1;
 while (l <= r) {
 int m = (l+r) / 2;
 if (x < T[m]) r = m;
 else if (x > T[m]) l = m + 1;
 else return m;
 }
 return -1;
}


// Fragment 2
int lookup2 (vector<int>& array, int target) {
 int start = 0;
```

```

 int end = array.size();
 int position = -1;
 while (start <= end and position == -1) {
 int middle = (start+end)/2;
 if (target < array[middle]) end = middle - 1;
 else if (target > array[middle]) start = middle + 1;
 else position = middle;
 }
 return position;
}

// Fragment 3
int lookup2 (vector<int>& v, int o) {
 int hi = v.size();
 int lo = 0;
 while (true) {
 int centre = (hi+lo)/2;
 if (centre == lo) {
 if (v[centre] == o) return centre;
 else if (v[centre+1] == o) return centre+1;
 else return -1;
 }
 if (v[centre] < o) lo = centre;
 else if (o < v[centre]) hi = centre;
 else return centre;
 }
}

```

3. Escriviu un algorisme de cost  $\Theta(\log n)$  que, donada una taula ordenada  $T$  amb  $n$  elements i un element  $x$ , retorni  $-1$  si  $x$  no és en  $T$  o l'índex de la primera posició de  $T$  que conté  $x$ .
4. Escriviu un algorisme de cost  $\Theta(\log n)$  que, donada una taula ordenada  $T$  amb  $n$  elements i dos elements  $x$  i  $y$  amb  $x \leq y$ , retorni el nombre d'elements en  $T$  que es troben entre  $x$  i  $y$  ( $x$  i  $y$  inclosos).
-  5. Dissenyeu un algorisme de cerca *ternària* en un vector. La idea consisteix en partir el vector en tres parts i cridar recursivament sobre la part que calgui.  
Analitzeu el cost del vostre algorisme i compareu-lo amb el de l'algorisme de cerca binària.
6. El cost d'una cerca lineal en una taula i en una llista és lineal. El cost d'una cerca binària en una taula ordenada és logarítmic. Es pot fer una cerca binària en una llista ordenada?
7. Ordeneu la taula  $\langle 3, 8, 15, 7, 12, 6, 5, 4, 3, 7, 1 \rangle$  amb l'algorisme d'ordenació per fusió recursiu i amb l'algorisme d'ordenació per fusió d'avall cap amunt.
8. És l'algorisme d'ordenació per fusió estable? De què depèn?

9. Digueu quant triga l'ordenació per fusió quan l'entrada es troba...

- permutada a l'atzar,
- ordenada,
- ordenada del revés,
- amb tots els elements iguals.

10. Quantes crides recursives cal guardar com a màxim en un instant donat a la pila per a ordenar per fusió una taula de  $n$  elements?

11. Escriviu l'algorisme d'ordenació per fusió usant fusions de tres subtaules en lloc de dues. Quin cost té?

12. Escriviu l'algorisme d'ordenació per fusió implementat d'avall cap amunt, amb ordenació per inserció de les subtaules petites.

☆☆ 13. Una manera d'accelerar l'algorisme d'ordenació per fusió és eliminar el temps necessari per copiar els elements fusionats de la taula auxiliar a la taula original. Per aconseguir-ho, es pot fer que els papers de la taula original i la taula auxiliar s'inverteixin a cada nivell. Escriviu aquest algorisme millorat.

14. Ordeneu la taula  $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$  amb l'algorisme d'ordenació ràpida i la partició de Hoare.

15. És estable l'algorisme de partició de Hoare?



16. Digueu quant triga l'algorisme d'ordenació ràpida amb la partició de Hoare quan l'entrada es troba...

- permutada a l'atzar,
- ordenada,
- ordenada del revés,
- amb tots els elements iguals.

17. Sigui  $T[i..j]$  una taula amb  $n = j - i + 1$  elements. Considereu l'algorisme d'ordenació anomenat *l'ordenació boja d'en Quim*:

- Si  $n \leq 2$ , la taula s'ordena trivialment.
- Si  $n \geq 3$ , "dividim" la taula en tres intervals  $T[i..k-1]$ ,  $T[k..\ell]$  i  $T[\ell+1..j]$ , on  $k = i + \lfloor n/3 \rfloor$ ,  $\ell = j - \lfloor n/3 \rfloor$ . L'algorisme ordena recursivament  $T[i..\ell]$ , després ordena  $T[k..j]$ , i finalment ordena de nou  $T[i..\ell]$ .


Demostreu que aquest algorisme ordena correctament. Doneu, raonadament, una recurrència per al temps que triga aquest algorisme i resoleu-la.

-  18. Escriviu un algorisme que compti el nombre d'inversions d'una taula amb  $n$  elements en temps  $\mathcal{O}(n \log n)$  en el cas pitjor.
- ☆  19. Sigui  $t[1..n]$  una taula d'enters. Dissenyeu un algorisme de cost  $\mathcal{O}(n \log n)$  que compti el nombre d'elements del conjunt

$$\left\{ i \in 1 \dots n : \left( \sum_{j \neq i, t[j] < t[i]} t[j] \right) < t[i] \right\}.$$

Dissenyeu un algorisme de cost  $\mathcal{O}(n \log n)$  que compti el nombre d'elements del conjunt

$$\left\{ i \in 1 \dots n : \left( \sum_{j < i, t[j] < t[i]} t[j] \right) < t[i] \right\}.$$

- ☆☆ 20. Dissenyeu un algorisme de cost  $\mathcal{O}(n \log n)$  tal que, donada una taula amb  $n$  enters, calculi la subtaula consecutiva no buida tal que la seva suma és el més propera possible a 0.
- ☆☆ 21. Trobeu un algorisme de cost  $\mathcal{O}(n)$  per al problema anterior suposant aquest cop que la taula està ordenada.
- ☆☆ 22. Dissenyeu algorismes de cost  $\mathcal{O}(n \log n)$  que comptin el nombre d'elements dels conjunts definits en els apartats següents:
- $\{i \in 1 \dots n : |\{j : j \neq i \wedge t[j] < t[i]\}| = i\}$
  - $\{i \in 1 \dots n : |\{j : j \neq i \wedge t[j] < t[i]\}| = t[i]\}$
  - $\{i \in 1 \dots n : |\{j : j < i \wedge t[j] < t[i]\}| = i\}$
  - $\{i \in 1 \dots n : |\{j : j < i \wedge t[j] < t[i]\}| = t[i]\}$
  - $\{i \in 1 \dots n : |\{j : j < i \wedge t[j] < t[i]\}| = |\{j : j > i \wedge t[j] < t[i]\}|\}$
  - $\{i \in 1 \dots n : |\{j : j < i \wedge t[j] < t[i]\}| = |\{j : j < i \wedge t[j] > t[i]\}|\}$
- ☆☆ 23. Dissenyeu un algorisme de cost  $\Theta(\log n)$  que, donades dues taules ordenades creixentment de  $n$  elements cadascuna i un cert  $k$ , trobi el  $k$ -èsim element més petit global.
- ☆☆ 24. Repetiu l'exercici anterior pero suposant ara que tenim 3 taules ordenades d'entrada.
25. Dissenyeu i analitzeu un algorisme que solucioni el problema de les torres de Hanoi.
-  26. Sigui  $V$  un vector de  $n$  enters en ordre creixent. Dissenyeu un algorisme  $\mathcal{O}(\log n)$  que retorni un índex  $i$  tal que  $V[i] = i$  si aquest existeix,  $-1$  altrament.



27. Aquest problema estudia un algorisme de dividir i vèncer per multiplicar dos números de  $n = 2^k$  bits. Recordeu que l'algorisme escolar utilitza  $\Theta(n^2)$  passos.

- a) Siguin  $x$  i  $y$  dos números de  $n$  bits cadascun de forma que  $x = a2^{n/2} + b$  i que  $y = c2^{n/2} + d$ . Comproveu que  $xy = (a2^{n/2} + b)(c2^{n/2} + d)$ .

Utilitzant aquesta idea, dissenyeu un algorisme de dividir i vèncer per reduir la multiplicació de dos nombres de  $n$  bits a quatre multiplicacions de nombres de  $n/2$  bits que es combinen mitjançant addicions i desplaçaments.

Analitzeu el cost de l'algorisme resultant.

- b) L'any 1962, Karatsuba i Ofman van descobrir una forma molt astuta per reduir les anteriors quatre multiplicacions de nombres de  $n/2$  bits a només tres: Comproveu que  $xy = ((a + b)(c + d) - ac - bd)2^{n/2} + ac2^n + bd$ .

Utilitzant aquesta idea, dissenyeu un nou algorisme de dividir i vèncer i analitzeu-ne el cost.

- c) Com es generalitzen els algorismes anteriors quan  $n$  no és una potència de 2?



28. Aquest problema estudia un algorisme de dividir i vèncer per multiplicar dues matrius  $n \times n$ . Recordeu que l'algorisme clàssic utilitza  $\Theta(n^3)$  passos.

Utilitzant la vella dita del “fila per columna”, el producte de dues matrius  $2 \times 2$  s'obté amb 8 productes de reals:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}.$$

Cap a l'any 1967, Strassen va descobrir que aquest producte de matrius es podia fer utilitzant només 7 productes de reals. Per això, va definir

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * (b_{00})$$

$$m_3 = (a_{00}) * (b_{01} - b_{11})$$

$$m_4 = (a_{11}) * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * (b_{11})$$

$$m_6 = (a_{10} + a_{11}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} + a_{11}) * (b_{10} + b_{11})$$

i va comprovar que

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}.$$


L'avantatge d'aquest fet és que, aplicant la idea recursivament, podem multiplicar dues matrius grans amb un temps inferior al  $\Theta(n^3)$  de l'algorisme clàssic: Suposeu

que tenim dues matrius  $A$  i  $B$  de talla  $n \times n$  cadascuna essent  $n$  una potència de 2. Llavors podem decompondre el producte de les dues matrius en quatre blocs de la mateixa talla de la forma següent:

$$\left( \begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array} \right) = \left( \begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right) * \left( \begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array} \right).$$

No és difícil veure que hom pot tractar els blocs d'aquestes matrius com si fossin números per obtenir el resultat correcte. Per exemple, la matriu  $C_{00}$  de talla  $\frac{n}{2} \times \frac{n}{2}$  correspon al producte de matrius  $M_1 + M_4 - M_5 + M_7$  on els  $M_i$  venen donats per les fórmules de Strassen, substituïnt números per matrius.

- a) Suposant que  $n$  és una potència de 2, quants passos requereix l'algorisme descrit per multiplicar dues matrius  $n \times n$ ?
- b) Quina senzilla modificació aplicariéu a l'algorisme per multiplicar matrius  $n \times n$  quan  $n$  no és una potència de 2? Quin cost tindria l'algorisme llavors?
- c) Doneu una fita inferior per al problema de la multiplicació de dues matrius  $n \times n$ .

- ★  29. El problema de la Subseqüència Màxima consisteix en, donada una taula  $A$  amb  $n$  enters, determinar la suma màxima que es pot trobar en qualsevol subtaula consecutiva de l'entrada. Per exemple, si la taula  $t$  conté els 10 elements

31, -41, 59, 26, -53, 58, 97, -93, -23, 84

llavors la Subseqüència Màxima és 187, que correspon a la suma dels elements  $A[2 \dots 6]$ . Formalment, es vol calcular

$$\max \left\{ \sum_{k=i}^j A[k] : 0 \leq i < n, 0 \leq j < n \right\}.$$

Fixeu-vos que es pot triar una subtaula buida, que té suma 0.

Escriviu un algorisme  $\Theta(n \log n)$  per resoldre el problema de la Subseqüència Màxima utilitzant una aproximació de dividir i vèncer.

☆☆ Escriviu un algorisme  $\Theta(n)$  per resoldre el mateix problema.

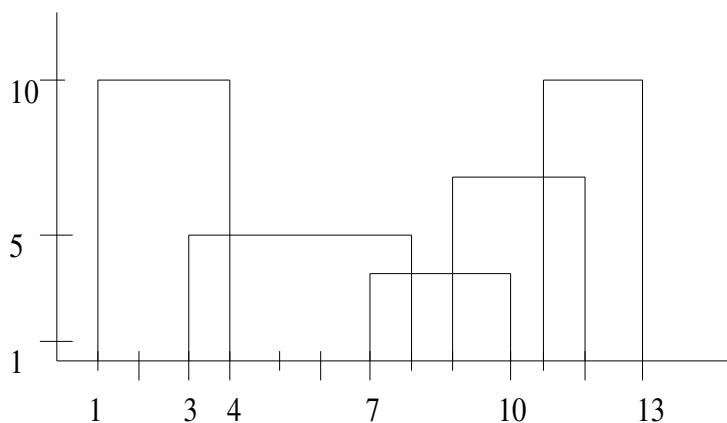
30. Donats  $n$  valors reals  $(v_1, \dots, v_n)$  dissenyar un algorisme que calculi quant val la diferència  $d$  entre els dos valors més propers:

$$d = \min_{\substack{1 \leq i, j \leq n \\ i \neq j}} \{|v_i - v_j|\}.$$

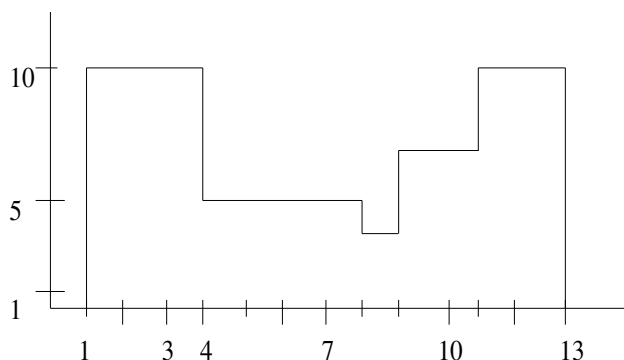
Suggeriment: Feu servir l'algorisme de partició del quicksort.



- ☆☆ 31. Dada la localización exacta y la altura de varios edificios rectangulares de la ciudad, obtener la *skyline*, línea de recorte contra el cielo, que producen todos los edificios eliminando las líneas ocultas. Ejemplo: La entrada es una secuencia de edificios y un edificio se caracteriza por una tripleta de valores  $(x_{\min}, x_{\max}, h)$ . Una posible secuencia de entrada para los edificios de la siguiente figura podría ser:  $(7,10,3)$ ,  $(9,12,7)$ ,  $(1,4,10)$ ,  $(3,8,5)$  y  $(11,13,10)$ .



Y ésta es la representación de la skyline de la salida que se debe producir:



Su secuencia asociada es :  $(1,4,10)$ ,  $(4,8,5)$ ,  $(8,9,3)$ ,  $(9,11,7)$  y  $(11,13,10)$ .

32. Donada una taula  $v = (v_0, \dots, v_{n-1})$  amb  $n$  valors reals, dissenyeu un algorisme que calculi quant val la diferència  $d$  entre els dos valors més propers:

$$d = \min_{0 \leq i, j, i \neq j} |v_i - v_j|.$$

- ☆ 33. Dissenyau un algorisme eficient per al problema de la selecció múltiple: Donats un vector  $A[1..n]$  d'elements amb  $n > 0$  i un vector  $j[1..p]$  d'enters amb  $p > 0$ , on  $1 \leq j[1] < j[2] < \dots < j[p] \leq n$ , cal trobar els elements  $j[1]$ -èsim,  $j[2]$ -èsim,  $\dots$ ,  $j[p]$ -èsim del vector  $A$  si aquest estigués ordenado. La solució proposada ha de ser més “eficient” que l'evident d'ordenar el vector  $A$  o que la d'iterar un algorisme de selecció  $p$  vegades, una vegada amb cada valor  $j[i]$  donat. Amb tot, no es demana demostrar que l'algorisme proposat és efectivament més eficient que les dues solucions trivials esmentades ni que se'n calculi el cost (és un problema matemàticament complex).

Exemple: El vector  $A$  conté elements següents:  $A[1] = 8$ ,  $A[2] = 14$ ,  $A[3] = 5$ ,  $A[4] = 7$ ,  $A[5] = 3$ ,  $A[6] = 1$ ,  $A[7] = 25$  i  $A[8] = 2$ . El vector  $j$ , que està ordenat creixentment, conté els índexos dels elements de  $A$  que es vol obtenir si  $A$  estigués ordenat. Per exemple,  $j[1] = 2$ ,  $j[2] = 5$  i  $j[3] = 7$ . Aleshores, el resultat és  $(2, 7, 14)$ .

34. L'algorisme d'ordenació ràpida, quicksort, no és massa eficient per ordenar cadenes, doncs la comparació de dues cadenes és una operació costosa (proporcional a la longitud de les cadenes comparades, en cas pitjor). Pitjor encara, els intercanvis són encara més costosos. Per això es demana el desenvolupament d'una variant de quicksort, **vqs**, per ordenar eficientment  $n$  cadenes. Per començar, el vector que rep com a entrada **vqs** no serà un vector de  $n$  cadenes, sinó un vector de  $n$  apuntadors al primer caràcter de cada cadena. El caràcter `'\0'` marca el fin d'una cadena. Aquest símbol és menor que qualssevol altre caràcter en l'ordre alfabètic. La notació  $A[i][j]$  permet accedir al  $j$ -èsim caràcter de la cadena  $i$ -èsima en el cas que la longitud de la cadena sigui inferior a  $j$ , mentre que  $A[i][j] = '\0'$  si  $j$  és la longitud de la cadena i estarà indefinit en altre cas. Com a ajut, el procediment a dissenyar, un cop feta la immersió de paràmetres, té l'especificació següent:

```
// Pre: 0 ≤ i, j < n, i + 1 < j,
// totes les cadenes en A[i..j] tenen un prefix comu
// de longitud k

void vqs (char* A[], int i, int j, int k)

// Post: A[i..j] esta ordenado creixentment segons
// l'ordre alfabetic
```

La crida inicial seria doncs **vqs**( $A$ , 0,  $n-1$ , 0). Cal que el vostre algorisme sigui molt detallat i que es dissenyin totes les funcions auxiliars que s'usin.



35. Cal organitzar l'horari d'un campionat entre  $n$  jugadors, cadascun dels quals ha de jugar exactament una vegada contra cada adversari. A més, cada jugador ha de jugar exactament un partit diari. Suposant que  $n$  és potència de 2, dissenyeu i implementeu un algorisme per construir l'horari que permeti acabar el campionat en  $n - 1$  dies. Analitzeu el cost de l'algorisme.

- ☆☆ 36. Donats un vector  $A[1..n]$  de  $n > 0$  elements i un vector  $w[1..n]$  de pesos associats (números reals positius), que satisfà  $\sum_{1 \leq i \leq n} w[i] = 1$ , la *mediana ponderada* de  $A$  és l'element  $x = A[j]$  més gran tal que

$$\sum_{A[l] < x} w[l] < \frac{1}{2} \quad \text{i} \quad \sum_{A[l] \geq x} w[l] \geq \frac{1}{2},$$

és a dir, és l'element  $x$  de  $A$  més gran tal que la suma dels pesos associats als elements menors que  $x$  és  $< 1/2$  i la suma dels pesos associats als elements més grans o iguals que  $x$  és  $\geq 1/2$ . Escriviu un algorisme eficient (al menys, el seu cost en cas mig hauria de ser  $o(n \log n)$ ) per trobar la mediana ponderada de  $A$  i analitzeu-ne el cost.

37. Tenim un vector de vectors  $V[0 .. n-1][2]$  amb informació sobre  $n$  persones. Cada posició  $V[i]$  guarda els cognoms de la  $i$ -èsima persona:  $V[i][0]$  guarda el primer cognom,  $V[i][1]$  guarda el segon cognom.

Volem ordenar el vector amb l'ordre habitual: Primer, les persones amb primer cognom més petit. En cas d'empat, van abans les persones amb segon cognom més petit. Supposeu que no hi ha dues persones amb els mateixos dos cognoms.

Per exemple, si el contingut inicial de  $V$  fos

|   | 0      | 1     | 2      | 3     |
|---|--------|-------|--------|-------|
| 0 | Garcia | Roig  | Garcia | Grau  |
| 1 | Pi     | Negre | Cases  | Negre |

el resultat final hauria de ser

|   | 0      | 1      | 2     | 3     |
|---|--------|--------|-------|-------|
| 0 | Garcia | Garcia | Grau  | Roig  |
| 1 | Cases  | Pi     | Negre | Negre |

De les combinacions següents, només una resol aquest problema en general. Digueu raonadament quina és i quin cost té en temps i en espai:

- Primer ordenem  $V$  amb *quicksort* usant el primer cognom, després amb *mergesort* usant el segon cognom.
- Primer ordenem  $V$  amb *mergesort* usant el primer cognom, després amb *quicksort* usant el segon cognom.
- Primer ordenem  $V$  amb *quicksort* usant el segon cognom, després amb *mergesort* usant el primer cognom.
- Primer ordenem  $V$  amb *mergesort* usant el segon cognom, després amb *quicksort* usant el primer cognom.

38. Recordeu que una inversió en una taula  $T[1 \dots n]$  és un parell de posicions de la taula en desordre, és a dir, un parell  $(i, j)$  tal que  $T[i] > T[j]$  amb  $1 \leq i < j \leq n$ .
- (a) Demostreu que, si en una taula hi ha una única inversió, llavors els dos elements d'aquesta apareixen consecutivament. És a dir, si  $(i, j)$  és l'única inversió de la taula, llavors  $i + 1 = j$ .
  - (b) Descriviu un algorisme de cost  $\Theta(\log n)$  en el cas pitjor que, donada una taula d'enters  $T[1 \dots n]$  que només té una inversió i un element  $x$ , digui si  $x$  és a  $T$ .
39. Es diu que una seqüència  $A = (a_1, \dots, a_n)$  de longitud  $n \geq 3$  és *unimodal* si existeix un índex  $p$ ,  $1 \leq p \leq n$ , tal que  $a_1 < a_2 < \dots < a_p$  i  $a_p > a_{p+1} > \dots > a_n$ , és a dir, creix fins a  $a_p$  i després decreix. A l'element  $a_p$  se l'anomena *pic*. Escriu un algorisme de dividir per vèncer que donat un vector que conté una seqüència unimodal trobi el pic de la seqüència, amb cost  $O(\log n)$ . No es valorarà cap solució amb cost superior a logarítmic. Justifica que el cost de l'algorisme proposat és  $O(\log n)$  i la correctesa de l'algorisme.

---

## Algorismes de programació dinàmica



1. Els nombres de Fibonacci es defineixen inductivament com segueix:

$$F(n) = \begin{cases} 0, & \text{si } n = 0 \\ 1, & \text{si } n = 1 \\ F(n-1) + F(n-2), & \text{si } n \geq 2. \end{cases}$$

Implementeu i analitzeu

- un algorisme recursiu per calcular  $F(n)$ ,
  - un algorisme que usi programació dinàmica per calcular  $F(n)$ ,
  - un algorisme recursiu per calcular alhora  $F(n)$  i  $F(n+1)$ .
2. Dissenyeu algorismes diferents per calcular  $\binom{n}{k}$ , és a dir, el nombre de maneres d'escollir  $k$  elements d'entre  $n$ . Lògicament,  $\binom{n}{k} = 0$  si  $k < 0$  o si  $k > n$ . Per definició,  $\binom{0}{0} = 1$ .

- El primer algorisme ha de ser recursiu i usar la igualtat

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

- El segon algorisme ha de fer servir la mateixa igualtat, però usant programació dinàmica.

- El tercer algorisme ha de ser recursiu i usar la igualtat

$$\binom{n}{k} = \binom{n}{k-1} \cdot \frac{n-k+1}{k}.$$

Quins dels tres algorismes són eficients?

3. Dissenyeu i analitzeu un algorisme que calculi quants nombres de  $n$  bits,  $m$  dels quals són u (per tant, amb  $n - m$  zeros), són múltiples de 3.
4. Dissenyeu i analitzeu un algorisme que calculi quants nombres de  $n$  bits,  $m$  dels quals són u (per tant, amb  $n - m$  zeros), són múltiples de 4.
5. Considereu les seqüències generades per la gramàtica següent:

$$\begin{array}{lcl} \text{seqüència} & \rightarrow & \text{nombre} \mid \text{nombre operació seqüència} \\ \text{operació} & \rightarrow & + \mid * \end{array}$$

on “*nombre*” indica qualsevol nombre real positiu. Dissenyeu un algorisme que calculi el màxim resultat possible d’aplicar les operacions d’una seqüència donada, suposant que es poden afegir tants parèntesi com es vulgui.

Per exemple, si la seqüència fos  $2 * 2.5 + 1$ , podríem avaluar-la com  $(2 * 2.5) + 1 = 6$ , o bé com  $2 * (2.5 + 1) = 7$ . Per tant l’algorisme hauria de retornar 7.

Funciona el vostre algorisme quan els nombres són negatius?

6. Dissenyeu i analitzeu un algorisme que determini el mínim nombre de transformacions que cal fer a una paraula per convertir-la en palíndroma (capicua). Les transformacions possibles són: afegir un caràcter, esborrar un caràcter, modificar un caràcter.

Per exemple, per transformar la paraula **arxbac** en un palíndrom calen com a mínim 2 transformacions; una possible manera seria: afegir una **c** al principi, obtenint **carxbac**, i després canviar la **r** per una **b**, obtenint **cabxbac**.

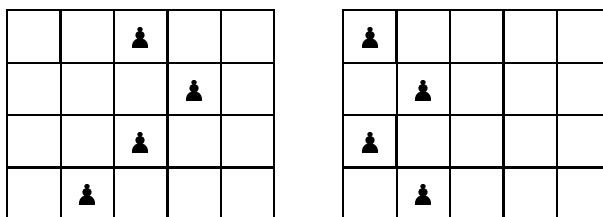
#### 7. Peons encadenats.

Considereu un taulell d’escacs amb  $n$  files i  $m$  columnes. Dissenyeu i analitzeu un algorisme que calculi el nombre de maneres diferents de posar  $n$  peons en el taulell (un per fila) de manera que

- cada peó (menys un) amenaci un altre peó,
- cada peó (menys un) sigui amenaçat per un altre peó.

Suposeu, en contra del que passa en el joc d’escacs, que els peons també es poden posar a la primera fila o a l’última.

Per exemple, aquestes són algunes maneres de posar 4 peons en un taulell  $4 \times 5$ :



8. Considereu un sistema de  $n$  segells, amb valors naturals  $0 < v_1 < v_2 < \dots < v_n$ . Dissenyeu i analitzeu un algorisme que calculi de quantes maneres diferents es pot aconseguir un cert valor positiu  $V$  usant tants segells com calgui de cada tipus.
- Per exemple, si els valors són 2, 5 i 7, llavors per a  $V = 12$  la resposta és 3, per a  $V = 7$  la resposta és 2, per a  $V = 4$  la resposta és 1, i per a  $V = 3$  la resposta és 0.
9. Torneu a resoldre el problema anterior, però suposant ara que a una carta només hi caben com a molt  $K$  segells. (Així doncs, al problema anterior teníem  $K = \infty$ .)
- Per exemple, per als valors 2, 5 i 7, amb  $V = 12$  i  $K = 3$ , la resposta ha de ser 2 (corresponent a les combinacions  $7 + 5$  i  $5 + 5 + 2$ ).
10. Torneu a resoldre el problema anterior, però suposant ara que a una carta s'han de posar exactament 2 segells.
- Per exemple, per als valors 2, 5 i 7, amb  $V = 12$ , la resposta ha de ser 1 (corresponent a la combinació  $7 + 5$ ).
11. Dissenyeu i analitzeu un algorisme que, donada una taula amb  $n \geq 1$  files i  $m \geq 1$  columnes —amb la posició  $(1, 1)$  a baix a l'esquerra— i una seqüència de posicions de la taula per les quals no es pot passar —en forma de parells  $(x, y)$ — calculi de quantes maneres diferents es pot anar de la posició  $(1, 1)$  fins a la posició  $(n, m)$  fent només passos cap a la dreta i cap amunt.
- Per exemple, donada la taula

```
X...
..X.
....
```

on 'X' denota una posició ocupada, i '.' denota una posició lliure, la resposta hauria de ser 3, corresponent als camins

```
X... X... X .
..X .X X.
.
```

12. Supposeu que sou els responsables d'una empresa dedicada a tallar barres d'acer. Donada una barra de  $M$  metres, s'obtenen  $n + 1$  barres més curtes tallant la barra original en  $n$  punts,  $0 < p_1 < p_2 < \dots < p_n < M$ . Cada tall costa un euro per cada metre que tingui la barra abans de tallar-la. Dissenyeu i analitzeu un algorisme que calculi el cost mínim de tallar una barra, donats  $M$ ,  $n$  i  $p_1, p_2, \dots, p_n$ .

Per exemple, si  $M = 9$ ,  $n = 2$ ,  $p_1 = 2$  i  $p_2 = 6$ , tallant primer a  $p_1$  el cost és  $9 + 7 = 16$ , mentre que tallant primer a  $p_2$  el cost només és  $9 + 6 = 15$ .

13. Torneu a resoldre el problema anterior, però suposant que ara podeu escollir per on tallar la barra, sempre i quan les longituds obtingudes al final no canviïn.

Per a l'exemple anterior, al final hem d'obtenir una barra de mida 2, una de mida 3 i una de mida 4. Tallant primer a la posició 5 i després a la posició 2, s'obtenen les mides desitjades i el cost total es veu reduït a  $9 + 5 = 14$ .

14. Donada una taula  $t[1 \dots n]$ , una *subseqüència ascendent* de llargada  $m$  és una llista dels elements de la taula  $t[i_1], t[i_2], \dots, t[i_m]$  amb  $1 \leq i_1 < i_2 < \dots < i_m \leq n$  tal que  $t[i_1] \leq t[i_2] \leq \dots \leq t[i_m]$ .

El problema de la Subseqüència Ascendent Màxima consisteix en, donada la taula  $t$ , determinar la llargada de la subseqüència ascendent més llarga de  $t$ .

Escriviu un algorisme de cost  $\Theta(n^2)$  per resoldre el problema de la Subseqüència Ascendent Màxima utilitzant una aproximació de dividir i vèncer.

15. *Subset sum*. Dissenyeu un algorisme que, donada una col·lecció  $C$  de  $n$  nombres enters positius (amb possibles repeticions) i un nombre enter positiu  $S$ , determini si hi ha un subconjunt de  $C$  que sumi exactament  $S$ .

Solucioneu aquest problema amb programació dinàmica, tot suposant que cada element de la col·lecció està fitat per una certa constant coneguda (com ara 1000). Quin és el cost del vostre algorisme? I si no es coneix cap fita superior per al valor dels elements?




16. *La motxilla*. Es disposa d'una motxilla que pot aguantar fins a  $C$  unitats de pes i d'una col·lecció de  $n$  objectes, cadascun dels quals té un pes enter  $p[i]$  i un valor enter  $v[i]$ . Es vol triar quins objectes cal col·locar a la motxilla, de forma que la suma dels seus valors sigui màxima però que la suma dels seus pesos no sobrepassi  $C$ . Els objectes no es poden dividir.

Solucioneu aquest problema amb programació dinàmica, tot suposant que el pes de cada objecte per una certa constant coneguda (com ara 1000) però que no hi ha cap fita als valors dels objectes. Quin és el cost del vostre algorisme?

Solucioneu de nou aquest problema amb programació dinàmica, tot suposant que el valor de cada objecte està fitat per una certa constant coneguda (com ara 1000) però que no hi ha cap fita als pesos dels objectes. Quin és el cost del vostre algorisme?




- ☆☆  17. *Camin mínims.* Implementeu l'algorisme de Floyd-Warshall per trobar les distàncies mínimes entre tots els vèrtexs d'un graf dirigit etiquetat amb pesos reals positius. Supposeu que els vèrtexs són nombres entre 1 i  $n$ , que el graf ve donat en una matriu d'adjacència, i que es coneix un valor real més gran que qualsevol etiqueta, el qual es pot usar com si fos  $\infty$ .

**Pista:** Penseu en la quantitat  $D(i, j, k)$ , definida com la distància mínima de tots els camins que van des de  $i$  fins a  $j$  sense passar per vèrtexs intermitjos més grans que  $k$ .

Quin cost en temps té el vostre algorisme? I en espai?

18. Modifiqueu l'algorisme de l'exercici anterior perquè retorni informació per poder reconstruir tots els camins mínims.
19. Dissenyau i analitzeu un algorisme que calculi la clausura transitiva d'un graf dirigit representat per una matriu d'adjacència. La clausura transitiva és una matriu de booleans que indica, per a cada parell de vèrtexos  $u$  i  $v$ , si hi ha algun camí per anar de  $u$  a  $v$ .

- ☆☆  20. Es vol construir un arbre binari de cerca amb les claus  $x_1 < x_2 < \dots < x_n$ . La probabilitat  $p(x_i)$  de consultar cada clau  $x_i$  és coneguda. (Evidentment, es té  $0 \leq p(x_i) \leq 1$  per a tota  $1 \leq i \leq n$ , així com  $\sum_{1 \leq i \leq n} p(x_i) = 1$ .) Sigui  $T$  qualsevol dels arbres binaris de cerca que es poden construir amb les  $n$  claus, i sigui  $d(T, x_i)$  la distància des de  $x_i$  fins a l'arrel de  $T$ . El cost esperat de cercar una clau en  $T$ , mesurat com el nombre de comparacions, és  $\sum_{1 \leq i \leq n} p(x_i)(1 + d(T, x_i))$ .

Dissenyau i analitzeu un algorisme que calculi el cost d'un arbre òptim (un de cost mínim).

**Pista:** Penseu com han de ser els subarbres d'un arbre òptim.

- ☆☆ 21. Tenemos un sistema constituido por  $n$  dispositivos conectados en serie, de manera que el dispositivo  $D_i$  se conecta al dispositivo  $D_{i+1}$ ,  $1 \leq i < n$ . Sea  $f_i$ ,  $1 \leq i \leq n$ , la fiabilidad del dispositivo  $D_i$ , es decir, la probabilidad de que funcione correctamente. Entonces la fiabilidad del sistema es

$$f = \prod_{1 \leq i \leq n} f_i$$

Aunque cada dispositivo sea muy fiable, la fiabilidad del sistema puede ser bastante menor (p.e. si  $n = 10$  y  $f_i = 0.99$  para  $1 \leq i \leq n$ , tenemos  $f \approx 0.904$ ).

Para aumentar la fiabilidad decidimos modificar el sistema de modo que la fase  $i$ -ésima está compuesta por  $m_i \geq 1$  réplicas del dispositivo  $D_i$  (en el sistema original  $m_i = 1$  para toda  $i$ ,  $1 \leq i \leq n$ ). La fiabilidad de la fase  $i$  viene dada entonces por  $\phi_i(m_i)$ , donde las funciones  $\phi_i$  satisfacen las siguientes propiedades : 1)  $\phi_i(1) = f_i$ ; 2) si  $m < m'$  entonces  $\phi_i(m) < \phi_i(m')$ ;  $\lim_{m \rightarrow \infty} \phi_i(m) = 1$ . Ahora la fiabilidad

del sistema es

$$f = \prod_{1 \leq i \leq n} \phi_i(m_i).$$

De la propietat (3) de las  $\phi_i$  se desprende que podemos acercar la fiabilidad de la fase  $i$  a 1 tanto como queramos (y otro tanto podemos decir de  $f$ ), colocando más y más réplicas de  $D_i$ . Pero la vida no es tan fácil ... Cada dispositivo  $D_i$  nos cuesta  $c_i$  euros y el coste total del sistema no puede superar  $E$  euros, por lo que tendremos que decidir cuidadosamente cuántos dispositivos ponemos en cada fase. Supondremos que  $E \geq c_1 + \dots + c_n$  (es decir, que podemos comprar al menos un dispositivo para cada fase!). Además, por razones diversas (stock disponible, solvencia del fabricante, etc.), puede estar limitado el número de unidades del dispositivo  $D_i$  que se pueden usar.

Diseña un algoritmo que dados dos vectores  $c[1..n]$  y  $f[1..n]$  con los costes y fiabilidades de los dispositivos, un valor real  $E > 0$ , un vector  $b[1..n]$  con el máximo de unidades disponibles de cada dispositivo y una función  $\phi_i(i, f, m)$  que calcula  $\phi_i(m)$ , halle el vector  $m[1..n]$  tal que la fiabilidad del sistema es máxima sin que su coste exceda el valor  $E$  y dentro de los límites de suministros representados por el vector  $b$ . (N.B.  $b[i] = +\infty$  si tenemos un suministro potencialmente ilimitado de unidades del dispositivo  $D_i$ ).



22. S'anomena *arbitratge* a l'ús de les discrepàncies en les taxes de canvi de divises per a obtenir un benefici. Per exemple, durant un breu espai de temps pot succeir que un euro (1 €) valgui 0.95 dòlars, un dòlar (1 \$) valgui 0.75 lliures esterlines i una lliura esterlina (1 £) valgui 1.45€. Llavors, començant amb 1 € podem acabar amb:

$$1\text{€} \times \frac{0.95\$}{1\text{€}} \times \frac{0.75\text{£}}{1\$} \times \frac{1.45\text{€}}{1\text{£}} = 1.033125\text{€},$$

amb un benefici net d'una mica més del 3.3%. Supposeu que hi ha  $n$  tipus diferents de divises i que tenim una matriu  $C[1..n, 1..n]$  on  $C[i, j]$  és la taxa de canvi de (una unitat de) la divisa  $i$  a la divisa  $j$ . Per exemple si l'euro és la divisa 1 i la lliura esterlina la divisa 2 llavors  $C[2, 1] = 1.45$ . Es compleix que  $C[i, j] = 1/C[j, i]$  per a tota  $i$  i  $j$ , i que  $C[i, i] = 1$ .

Implementa en C++ o pseudocodi un algorisme de programació dinàmica que donada la matriu  $C$  i l'identificador  $i$  d'una divisa,  $1 \leq i \leq n$ , ens proporioni el valor del millor arbitratge, on el millor arbitratge és una seqüència  $\langle i_1, i_2, \dots, i_k \rangle$  d'identificadors diferents entre sí i diferents d' $i$  tal que

$$C[i, i_1] \cdot C[i_1, i_2] \cdots C[i_{k-1}, i_k] \cdot C[i_k, i]$$

és màxim. Analitza el cost del teu algorisme.

23. Sigui  $s[1..n]$  una cadena de caràcters. Es diu que  $s$  és un *palíndrom* si es llegeix igual d'esquerra a dreta que de dreta a esquerra, és a dir, si  $s[1] \dots s[n] = s[n] \dots s[1]$ . Per exemple, **senentesisnensisetnenes** és un palíndrom.

Direm que  $s$  té *un retall en forma de palíndroms* si es pot retallar en punts diferents fent que cada subcadena sigui un palíndrom. Per exemple, per a  $s = \text{ababbbabbababa}$ , el retall  $\text{aba|b|bbabb|a|b|aba}$  té forma de palíndroms. Observeu que qualsevol cadena té, almenys, un retall en forma de palíndroms (tallant a cada caràcter).

Definim  $t[i, j]$  (amb  $1 \leq i \leq j \leq n$ ) com el nombre mínim de talls que calen per fer un retall en forma de palíndroms de  $s[i..j]$ . Noteu que si  $s[i..j]$  és un palíndrom, no cal fer cap tall.

Esbosseu un algorisme de programació dinàmica per calcular el nombre mínim de talls que calen per obtenir un retall en forma de palíndroms de  $s$ . Analitzeu el cost del vostre algorisme.



24. La distància d'edició entre dues seqüències és el menor nombre d'operacions elementals d'edició (eliminació, substitució i inserció d'elements) que permeten transformar una seqüència en l'altra. Per exemple, la distància d'edició entre les seqüències  $[b, l, a, i, r]$  i  $[l, i, a, r]$  és igual a 3, que correspon entre d'altres a qualssevol de les seqüències de transformacions següents:

- $\text{blair} \rightarrow \text{lair} \rightarrow \text{liir} \rightarrow \text{liar}$  (una eliminació i dues substitucions)
- $\text{blair} \rightarrow \text{lair} \rightarrow \text{lar} \rightarrow \text{liar}$  (dues eliminacions i una inserció)
- $\text{blair} \rightarrow \text{lair} \rightarrow \text{lir} \rightarrow \text{liar}$  (dues eliminacions i una inserció)
- $\text{blair} \rightarrow \text{llair} \rightarrow \text{liar} \rightarrow \text{liar}$  (dues substitucions i una eliminació)

Dissenyau i analitzeu un algorisme de programació dinàmica per trobar la distància d'edició entre dues seqüències  $[a_1, a_2, \dots, a_m]$  i  $[b_1, b_2, \dots, b_n]$ .

- ☆☆ 25. Es vol calcular el producte  $A_1 \cdot A_2 \cdots A_n$  d'una seqüència de  $n$  matrius. L'ordre en què es fan les multiplicacions no afecta el resultat perquè la multiplicació de matrius és associativa, però en determina l'eficiència. Per exemple, si  $A_1$  és una matriu de 10 per 30,  $A_2$  és una matriu de 30 per 5 i  $A_3$  és una matriu de 5 per 60, el càlcul de  $(A_1 \cdot A_2) \cdot A_3$  mitjançant l'algorisme convencional només requereix

$$(10 \cdot 30 \cdot 5) + (10 \cdot 5 \cdot 60) = 1500 + 3000 = 4500$$

productes escalars, mentre que el càlcul de  $A_1 \cdot (A_2 \cdot A_3)$  requereix un total de

$$(30 \cdot 5 \cdot 60) + (10 \cdot 30 \cdot 60) = 9000 + 18000 = 27000$$

productes escalars. Dissenyau i analitzeu un algorisme per trobar una parentització òptima del producte  $A_1 \cdot A_2 \cdots A_n$  de  $n$  matrius.

26. Tenim un text de longitud  $M$  sense salts de línia, una longitud màxima  $x$  de línia, i un array  $A[0..n+1]$ , ordenat creixentment, que indica a quines posicions del text es poden inserir salts de línia, amb el conveni  $A[0] = 0$  i  $A[n+1] = M$ . Si  $A[i] = j$ ,  $1 \leq i \leq n$ , llavors podem insertar un salt de línia entre els caràcters  $j$ -èssim i  $(j+1)$ -èssim del text,  $1 \leq j < M$ .

Escriviu un algorisme de programació dinàmica en pseudocodi o C++, que donats l'array  $A$  i la longitud màxima  $x$  d'una línia, determini com inserir salts de línia en posicions autoritzades, de manera que cap línia tingui més d' $x$  caràcters (sense comptar el salt de línia) i que el nombre total de línies sigui mínim. Si no existeix cap manera de dividir el text en línies amb  $\leq x$  caràcters cadascuna, s'ha de tornar com a resultat  $+\infty$ . Per exemple, si  $M = 15$ ,  $x = 10$ ,  $A[1] = 3$  i  $A[2] = 7$  (i  $A[0] = 0$ ,  $A[3] = 15$ ) llavors el nombre mínim de línies és 2, inserint un salt de línia entre el caràcters setè i vuitè del text.

- Escriuiu la recurrència per al nombre de línies mínim necessari  $L(i, j)$  per a subdividir el subtext que comença en  $A[i-1]$  i acaba en  $A[j+1]$ .
- Escriuiu l'algorisme iteratiu per a obtenir la solució a la recurrència.
- Calculeu el cost en temps i en espai del vostre algorisme.

---

## Algorismes voraços



1. Cal programar una màquina expenedora per tal que torni canvi usant el mínim nombre de monedes possibles. Dissenyeu un algorisme voraç que resolgui aquest problema quan el canvi s'ha de donar en monedes de cèntims d'euros de curs legal (monedes de 50, 20, 10, 5, 2 i 1 cèntims).

Mostreu que el vostre algorisme troba la solució òptima.

Trobeu una combinació de valors de les monedes per a la que el vostre algorisme no trobi necessàriament el nombre mínim de monedes per fer el canvi.




2. Un automobilista ha de fer un viatge seguint una ruta predeterminada. Amb el dipòsit ple pot fer  $x$  km. Sabem d'antuvi les ubicacions de totes les benzineres del trajecte, i que no n'hi ha cap tram de més de  $x$  km sense benzinera. Dissenyeu i analitzeu un algorisme voraç que permeti realitzar el trajecte fent un mínim d'aturades per posar benzina.
3. Una familia que viaja con niños va desde una ciudad  $A$  a otra ciudad  $B$  en coche, recorriendo el trayecto a velocidad constante. Escribir un algoritmo que planifique las paradas, sabiendo que los niños deben ir a la baño en intervalos de no más de  $Z$  kilómetros. El trayecto tiene  $n + 2$  puntos de parada, siendo que ningún par de puntos de parada consecutivos distan en más de  $Z$  kilómetros. El punto de parada 0 es la ciudad de origen, y el punto de parada  $n + 1$  es la ciudad de destino. Justifica la corrección de tu respuesta y analiza el coste del algoritmo.

```
// D[0] = 0
// D[i] = distancia al origen de la parada i-esima, 1 <= i <= n+1
```

```

void planifica(double Z, const vector<double>& D,
 vector<int>& sol);
// sol[j] = indice del punto de parada j-esimo

```

-  4. Donat un conjunt de reals  $X = \{x_1, x_2, \dots, x_n\}$ , es vol trobar el conjunt més petit possible d'interval·s unitaris que recobreixin tots els punts. És a dir, trobar un conjunt d'interval·s  $I = \{[i_1, i_1 + 1], [i_2, i_2 + 1], \dots, [i_m, i_m + 1]\}$  tal que  $\forall j \mid 1 \leq j \leq n \mid \exists k \mid 1 \leq k \leq m \mid x_j \in [i_k, i_k + 1]$  amb el mínim valor possible de  $m$ . Dissenyau i analitzeu un algorisme voraç per resoldre aquest problema.
-  5. Es vol emmagatzemar en un dispositiu de capacitat  $L$  un conjunt de  $n$  arxius. La talla de l'arxiu  $i$ -èsim ve donada per  $\ell[i]$ , i se sap que  $\sum_{i=1}^n \ell[i] > L$ . Dissenyau un algorisme que seleccioni un subconjunt dels arxius tot maximitzant el nombre d'arxius que es poden emmagatzemar al dispositiu. Justifiqueu la correctesa del vostre algorisme i analitzeu el seu cost.
6. Dissenyau un algorisme per calcular com emmagatzemar  $n$  arxius de talla  $\ell[i]$  en una cinta magnètica, de forma que el temps mitjà de lectura d'un fitxer sigui mínim. Se suposa que tots els fitxers es llegeixen amb la mateixa freqüència, i que la cinta es rebobina després de llegir cada fitxer. Així doncs, el temps mitjà de lectura serà  $\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^i \ell[p[j]]$  on  $p$  és la permutació escollida.
-  7. S'han de planificar  $n$  tasques en un processador que triga un temps  $\ell[i]$  en completar la  $i$ -èsima tasca. A més, cada tasca ha d'acabar, com a molt tard, a l'instant  $t[i]$ . Una seqüència factible és una ordenació de les tasques tal que la seva execució en aquell ordre fa que cada tasca  $i$  acabi com a molt tard a l'instant  $t[i]$ . Dissenyau un algorisme voraç per calcular una seqüència factible o indicar que no n'hi ha cap.
8. S'han de planificar  $n$  tasques en un processador que triga una unitat de temps en completar cadascuna. Per a cada tasca  $i$  hi ha un instant d'acabament  $t[i]$  tal que si completem la tasca abans d'aquell instant rebem un benefici  $b[i] \geq 0$ . Dissenyau un algorisme voraç que calculi una permutació  $p$  que representi l'ordre en el qual executar les tasques tot maximitzant el benefici net  $\sum_{p[i] \leq t[i]} b[i]$ .
9. Donat un graf no dirigit  $G = (V, E)$ , es diu que un subconjunt dels vèrtexs  $U \subseteq V$  és un recobriment de  $G$  si cada aresta en  $E$  incideix en almenys un vèrtex de  $U$ . Un recobriment  $U$  és mínim si no hi ha cap altre recobriment amb menys vèrtexs.
- El Professor Gran proposa l'algorisme voraç següent: Començem amb  $E' := E$ ,  $U := \emptyset$  i  $S := V$ . Mentre  $E' \neq \emptyset$ , agafem un vèrtex  $u$  de  $S$ ; si el nombre de veïns de  $u$  en  $E'$  és zero, esborrem  $u$  de  $S$ ; sinó incloem  $u$  a  $U$ , l'esborrem de  $S$  i esborrem de  $E'$  totes les seves arestes adjacents.
- (a) Mostreu que aquest algorisme proporciona un recobriment però que no és necessàriament òptim.

- (b) El Professor Alzina proposa modificar l'algorisme anterior triant sempre el vèrtex de  $S$  que tingui més veïns en  $E'$ . Mostreu que aquest algorisme tampoc és correcte.
- (c) Considerem ara el cas restringit a arbres. Proposeu un algorisme que resolgui el problema del recobriment mínim restringit a arbres en temps lineal.
10. Tenim informació emmagatzemada en  $n$  cintes, on la  $i$ -èsima cinta conté  $m[i]$  registres ordenats en forma creixent. Es vol obtenir una única cinta amb tota la informació ordenada, efectuant  $n - 1$  operacions de fusió entre dues cintes (una fusió llegeix dues cintes i en grava una de nova amb la reunió ordenada de la informació continguda a les dues cintes d'entrada). L'ordre en el qual es facin les fusions afecta l'eficiència del procés. Dissenyeu un algorisme voraç que minimitzi el nombre de còpies de registres.

Exemple: Si  $n = 3$ , i  $m_A = 30$ ,  $m_B = 20$  i  $m_C = 10$ , tenim:

- Opció 1:  $((A, B), C)$      $X = (A, B)$  requereix 50 còpies  
 $Y = (X, C)$  requereix 60 còpies *Total: 110 còpies*
- Opció 2:  $((C, B), A)$      $X = (C, B)$  requereix 30 còpies  
 $Y = (X, A)$  requereix 60 còpies *Total: 90 còpies*
- Opció 3:  $((A, C), B)$      $X = (A, C)$  requereix 40 còpies  
 $Y = (X, B)$  requereix 60 còpies  
*Total: 100 còpies*

- ☆☆ 11. Un melòman disposa d'una llarga col·lecció de peces gravades en *cassettes*, però està fart d'haver-se d'esperar per sentir la peça que li ve de gust. Dissenyeu i analitzeu un algorisme voraç que donades  $n$  cançons amb les seves respectives durades  $d[i]$  i les freqüències  $f[i]$  amb que són triades, trobi una permutació  $p$  de les  $n$  cançons en una cinta de tal forma que es minimitzi el temps d'accés mitjà. Es vol doncs que  $\sum_{i=1}^n f[p[i]] \sum_{j=1}^{i-1} d[p[j]]$  sigui mínim.



12. Donat un tauler  $n \times n$  amb  $n$  fitxes a unes certes posicions  $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ , i una fila  $i$  (amb  $0 \leq i < n$ ), cal calcular el nombre mínim de moviments per posar les  $n$  fitxes a la fila  $i$ , una a cada columna. Els moviments permesos són cap a la dreta, esquerra, amunt i avall. Durant aquests moviments es poden empilar tantes fitxes a la mateixa posició com calgui.

Aquest és un exemple amb  $n = 5$ ,  $i = 1$ , i les fitxes a les posicions  $(0, 4)$ ,  $(1, 2)$ ,  $(2, 2)$ ,  $(4, 1)$  i  $(4, 4)$ . En aquest cas la resposta és 11. Fixeu-vos que, en general, hi ha moltes maneres diferents d'obtenir el mínim nombre de moviments (aquí només se'n mostren dues).





15. Quin sentit té calcular camins mínims quan hi ha hi ha cicles de cost negatiu en un graf?

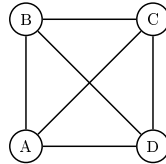
Mostreu un graf per al qual l'algorisme de Dijkstra no funciona quan hi ha pesos negatius, malgrat no haver-hi cicles de cost negatiu.

16. Modifiqueu l'algorisme de Dijkstra per tal que compti el nombre de camins mínims entre dos vèrtexs donats. Què passaria si el graf tingués cicles de cost zero? I si tingués arcs de cost zero però no cicles de cost zero?

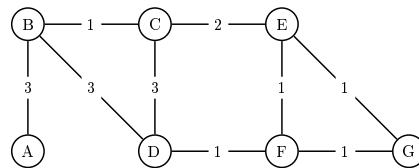


17. Modifiqueu l'algorisme de Dijkstra per tal que si hi ha més d'un camí mínim entre dos vèrtexs donats, en retorni un de qualsevol amb el nombre mínim d'arcs.

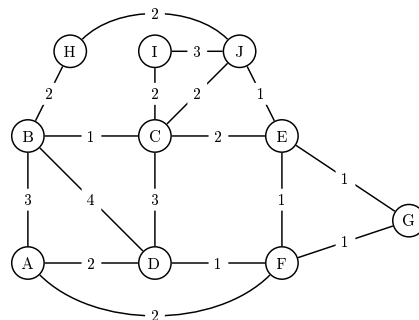
18. Doneu tots els possibles arbres d'expansió del graf següent:



19. Doneu tots els possibles arbres d'expansió mínims del graf següent:



20. Mostreu quin arbre retorna l'algorisme de Prim sobre el graf següent. En cas d'haver-hi més d'una possibilitat, trenqueu-la segons l'ordre alfabètic.

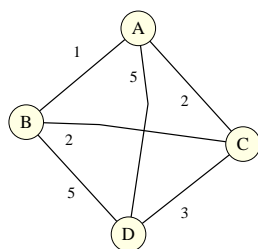


21. Digueu si l'algorisme de Prim funciona amb grafs amb pesos negatius.

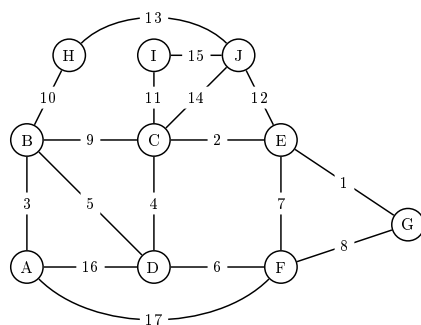
22. Una empresa disposa d'un conjunt d'ordinadors distribuïts dins un àmbit geogràfic. Diverses parelles d'aquests ordinadors estan físicament connectats mitjançant una línia qualsevulla de comunicació (bi-direccional). Cada connexió té un cost associat, en funció d'alguns paràmetres (longitud de la línia, qualitat del senyal, ...). Dos ordinadors estan comunicats si existeix una seqüència de línies de comunicació que uneix ambdós ordinadors. L'empresa vol calcular quines línies de comunicació ha d'usar per que es compleixi que:
- (a) dos ordinadors distingits estiguin comunicats entre sí amb el mínim cost possible.
  - (b) tots els ordinadors estiguin comunicats, de manera que el cost total de la xarxa sigui mínim.
23. La región de Aicenev, al sur del Balquistán, es famosa en el mundo entero por su sistema de comunicaciones basado en canales (grandes y pequeños) que se ramifican. Todos los canales son navegables y todas las localidades de la región están situadas junto a algún canal. Una vez al año los alcaldes de la región se reúnen en la localidad de San Marcos y cada uno de ellos se desplaza en su vaporetto oficial. Proponed a los alcaldes un algoritmo que les permita encontrar la manera de llegar a San Marcos de manera que entre todos hayan recorrido el mínimo de distancia. A consecuencia de la crisis energética, los alcaldes han decidido compartir los vaporettos. Cada uno va en el suyo hasta alguna otra ciudad, allí se une a otro u otros, van juntos hasta algún otro sitio, formando así grupos que se dirigen a San Marcos. Discutid la validez de la solución anterior. Proponed otra, si es necesario, para minimizar la distancia recorrida por los vaporettos (no necesariamente por los alcaldes).
24. Sigui el graf no dirigit i etiquetat  $G = (V, E)$  amb  $|V| = n$  i  $|E| = e$ . Quin algorisme aplicaries per trobar un arbre mínim d'expansió en el tres casos següents i per què:
- (a)  $e < n - 1$
  - (b)  $2n > e > n - 1$
  - (c) hi ha una aresta entre tot parell de nodes

Si en algun apartat  $G$  ha de complir alguna condició addicional, esmenta-la.

25. Donat el graf de la figura, aplica-li els algorismes de Prim i Kruskal. Ves mostrant como creix l'arbre d'expansió pas a pas. En l'algorisme de Prim, comença pel node A.

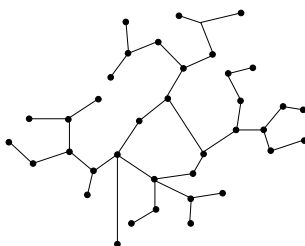


26. Digueu quin és l'arbre d'expansió mínim que retorna l'algorisme de Prim sobre el graf següent, tot suposant que comença pel vèrtex A.

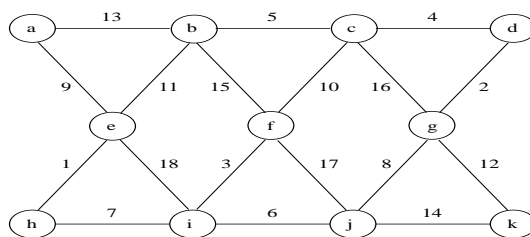


27. Sigui  $G$  un graf no dirigit connex amb  $n$  vèrtexs,  $m$  arestes i un únic cicle que té llargada  $\ell$ . Digueu (breument però de forma justificada) quants arbres d'expansió diferents té aquest graf.

Anomenem *graf unicíclic* a un graf que conté exactament un cicle. Anomenem *graf d'expansió unicíclic* d'un graf  $G = (V, E)$  a un graf connex unicíclic  $H = (V, E')$ , amb  $E' \subseteq E$ . Per exemple, el graf següent és un graf connex unicíclic.



28. Trobeu un graf d'expansió unicíclic de cost mínim (és a dir, amb la mínima suma de pesos a les arestes) del graf següent.



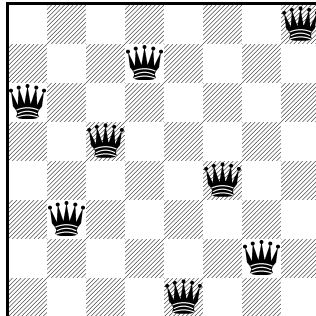
29. Descriu i justifiqueu un algorisme (com més eficient millor) per trobar un graf d'expansió unicíclic de cost mínim d'un graf no dirigit amb pesos positius a les arestes. Analitzeu el cost del vostre algorisme en el cas pitjor.
30. Considereu un fitxer que només conté els caràcters A, B, C, D i E. El caràcter A apareix 35000 cops, el caràcter B apareix 1000 cops, els caràcters C i D apareixen 2000 cops, i el caràcter E apareix 1500 cops. Dibuixeu un arbre de Huffman que minimitzi la llargada d'aquest fitxer. Quina serà la llargada del fitxer un cop condificat?
31. El problema de la *motxilla fraccionària* es defineix de la manera següent: donats  $n$  objectes amb valors unitaris  $v_1, \dots, v_n$  i pesos  $p_1, \dots, p_n$  i una capacitat  $0 < C < \sum_{1 \leq i \leq n} p_i$ , l'objectiu és determinar quina quantitat (una fracció entre 0 i 1) de cadascun dels  $n$  objectes hauriem de posar a una motxilla amb capacitat màxima  $C$  tot maximitzant el valor que hi posem. Formalment, es tracta de determinar els valors  $x_1, \dots, x_n$ , amb  $x_i \in [0, 1]$  tals que  $\sum_{1 \leq i \leq n} x_i p_i < C$  (el pes acumulat no excedeix la capacitat de la motxilla) i que maximitzen  $\sum_{1 \leq i \leq n} x_i v_i$  (valor acumulat a la motxilla).
- Dissenyu un algorisme voraç per a resoldre aquest problema. Demostreu la seva correctesa i analitzeu el seu cost en funció d' $n$ .



---

## Algorismes de cerca exhaustiva

1. *Les  $n$  reines.* Dissenyeu un algorisme que escrigui totes les possibles maneres de col·locar  $n$  reines en un tauler amb  $n \times n$  escacs de forma que cap reina n'amenaci cap altra.


Per exemple, aquesta és una configuració de 8 reines:




-  2. Dissenyeu un algorisme que compti de quantes maneres es poden col·locar  $n$  reines en un tauler amb  $n \times n$  escacs de forma que cap amenaci cap altra.
-  3. Dissenyeu un algorisme que trobi una possible manera de col·locar  $n$  reines en un tauler amb  $n \times n$  escacs de forma que cap amenaci cap altra o indiqui que no hi ha tal col·locació.

4. *Salts de cavall.* En un tauler amb  $n \times n$  escacs es col·loca un cavall en una casella donada. Dissenyeu un algorisme per trobar si existeix alguna forma d'aplicar  $n^2 - 1$  moviments de cavall de forma que el cavall visiti totes les caselles del tauler.


Per exemple, aquesta és una possible solució per a un tauler  $5 \times 5$  començant des del centre:

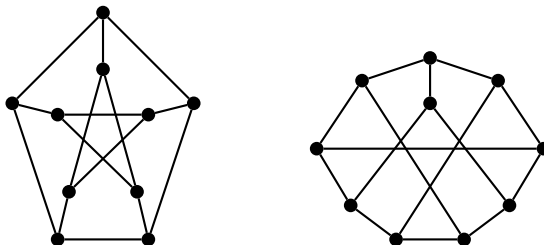
|    |    |                                                                                   |    |    |
|----|----|-----------------------------------------------------------------------------------|----|----|
| 22 | 5  | 16                                                                                | 11 | 24 |
| 15 | 10 | 23                                                                                | 6  | 1  |
| 4  | 21 |  | 17 | 12 |
| 9  | 14 | 19                                                                                | 2  | 7  |
| 20 | 3  | 8                                                                                 | 13 | 18 |

-  5. *El quadrat llatí.* Un quadrat llatí d'ordre  $n$  és una taula  $n \times n$  on cada casella està pintada amb un color escollit d'entre  $n$  possibles sense que cap fila ni cap columna contingui colors repetits. Per exemple, aquest és un quadrat llatí d'ordre 5:

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 2 | 3 | 1 | 4 |
| 1 | 3 | 4 | 2 | 0 |
| 2 | 1 | 0 | 4 | 3 |
| 3 | 4 | 1 | 0 | 2 |
| 4 | 0 | 2 | 3 | 1 |

Dissenyeu un algorisme que escrigui tots els quadrats llatins d'ordre  $n$ .

-  6. *Hamiltonià?* Dissenyeu i analitzeu un algorisme per determinar si un graf connex no dirigit és hamiltonià. Modifiqueu-lo per obtenir-ne un cicle hamiltonià si aquest existeix.
7. *Isomorfisme de grafs.* Dos grafs  $G_1$  i  $G_2$  es diuen *isomorfs* si existeix una bijectió  $f$  entre els vèrtexs d'ambós grafs de forma que  $\{u, v\} \in E(G_1)$  si i només si  $\{f(u), f(v)\} \in E(G_2)$ . Per exemple, els dos grafs següents son isomorfs.



Dissenyeu i analitzeu un algorisme per determinar si dos grafs (donats a través de matrius d'adjacència) són isomorfs.

8. *Clique*. Donat un graf no dirigit  $G = (V, E)$ , una *clique* és un subconjunt de vèrtexs  $S \subseteq V$  tal que per a qualsevol parell de vèrtexs  $u$  i  $v$  en  $S$  hi ha una aresta entre  $u$  i  $v$  en el graf.

Dissenyeu i analitzeu un algorisme que donat un graf  $G$  i un natural  $k$ , determini si  $G$  conté alguna clique de talla  $k$ .



9. *La motxilla*. Es disposa d'una motxilla que pot aguantar fins a  $C$  unitats de pes i d'una col·lecció de  $n$  objectes, cadascun dels quals té un pes  $p[i]$  i un valor  $v[i]$ . Es vol triar quins objectes cal col·locar a la motxilla, de forma que la suma dels seus valors sigui màxima però que la suma dels seus pesos no sobrepassi  $C$ . Els objectes no es poden dividir.

Dissenyeu i analitzeu un algorisme per al problema de la motxilla. Feu-ho amb l'esquema de tornada enrera (*backtracking*) i amb l'esquema de ramificació i poda (*branch and bound*).

10. *Assignació de tasques*. Un cap de personal disposa de  $n$  treballadors per realitzar  $n$  tasques. El temps que triga el treballador  $i$  per realitzar la tasca  $j$  ve donat per  $T[i][j]$ . Dissenyeu un algorisme que assigni una tasca a cada treballador de forma que es minimitzi la suma total de temps.

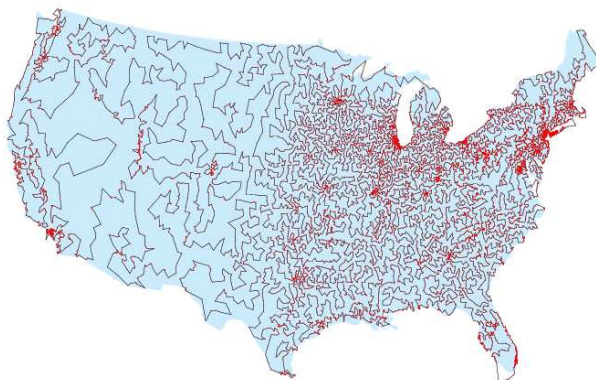
11. El nostre cap de personal disposa ara de  $p$  treballadors per efectuar  $n$  tasques ( $p < n$ ) amb una relació de precedència (un ordre parcial) entre elles. La tasca  $i$  requereix un temps  $T[i]$ , independentment del treballador que la realitzi. La relació de precedència ve donada a través d'una taula **prec** $[i][j]$  que indica si la tasca  $i$  s'ha d'acabar abans de començar la tasca  $j$ .

Dissenyeu un algorisme que trobi una assignació de tasques a persones (quina tasca ha de fer cada persona i quan l'ha de començar) que respecti les precedències i minimitzi el temps necessari.

Advertiment: els treballadors no tenen perquè estar permanentment ocupats.



12. *El viatjant*. Un viatjant ha de visitar  $n$  clients que viuen en  $n$  ciutats diferents. La distància per anar de la ciutat  $i$  a la ciutat  $j$  ve donada per  $D[i][j]$ . El viatjant vol, sortint de la seva ciutat, passar exactament un cop per cada ciutat on viuen els seus clients i retornar al punt de partida. A més, vol que la distància total recorreguda sigui mínima. Per exemple, aquest és el pla de viatge d'un viatjant de comerç pels EEUU:



Dissenyu i analitzeu un algorisme per resoldre el problema del viatjant. Feu-ho amb l'esquema de tornada enrera (*backtracking*) i amb l'esquema de ramificació i poda (*branch and bound*).

13. *Conjunt independent.* Dissenyu i analitzeu un algorisme que, donat un graf  $G = (V, E)$  i un nombre natural  $k$ , determini si  $G$  té un conjunt independent de  $k$  vèrtexs (és a dir, si existeix un subconjunt  $A \subseteq V$  de  $k$  vèrtexs tal que  $\{u, v\} \notin E$  per a tot  $u, v \in A$ ).
14. *Recobriments.* Dissenyu i analitzeu un algorisme que, donat un graf  $G = (V, E)$  i un nombre natural  $k$ , determini si  $G$  té un recobriments de les arestes amb  $k$  vèrtexs. Recordeu que això vol dir determinar si existeix un subconjunt  $A \subseteq V$  de  $k$  vèrtexs tal que per a cada  $\{u, v\} \in E$ , al menys un dels extrems  $u$  o  $v$  pertany a  $A$ .
15. *L'agència matrimonial.* Una agència matrimonial treballa per a  $n$  clients i  $n$  clientes. Cada client i clienta ha omplert un formulari rosa a través del qual l'agència pretén aparellar-los de la millor forma possible. Concretament, per a cada client  $i$  i cada clienta  $j$  es disposa d'un valor positiu  $M[i][j]$  que correspon a la preferència del client  $i$  per la clienta  $j$  i d'un valor positiu  $F[i][j]$  que correspon a la preferència de la clienta  $j$  pel client  $i$ . La preferència mutua entre el client  $i$  i la clienta  $j$  ve definit com  $M[i][j] * F[i][j]$ . L'agència desitja aparellar els  $n$  clients amb les  $n$  clientes de forma que la suma de les preferències mútues sigui màxima.

Dissenyu i analitzeu un algorisme per resoldre el problema de l'agència matrimonial.

16. *Subset sum.* Dissenyu un algorisme que, donada un col·lecció  $C$  de  $n$  nombres enters positius (amb possibles repeticions) i un nombre enter positiu  $S$ , determini si hi ha un subconjunt de  $C$  que sumi exactament  $S$ .

Solucioneu aquest problema amb l'esquema de tornada enrera (*backtracking*) i amb l'esquema de ramificació i poda (*branch and bound*).

17. *Coloració.* Es disposa de  $c \geq 3$  colors diferents i d'un mapa de  $n$  països, juntament amb una taula de booleans  $v[i][j]$  que indica si dos països  $i$  i  $j$  són veïns. Escriuiu



un algorisme de cerca exhaustiva que trobi totes les coloracions possibles del mapa amb la restricció que dos països veïns han de ser pintats amb colors diferents.

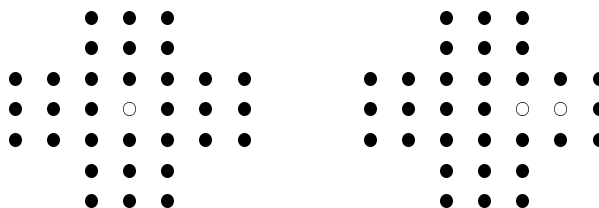
Com solucionareu el problema per a  $c = 2$  (bicoloració)?

- ☆☆ 18. *Inequacions lineals 0-1.* Dissenyeu un algorisme que, donada una matriu  $A$  de  $m \times n$  enters i un vector  $b$  de  $m$  enters, determini si existeix un vector  $x \in \{0, 1\}^n$  tal que  $Ax \geq b$ .

19. *Empaquetament.* Tenim una col·lecció de  $n$  objectes que cal empaquetar en envasos de capacitat  $C$ . L'objecte  $i$  té volum  $v[i]$ . Dissenyeu un algorisme que calculi quin és l'empaquetament òptim, és a dir, aquell que minimitza el nombre d'envasos. Els objectes no es poden fraccionar però, per simplificar el problema, podeu considerar que un objecte cap dins d'un envàs si el volum de l'objecte és igual o inferior a l'espai lliure que queda a l'envàs (independentment de la forma que l'objecte pugui tenir.)

Solucioneu aquest problema amb l'esquema de tornada enrera (*backtracking*) i amb l'esquema de ramificació i poda (*branch and bound*).

- ☆☆ 20. *El solitari.* L'àvia Victòria sovint jugava a un solitari que mai va arribar a resoldre (vegeu figura esquerra).



Aquest solitari es juga en un tauler de 33 caselles on, per començar, a cada forat es col·loca un fitxa (●), excepte en el forat central, que queda buit (○). Les fitxes poden saltar sobre les seves veïnes (immediates) en direcció horitzontal o vertical si la casella de destinació és buida. En aquest cas, la fitxa sobre la qual s'ha saltat es retira. Per exemple, si des de la situació inicial, mostrada a la figura del centre, la fitxa de la quarta fila i sisena columna saltés cap a l'esquerra, s'arribaria a la configuració mostrada a la figura de la dreta. L'objectiu és trobar una seqüència de salts que retiri totes les fitxes del tauler excepte una, que ha de quedar al centre.

Dissenyeu un algorisme que resolgui aquest solitari.

21. Probablement ja heu perdut moltes hores jugant al Sudoku... Les regles són senzilles: Hom reb una graella  $9 \times 9$  amb alguns números i cal que ompli les caselles buides de forma que cada fila, cada columna i cada caixa  $3 \times 3$  contingui tots els números del 1 al 9. Per exemple, el sudoku de l'esquerra té com a única solució la graella de la dreta:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   | 2 | 3 |   | 7 |
|   |   |   |   |   | 6 | 4 | 5 |   |
| 1 |   |   | 9 | 3 |   |   |   |   |
|   |   |   |   | 6 | 1 | 8 |   |   |
|   | 4 | 8 |   |   |   | 5 | 6 |   |
|   |   | 6 | 4 | 2 |   |   |   |   |
|   |   |   |   | 7 | 5 |   |   | 8 |
|   | 2 | 9 | 1 |   |   |   |   |   |
| 4 |   | 5 | 6 |   |   |   |   |   |

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 9 | 6 | 4 | 8 | 5 | 2 | 3 | 1 | 7 |
| 3 | 8 | 2 | 7 | 1 | 6 | 4 | 5 | 9 |
| 1 | 5 | 7 | 9 | 3 | 4 | 2 | 8 | 6 |
| 7 | 9 | 3 | 5 | 6 | 1 | 8 | 2 | 4 |
| 2 | 4 | 8 | 3 | 9 | 7 | 5 | 6 | 1 |
| 5 | 1 | 6 | 4 | 2 | 8 | 7 | 9 | 3 |
| 6 | 3 | 1 | 2 | 7 | 5 | 9 | 4 | 8 |
| 9 | 2 | 9 | 1 | 4 | 3 | 6 | 7 | 5 |
| 4 | 7 | 5 | 6 | 8 | 9 | 1 | 3 | 2 |

Escriviu un programa que resolgui sudokus. L'entrada és una matriu de  $9 \times 9$  números entre 0 i 9. Els zeros representen caselles buides.

22. Els mossos d'esquadra han establert una xarxa de vigilància d'una organització criminal. Per aquest motiu, analitzen durant un llarg període de temps les trucades que fan els membres del grup. Així, obtenen un graf dirigit anomenat *graf de relacions jeràrquiques*. L'inspector en cap, que té molta experiència, sap que quan un subgrup prou gran dels membres de l'organització han establert comunicació entre tots ells, llavors l'organització és prou feble per intentar introduir-hi un infiltrat.

Com a membres de la policia científica, l'inspector us ha donat el graf de relacions jeràrquiques i us ha encarregat que escriviu un programa que esbrini si la grandària d'aquest subgrup és superior o igual a  $k$  per tal de decidir el moment oportú per infiltrar-se a l'organització.

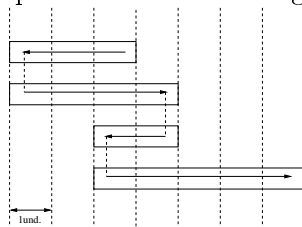
23. S'han de distribuir  $n$  convidats en una taula on hi ha lloc per  $n$  comensals. Es disposa d'una funció  $\text{adj}(a, b)$  que retorna cert si els llocs  $a$  i  $b$  són adjacents, i fals si no ho són. També hi ha una funció  $\text{af}(k, l)$  que retorna un valor comprès entre 0 i 5, segons el grau d'afinitat que els convidats  $k$  i  $l$  tinguin entre si (0 indica aversió profunda i 5 simpatia desbordant). Les funcions  $\text{adj}(i, j)$  i  $\text{af}(k, l)$  són simètriques. Dissenyau un algorisme de tornada enrera que calculi la distribució de convidats a la taula que optimitza el seu benestar. Aquest benestar es calcularà sumant els afectes dels comensals asseguts en posicions adjacents.

- ☆☆ 24. Disposem d'un tauler rectangular que medeix  $A \times B$ , amb  $1 < A < B$ , i de  $k$  peces idèntiques, també rectangulars, que medeixen  $a \times b$ , amb  $1 < a < b$ . Es garanteix que l'àrea del tauler coincideix amb la de totes les peces, o sigui,  $k \times a \times b = A \times B$ . Dissenyar un algorisme que trobi totes les maneres, si es que n'hi ha alguna, de cobrir el tauler amb les  $k$  peces de manera que no deixin forats ni se solapin, ni quedin, parcialment o totalment, fora del tauler. Féu servir un TAD per les operacions d'inicialitzar el tauler, col·locar una peça, treure-la o comprovar si n'hi cap. Escollir una representació pel TAD, explicar que fa cada operació i com actua sobre la representació escollida. Suggestir mètodes per millorar l'eficiència, evitant

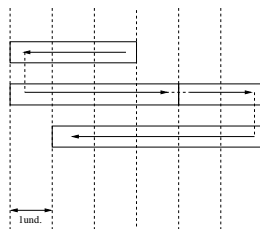
de construir configuracions parcials que no puguin conduir a una solució, i per eliminar solucions redundants.

25. Cierta número  $N$  de factorías de tecnología punta, recién creadas en la República Popular Socialista de Fanfanisflan, requieren una instalación informática y personal que la dirija. El Gobierno Supremo de la Nación les ha asignado  $N$  máquinas y  $N$  técnicos recién licenciados. Pero no todas las máquinas son apropiadas para las necesidades de todas las empresas. Se dispone de una función booleana apropiada? $(m, e)$  que indica si la máquina  $m$  es apropiada para la empresa  $e$ . Asimismo, no todos los técnicos conocen todas las máquinas, por lo que disponemos de una función booleana conoce? $(t, m)$  que indica si el técnico  $t$  conoce la máquina  $m$ . Finalmente, no todos los técnicos aceptan trabajar en todas las empresas, debido a que algunas de ellas se relacionan con la industria bélica; disponemos de la previsible función booleana aceptaria? $(t, e)$  que indica si el técnico  $t$  aceptaría trabajar en la empresa  $e$ . Se pide un programa capaz de encontrar una manera de asignar a cada empresa una máquina apropiada a sus necesidades, y un técnico que conozca la máquina y que acepte trabajar en la empresa, si es que tal asignación existe; y lo indique adecuadamente si tal asignación no existe.
26. Un concurs consisteix en el següent: hi ha dos jocs de boles numerades des de 1 fins a  $N$ , un de boles negres i un de blanques; en total  $2N$  boles. D'aquestes boles se'n col·loquen  $N$ , una per cada número, dins una caps. Cada concursant fa una aposta sobre el color de quatre de les boles de la caps, indicant número i color (e.g., la 1 negra, la 7 blanca, la 11 blanca i la 12 negra). En total hi ha  $M$  concursants. L'organitzador del joc, que és un trampós, vol calcular, un cop conegudes les  $M$  apostes, una combinació de les boles de la caps tal que ningú no encerti les seves quatre prediccions. Dissenyar un algorisme que la calculi o bé que l'indiqui que no existeix.
27. Disponemos de una regla de carpintero un tanto inusual. Está compuesta por una serie de segmentos consecutivos y articulados y cada uno de ellos tiene una longitud asociada que no tiene porqué ser siempre la misma. Se trata de diseñar un algoritmo eficiente que indique cómo hay que plegar la regla para que se minimice la longitud que ocupa una vez plegada. Justificad el esquema elegido.

Ejemplo clarificador: Supongamos que la regla se compone de 4 segmentos de longitudes 3,4,2,5 en esta secuencia, que es inalterable. Una forma de plegarlo bastante intuitiva sería la que se muestra en la figura que viene a continuación:



Como puede apreciarse, la longitud que ocupa plegada es 7 y la solución sería  $\langle \text{izq}, \text{der}, \text{izq}, \text{der} \rangle$ .



Sin embargo, plegándola como se indica en la siguiente figura se consigue una longitud de 6, que es la mínima, y que tiene como solución  $\langle \text{izq}, \text{der}, \text{der}, \text{izq} \rangle$ .

28. Dissenyen un algorisme de tornada enrere per, donat  $n \geq 1$  i donats  $N = n(n-1)/2$  valors enters positius  $D_1, D_2, \dots, D_N$ , trobi els  $n$  valors  $x_1, x_2, \dots, x_n$  de manera que cada  $D_i$  sigui la distància entre dos  $x_j$ 's o bé ens digui que no existeixen aquests  $n$  valors. És a dir, s'han de trobar  $x_1, x_2, \dots, x_n$  tals que per a tota  $i$ ,  $1 \leq i \leq N$ , existeixen  $j$  i  $k$ ,  $1 \leq j, k \leq n$  tals que  $D_i = |x_j - x_k|$ , o bé s'ha de dir que no existeix un conjunt de  $n$  valors  $x$  que satisfacin aquesta propietat.

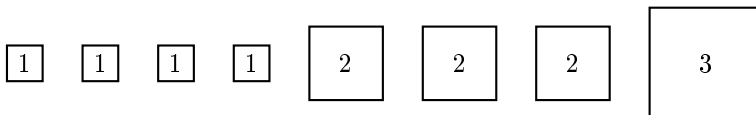
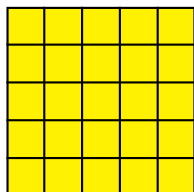
Es pot suposar sense pèrdua de generalitat que  $x_1 = 0 \leq x_2 \leq x_3 \leq \dots \leq x_n$ . Llavors és obvi que, si existeix una solució, per tota  $j$ ,  $1 < j \leq n$ ,  $x_j = D_i$  per alguna  $i$ ,  $1 \leq i \leq N$  i  $x_n = D_{\max} = \max\{D_i \mid 1 \leq i \leq N\}$ . Per exemple, amb  $n = 4$  i  $D = \langle 2, 4, 5, 6, 7, 11 \rangle$  una solució possible és  $x_1 = 0$ ,  $x_2 = 5$ ,  $x_3 = 7$  i  $x_4 = 11$ .

Implementeu la vostra solució en C++ i expliqueu detalladament la vostra solució justificant la seva correctesa.

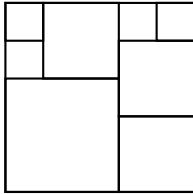
```
void distancies(int n, const vector<int>& D,
 vector<int>& x, bool& hi_ha_sol);

// hi_ha_sol == true i per a tota i, 0 ≤ i < n,
// existeixen j i k, 0 ≤ j, k < n tals que
// x[j] ≥ x[k] i D[i] = x[j] - x[k]; o
// bé hi_ha_sol == false
```

29. Considereu un puzzle que consisteix a col·locar  $k$  fitxes quadrades, cadascuna de talla  $a_i \times a_i$ , sobre un tauler quadrat de mida  $n \times n$  sense deixar forats ni solapar fitxes. L'àrea del tauler coincideix amb la de totes les fitxes, és a dir  $\sum_{i=0}^{k-1} a_i^2 = n^2$ . Per exemple, per a aquest tauler  $5 \times 5$  i aquestes fitxes



aquesta seria una possible solució:



Dissenyeu i implementeu en C++ un algorisme de tornada enrera (*backtracking*) per saber si, donada la mida del taulell i les mides de les fitxes, aquest puzzle té alguna solució o no. Per fer-ho, completeu la classe `Puzzle` i utilitzeu la classe `Tauler` que es donen a continuació.

Comentaris: Heu de seguir l'especificació de les dues classes. Encara que no cal, podeu afegir camps i/o mètodes privats a la classe `Puzzle`, però no podeu alterar la seva part pública. Totes les operacions de la classe `Tauler` prenen temps constant. No heu d'implementar la classe `Tauler`. L'eficiència del vostre algorisme no importa massa aquí, és més important que demostreu que sabeu escriure un algorisme de tornada enrera.

```
class Puzzle {

 int _n; // Nombre de files i de columnes del taulell
 int _k; // Nombre de fitxes
 vector<int> _A; // Taula amb la mida de cada fitxa
 bool _trob; // Indica si s'ha trobat una solució
 Tauler _T; // Tauler

 // Completeu aquesta funció i doneu la seva crida inicial
 void recursiu (...) {
 ...
 }

public:

 // Soluciona el puzzle per a un taulell n x n amb fitxes
 // de mida A.
 Puzzle (int n, vector<int> A) : _n(n), _A(A), _k(A.size())
 _trob(false), _T(Tauler(n)) {

 recursiu(...);
 }

 // Indica si el puzzle té solució.
 bool té_solució () {
 return _trob;
 }
};
```

```

 }
};

class Tauler {

public:

 // Construeix un tauler buit de mida n x n.
 Tauler (int n);

 // Indica si es pot col·locar una fitxa des de la posició (x1,y1)
 // a la posició x2,y2
 // (x1,y1) són les coordenades on s'intenta col·locar una
 // cantonada de la fitxa, (x2, y2) són les coordenades de la
 // cantonada oposada.
 bool es_pot (int x1, int y1, int x2, int y2);

 // Col·loca una fitxa des de la posició (x1,y1) a la posició
 // (x2, y2).
 // Precondició: es_pot(x1,y1,x2,y2)
 bool col·loca (int x1, int y1, int x2, int y2);

 // Retira la fitxa que es troba sobre la posició (x,y)
 // (el TAD ja recorda la seva mida).
 // Si no s'havia col·locat cap fitxa prèviament en aquesta
 // posició, no fa res.
 void retira (int x, int y);

 // Indica si la posició (x,y) està ocupada per alguna fitxa.
 bool ocupada (int x, int y);

};

```

30. La companyia informàtica per la qual trebal·leu vol vendre els seus serveis a cadenes de radio que es volen estendre a noves regions i que volen posar les seves antenes de difusió a ciutats ben escollides. L'únic problema és que les antenes són molt cares, per tant, les cadenes de ràdio volen cobrir tots els nous territoris posant el mínim nombre d'antenes possible. Suposant que una antena posada a una ciutat, cobreix aquesta ciutat i les seves ciutats veïnes, el vostre cap us demana d'escriure un algorisme de backtracking que, donada la matriu booleana i simètrica  $M$  de dimensions  $n \times n$  indicant per a cada parell  $(i, j)$  si les ciutats  $i$  i  $j$  són veïnes, calculi el subconjunt més petit possible de ciutats a on posar antenes de manera que les  $n$  ciutats tinguin cobertura.

---

# Introducció a la NP-completesa

1. *Tragicomèdia en tres actes.* Considereu els problemes **NP**-complets següents:

- **BIN-PACKING:** Donades  $n$  peces de mides  $d_1, \dots, d_n$  i  $k$  contenidors de capacitat  $m$ , decidir si es poden col·locar totes les peces en els contenidors quedant tots ells plens.
- **TRIPARTITE-MATCHING:** Donats tres conjunts  $C_1, C_2, C_3$  disjunts amb  $n$  elements cadascun d'ells, i un subconjunt  $S \subseteq C_1 \times C_2 \times C_3$ , decidir si es poden escollir  $n$  elements de  $S$  de manera que tot element de  $C_1, C_2$  i  $C_3$  aparegui una única vegada entre aquestes  $n$  ternes.
- **3-COLORABILITAT:** Donat un graf, decidir si es poden assignar colors als vèrtexs, a escollir d'entre tres colors possibles, de manera que cap aresta tingui els seus extrems del mateix color.
- **GRAF-HAMILTONIÀ:** Donat un graf, decidir si hi ha un cicle que passa per tots els seus vèrtexs un sol cop.
- **CONJUNT-DOMINADOR:** Donat un graf i un natural  $k$ , decidir si hi ha un subconjunt de  $k$  vèrtexs tals que tot altre vèrtex és adjacent a un d'ells.

Cadascuna de les escenes següents presenta un problema. Mostreu que aquest problema és difícil tot veient que la seva versió decisional es pot identificar amb algun dels problemes de la llista anterior.

*Escena 1:* En Roy, a petició d'en Jonny, és l'encarregat d'organitzar un sopar per al grupet de col·legues de sempre. Ha demanat taula en un restaurant; és una taula rodona molt gran i hi caben tots. Quan arriben al lloc, però, apareixen les primeres dificultats.

JONNY: Escolta Roy, es veu que algunes persones del grup estan enfadades entre elles. Seria convenient que dues persones enfadades no seguessin l'una al costat de l'altra.

ROY: Ja comencem... Mira, anem al gra. Vull que recopilis tota la informació al respecte; és a dir, per a cada parell de persones, si estan enfadades o no. Jo miraré com les podem fer seure sense que ningú tingui un veí de taula amb el qual està enfadat.

*Escena 2:* Tenint totes les dades a la ma, i després d'una bona estona, en Roy no ha trobat encara cap solució. A més, en Jonny torna a aparèixer amb cara de circumstancies.

JONNY: Ep, es veu que els que estan enfadats mútuament no volen ni tan sols seure a la mateixa taula. Però tranquil, he preguntat al restaurant i m'han dit que ens deixen tres taules grans.

ROY: Vejam, intentaré distribuir a la gent en tres grups de manera que en un mateix grup no hi hagi dues persones mútuament enfadades.

*Escena 3:* Mentre en Roy pensa com distribuir la gent al voltant de les tres taules, en Jonny apareix de nou informant que el restaurant és a punt de tancar, i que ja poden anar buscant un altre lloc per sopar.

ROY: Mira, abans de buscar un altre lloc, m'agradaria aclarir quatre coses a la gent: Això de "ai, ai, no vull seure amb en tal o amb en qual" és una criaturada! Vull parlar amb ells seriosament. El problema és que estic una mica afònic i si parlo amb tothom em quedaré sense veu. Vull que escullis  $k$  representants de manera que, per a qualsevol dels del grup, hi hagi un dels  $k$  representants amb qui no estigui enfadat. D'aquesta manera els  $k$  representants faran arribar a tot el grup el missatge.

*Escena 4:* Com que en Jonny no s'en surt en triar els  $k$  representants, finalment agafa un megàfon i comunica a tothom que si continuen amb tanta conya, hauran d'anar cap a caseta.

JONNY: Escolta, tinc un mapa de la ciutat on he marcat les places on hi ha bons restaurants, i els carrers que les uneixen. Podríem anar passejant per totes i veure si tenen taules lliures.

ROY: Ok, deixa'm una estona el mapa per escollir el trajecte, que fa fred i no tinc ganes de passar dues vegades pel mateix lloc.



Tresor meu...  
quin problema més llarg...



*Escena 5:* Malgrat tenir totes les dades a mà, i després d'una bona estona, en Roy no se'n surt. En Jonny el vol ajudar:

JONNY: Escolta Roy, jo tinc un munt de col·legues a la ciutat i a cadascuna d'aquestes places hi tinc un amic que hi viu. Dona'm pasta que els truco i els demano que mirin si hi ha lloc als restaurants de la seva plaça.

ROY: Mira tio, estic escurat i no tinc diners per tantes trucades. Te'n dono prou per fer  $m$  trucades. Si de cas, demana als teus amics que mirin no només a la plaça on viuen, sinó també a les places que disten un carrer d'on ells són. Escull als amics adequats perquè puguin mirar totes les places.

*Escena 6:* En no sortir-se'n, en Jonny truca al primer de la llista, i resulta que a prop seu hi ha un restaurant amb taules lliures. Quan hi arriben, resulta que només hi ha una taula amb  $k$  cadires i que tanquen el restaurant d'aquí poc.

JONNY: Ja està, tinc la solució: Anirem sopant primer uns i després els altres, quan algú hagi acabat, sortirà i n'entrarà un altre. Mira, fa un temps vaig dedicar-me a recopilar informació sobre quan trigava cada persona en sopar. Ja saps que tinc hobbies una mica estranys...

ROY: Mare meva...

JONNY: Amb aquestes dades, podem fer una planificació distribuint les persones a les cadires, de manera que tothom pugui sopar abans que tanquin.

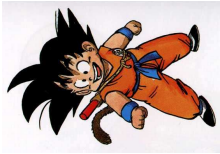
ROY: D'acord, dona'm aquesta llista del temps que triga cadascú a sopar i veuré que hi puc fer. Algun dia m'hauràs d'explicar quines altres dades reculls sobre els demés...

*Escena 7:* Entretant, en Jonny rep la trucada d'un amic dient que a prop d'allà hi ha un restaurant molt guai. Se n'hi van tots de dret. Quan hi arriben, resulta que només hi ha un munt de tauletes petites, i a sobre, totes de colors diferents.

JONNY: Escolta Roy. Es veu que la penya estan una mica avorrits i volen aprofitar la situació per a asseure's adequadament per parelles. Resulta que som tants nois com noies, i cadascú té les seves preferències. Però les taules de colors també juguen el seu paper. Per exemple, tal persona accepta seure amb tal altra només si la taula és de color rosa, i amb no sé quina altra només només si la taula és de color blau. Haurem de trobar una distribució que satisfaci les restriccions de tothom. Es veu que amb tanta estona sense menjar, la penya està una mica anada.

ROY: A la penya els fotrè jo una bona pinya. Mare de déu, no estic ara per gaires tonteries, eh! Però vaja, ho intentaré.

★ *Escena 8:* No se'n surten i s'ha fet molt tard. En Roy i en Jonny només troben una tasca on hi ha una taula amb una cadira. Això sí, obren un munt d'hores.



JONNY: Resulta que la tribe estan una mica ratllats. Alguns se'n van a fer un volt i tornaran més tard. Aquí tinc apuntat, per cada persona, a quina hora tornarà i a quina hora se n'haurà d'anar a casa. I encara tinc la llista del temps que necessita cada persona per sopar. Crec que, amb aquestes dades, hauria de ser fàcil veure si podem distribuir a tothom segons els seus horaris perquè puguin anar seient a la cadira i sopant.

ROY: Escolta Jonny, n'estic fins als collons. Vull que sàpigues que aquesta és la darrera cosa que faig per aquesta gent. Va, vinga, dona'm aquestes dades i veuré que hi puc fer.

[En aquesta escena no n'hi ha prou amb identificar un problema de la llista; cal fer una reducció.]

*Escena 9:* Finalment, en Roy i en Jonny decideixen sopar tots dos junts en aquell lloc, només els ha calgut demanar una altra cadira a la cambrera. Als demés, els han donat una adreça d'un lloc on només hi ha un descampat, a la zona amb major índex de delinqüència i criminalitat de la ciutat.

JONNY: Vinga home, no t'ho prenguis així! Això d'avui simplement ha estat mala sort.

ROY: Mira, de mala sort, res. No és el primer cop que em passa això. És que sóc un passarell i sembla que no n'aprendré mai. Ara, vull que escoltis amb atenció. Sé que t'ho he dit altres cops, però aquesta vegada va de debò: No tornaré a organitzar res per aquesta penya mai més. I quan dic mai és mai. De fet, ni tan sols tornaré a quedar amb ells.

JONNY: D'acord, d'acord! Llàstima, però. La setmana vinent havíem pensat anar tots a una casa de colònies. Ja saps, natura, aire pur, tranquil·litat, diversió, carn a la brasa, banys al riu tots despullats...

ROY: Vaja! Sona bé això! Però no sé, després les coses sempre acaben sortint malament.

JONNY: A més, aquest cop vindrà la Steffy. Te'n recordes?

ROY: LA STEFFY !?!?!?

JONNY: Sí, i diu que té moltes ganes de veure't. Crec que s'endurà una desil·lusió molt gran si no vens.

ROY: Hmmm... bé... potser n'he fet un gra massa. Si en el fons són bons païos... A vegades estan inaguantables, però són bons païos en definitiva, i això és el que compta. Al cap i a la fi, l'amistat està plena de mals moments, però sobretot de moments meravellosos plens de joia. Va vinga! Què punyetes? M'hi apunto! I si vols t'ajudo a organitzar les coses.

JONNY: Perfecte! Ets el millor! Un crack, sí senyor! Va, vinga, posem-nos-hi ara mateix. A veure, l'únic problema que podria haver-hi és que només hi ha  $k$  cotxes i...



2. *Partició.* Reduïu el problema SUBSET-SUM al problema de la PARTICIÓ: Donat un conjunt d'enters  $S$ , trobar si aquest es pot particionar en dos subconjunts  $S_1$  i  $S_2$  tal que  $S = S_1 \cup S_2$  i  $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ .

Demostreu que PARTICIÓ pertany a **NP**.



3. *Subgraf induït.* Reduïu el problema de CLIQUE al de SUBGRAF-INDUÏT: Donats dos grafs  $G_1$  i  $G_2$ , determinar si  $G_1$  és un subgraf induït de  $G_2$ .

Demostreu que SUBGRAF-INDUÏT pertany a **NP**.

4. *Subgraf esborrat.* Reduïu el problema de GRAF-HAMILTONIÀ a SUBGRAF-ESBORRAT: Donats dos grafs  $G_1$  i  $G_2$ , determinar si un graf isomorf a  $G_1$  s'obté a partir de  $G_2$  tot esborrant arestes.

Demostreu que SUBGRAF-ESBORRAT pertany a **NP**.

5. *Variant del problema d'empaquetament.* Reduïu el problema de BIN-PACKING (donats els naturals  $s_1, \dots, s_n, k, M$ , saber si els  $s_1 \dots s_n$  es poden separar en  $k$  grups diferents on cadascun sumi exactament  $M$ ) a la variant  $\text{BIN-PACKING}_{\leq}$  (donats els naturals  $s_1 \dots s_n, k, M$ , saber si els  $s_1 \dots s_n$  es poden separar en  $k$  grups diferents on cadascun sumi  $M$  com a molt).

Demostreu que  $\text{BIN-PACKING}_{\leq}$  pertany a **NP**.



6. *Fer un puzzle és difícil.* El problema del PUZZLE és: Donat un natural  $M$  i  $2n$  naturals  $s_1, t_1, \dots, s_n, t_n$ , on  $s_i, t_i$  representen les mides dels costats d'un rectangle, veure si es poden col·locar totes les peces omplint totalment un taulell  $M \times M$ .

Reduïu BIN-PACKING a PUZZLE. Demostreu que PUZZLE pertany a **NP**.

7. *Variant de satisfactibilitat.* Reduïu CNF-SAT a 3-SAT (com CNF-SAT, però on totes les clàusules tenen exactament 3 literals).

Demostreu que 3-SAT pertany a **NP**.



8. *Variant més difícil de satisfactibilitat.* Reduïu 3-SAT a 1-EN-3-SAT (com 3-SAT, però l'assignació ha de satisfer exactament un únic literal per clàusula).

Demostreu que 1-EN-3-SAT pertany a **NP**.

9. *Deduïr un traductor de paraules a partir d'exemples de traducció és difícil.* Reduïu 1-EN-3-SAT (vegeu problema anterior) a MORFISME-PARAULES: donades les paraules  $u_1, v_1, u_2, v_2, \dots, u_n, v_n$ , decidir si hi ha un morfisme de paraules  $\sigma$  tal que  $\sigma(u_i) = v_i$ . Un morfisme és una funció  $\sigma$  de paraules a paraules que compleix  $\sigma(uv) = \sigma(u)\sigma(v)$ . Es sol definir només pels símbols de l'alfabet, i llavors queda definit per extensió per qualsevol paraula:  $\sigma(a_1 a_2 \dots a_m) = \sigma(a_1)\sigma(a_2) \dots \sigma(a_m)$ .

Repetiu el problema, però reduïnt ara directament desde CNF-SAT.

Demostreu que MORFISME-PARAULES pertany a **NP**.

10. *Equacions lineals sobre els naturals.* Reduïu 1-EN-3-SAT a EQUACIONS-LINEALS: donat un conjunt d'equacions lineals sobre els naturals i amb variables interpretades sobre els naturals, saber si aquest sistema té solució.

Repetiu el problema, però reduïnt ara directament desde CNF-SAT.

- ☆☆ 11. *Inequacions lineals sobre els enters.* Reduïu 3-SAT a INEQUACIONS-LINEALS: donat un conjunt d'inequacions lineals sobre els enters amb variables enteres, saber si aquest sistema té solució.

12. *El joc de formar paraules.* El problema FORMA-PARAUULA consisteix a, donades  $m+1$  paraules  $w_0, w_1, \dots, w_m$ , veure si es poden escollir algunes paraules de  $w_1, \dots, w_m$  (sense repeticions) de manera que, concatenades en algun ordre, formin  $w_0$ .

Demostreu que FORMA-PARAUULA pertany a **NP**.

Considereu la següent “reducció” de BIN-PACKING a FORMA-PARAUULA:

- L'entrada de BIN-PACKING són  $n$  peces de mides  $d_1, \dots, d_n$  i  $k$  contenidors de capacitat  $q$ .
- Agafem  $m := n + k$ .
- Per a cada peça amb mida  $d_i$ , construïm una paraula  $w_i = \mathbf{a}^{d_i}$ , és a dir, una paraula formada per  $d_i$  as.
- Per a cada contenidor  $j$ , construïm una paraula  $w_{n+j}$  formada per una sola  $\mathbf{b}$ .
- Construïm la paraula  $w_0 = (\mathbf{a}^q \mathbf{b})^k$ .

Comproveu que una instància de BIN-PACKING és positiva si i només si la seva transformació a FORMA-PARAUULA és una instància positiva.

Perquè aquesta reducció no demostra que FORMA-PARAUULA és **NP**-complet?

13. Possiblement recordareu que en Jonny i en Roy havien quedat per anar amb la seva nombrosa colla (i l'Steffy!) a passar un cap de setmana a una casa de colònies. Avui han decidit anar a fer ràfting, però només queda una barca. A més, la barca només funciona si les persones que hi pugen pesen exactament  $T$  grams en total (perquè si pesen més, la barca s'enfonsaria i, si pesen menys, el corrent no seria prou fort per portar-los fins al final).

Llegiu el diàleg següent, assumint que tots els personatges diuen la veritat.

JONNY: Roy, demana a tothom el seu pes, per veure si val la pena que lloguem la barca.

ROY: Aquí ho tens, ja saps que m'agrada fer llistes de la gent.

STEFFY: D'això, nois... val més que no us hi poseu. El problema que voleu resoldre (donat un enter  $T$  i donats  $n$  enters estrictament positius  $p_1, \dots, p_n$ , determinar si n'hi ha uns quants que sumen  $T$ ) és **NP**-complet.

JONNY: Ostres, així ja l'hem fotuda!

ROY: Espera! Casualment, he portat una llàntia màgica que vaig comprar al mercat d'Istanbul, en una botiga on parlaven català. Em van dir que si la fregues, surt un geni que resol eficientment qualsevol problema **NP**.

JONNY: Va, vinga, passa-me-la!

*[En Jonny frega la llàntia i en surt un geni envoltat de fum.]*

GENI: Salutacions, noi. Si em dónes  $m$  enters  $b_1, \dots, b_m$ , determinaré al moment si hi ha algun subconjunt no buit que sumi zero.

JONNY: Merda, Roy! Aquest no és el nostre problema!

ROY: Ummm... potser aquell venedor d'Istanbul em va enganyar.

STEFFY: No patiu, nois. El venedor no va mentir... Tinc la solució!

- (a) Expliqueu quina solució ha trobat l'Steffy per saber si alguns d'ells poden fer ràfting (és a dir, doneu una reducció del problema d'en Jonny al problema del geni).
- (b) Acabeu de demostrar que el problema del geni és **NP**-complet.

14. Tenim un problema  $X$  i hem pogut establir que **SAT** es redueix a  $X$  (en símbols,  $\text{SAT} \leq_m X$ ). Quina de les següents afirmacions podem deduir:

- (a)  $X$  està a la classe NP, però no sabem si és o no NP-complet.
- (b)  $X$  és un problema NP-complet.
- (c) Cap de les anteriors.

Adicionalment, un company ha trobat ara una reducció en sentit invers,  $X \leq_m \text{SAT}$ . Què podem deduir ara?

- (i)  $X$  està a la classe NP, però no sabem si és o no NP-complet.
- (ii)  $X$  és un problema NP-complet.
- (iii) Cap de les anteriors.

15. Els problemes de la satisfactibilitat de fórmules booleanes (**SAT**) i de determinar si un graf donat és hamiltonià (**HAM**) són dos problemes NP-complets ben coneguts. El problema de determinar si un graf donat conté un circuit eulerià (**EUL**) té una solució en temps  $\Theta(n + m)$ , sent  $n$  el nombre de vèrtexos del graf i  $m$  el nombre d'arestes.

Quina de les següents frases podem afirmar que és certa?

- (a) **SAT** es redueix a **HAM** i **HAM** es redueix a **EUL**.
- (b) **EUL** es redueix a **SAT** i **SAT** es redueix a **HAM**.

- (c) SAT i HAM es redueixen l'un a l'altre; EUL només es redueix a HAM.
- (d) SAT es redueix a EUL i EUL es redueix a HAM.

N.B. Una resposta incorrecta resta 0.25 punts de la nota global.

## Soluciones

### Anàlisi d'algorismes

**Solució 1.5:** Se demuestra por inducción. Sea  $S_k = \sum_{i=1}^k i2^{-i}$ . Para  $k = 1$ ,  $S_1 = 1/2$  que es igual a la expresión  $2 - k2^{-k} - 2^{1-k} = 2 - 1/2 - 1 = 1/2$ . Establecida la base de inducción, suponemos que la fórmula es correcta para  $k$ , y demostramos que entonces es también correcta para  $k + 1$ :

$$\begin{aligned}
 S_{k+1} &= S_k + (k+1)2^{-(k+1)} = 2 - k2^{-k} - 2^{1-k} + (k+1)2^{-(k+1)} \\
 &= 2 - 2k2^{-(k+1)} - 2^{1-k} + (k+1)2^{-(k+1)} \\
 &= 2 - (k+1)2^{-(k+1)} + 2 \cdot 2^{-(k+1)} - 2^{1-k} \\
 &= 2 - (k+1)2^{-(k+1)} + 2^{-k} - 2^{1-k} \\
 &= 2 - (k+1)2^{-(k+1)} + 2^{-k}(1-2) = 2 - (k+1)2^{-(k+1)} - 2^{1-(k+1)}.
 \end{aligned}$$

**Solució 1.8:** Gráficamente  $H_n$  es la suma del área de  $n$  rectángulos, cada uno de los cuales tiene una base de longitud 1 y una altura de  $1/i$ ,  $i = 1..n$ . El rectángulo  $i$ -ésimo tiene su base entre  $i-1$  e  $i$  y su área es menor que el área de la función  $1/x$  entre  $x = i-1$  y  $x = i$ , si  $i > 1$ , y su área es 1 si  $i = 1$ ; su área es mayor que el área de la función  $1/(x+1)$  entre  $x = i-1$  y  $x = i$ . Por lo tanto, sumando para  $i = 1$  a  $i = n$ ,

$$\int_0^n \frac{dx}{1+x} \leq \sum_{i=1}^n \frac{1}{i} \leq 1 + \int_1^n \frac{dx}{x}.$$

Es decir,

$$\ln(1+x)|_0^n = \ln(n+1) \leq H_n \leq 1 + \ln(x)|_1^n = 1 + \ln n.$$

Luego  $H_n = \Theta(\ln n)$ . De hecho,  $H_n = \ln n + O(1)$ .

**Solució 1.12:** Com que el temps que es triga és  $f(n)$   $\mu\text{s}$ , la talla més gran que es pot processar en  $t$   $\mu\text{s}$  és  $\lfloor f^{-1}(t) \rfloor$ , on  $f^{-1}$  és la funció inversa de  $f$ , e.g., si  $f(n) = \sqrt{n}$ ,  $f^{-1}(t) = t^2$ . Només cal tenir en compte que 1 segon =  $10^6 \mu\text{s}$ , 1 minut =  $6 \cdot 10^7 \mu\text{s}$ , 1 hora =  $3.6 \cdot 10^9 \mu\text{s}$ , 1 dia =  $8.64 \cdot 10^{10} \mu\text{s}$ , 1 mes =  $2.592 \cdot 10^{12} \mu\text{s}$ , 1 any =  $3.1536 \cdot 10^{13} \mu\text{s}$  i 1 segle =  $3.1536 \cdot 10^{15} \mu\text{s}$ . A la taula hi ha entrades marcades amb '\*', doncs es tracta de números inconcebiblement grans.

|            | 1 segon                 | 1 minut                   | 1 hora                | 1 dia                   | 1 mes                    | 1 any                       | 1 segle                     |
|------------|-------------------------|---------------------------|-----------------------|-------------------------|--------------------------|-----------------------------|-----------------------------|
| $\log n$   | $9.9 \cdot 10^{301029}$ | $5.5 \cdot 10^{18061799}$ | *                     | *                       | *                        | *                           | *                           |
| $\sqrt{n}$ | $10^{12}$               | $3.6 \cdot 10^{15}$       | $1.296 \cdot 10^{19}$ | $7.46496 \cdot 10^{21}$ | $6.718464 \cdot 10^{24}$ | $9.945192969 \cdot 10^{26}$ | $9.945192960 \cdot 10^{30}$ |
| $n$        | $10^6$                  | $6 \cdot 10^7$            | $3.6 \cdot 10^9$      | $8.64 \cdot 10^{10}$    | $2.592 \cdot 10^{12}$    | $3.1536 \cdot 10^{13}$      | $3.1536 \cdot 10^{15}$      |
| $n^2$      | 1000                    | 7745                      | 60000                 | 293938                  | 1609968                  | 5615692                     | 56156922                    |
| $n^3$      | 100                     | 391                       | 1532                  | 4420                    | 13736                    | 31593                       | 146645                      |
| $2^n$      | 19                      | 25                        | 31                    | 36                      | 41                       | 44                          | 51                          |

**Solució 1.13:** Supongamos que el algoritmo de tiempo lineal tiene complejidad  $T_A(n) = a \cdot n$ . Se nos dice que si  $n = 10^8$  entonces  $T_A(n) = T$ . Y también se nos dice que la constante de proporcionalidad oculta es siempre la misma, así que el otro algoritmo tiene complejidad  $T_B(n) = an\sqrt{n}$ . Por lo tanto,

$$T_B(n) = \frac{T}{10^8} n\sqrt{n}.$$

Si disponemos de un tiempo  $T$  y despejamos  $n$

$$T = \frac{T}{10^8} n\sqrt{n} \implies n\sqrt{n} = 10^8 \implies n = 10^{16/3} = 215443.469$$

Si los procesadores pasan a ser 100 veces más rápidos entonces el algoritmo lineal podrá resolver una instancia de tamaño  $10^{10}$  en el tiempo  $T$  y por el razonamiento de antes tendremos

$$n\sqrt{n} = 10^{10} \implies n = 10^{20/3} = 4641588.834$$

**Solució 1.14:** Per ordre de creixement, de menor a major, tenim:

- $\frac{n}{\ln n}, \frac{n}{\log_2 n} \in \Theta\left(\frac{n}{\log n}\right)$ , ja que la base dels logaritmes no és rellevant.
- $3 \ln(7^n) = 3n \ln(7) \in \Theta(n)$
- $5n \ln n, \log_\pi(n^n) = n \log_\pi n \in \Theta(n \log n)$
- $4n\sqrt{n} \in \Theta(n^{3/2})$

**Solució 1.16:**



**Fragmento 5:** El coste de la sentencia más interna(`s++`) es  $\Theta(1)$ . El bucle de la `j` hace exactamente  $i$  iteraciones, por lo que su coste es, por la regla del producto,  $\Theta(i)$ . Para ser más precisos el coste es  $\Theta(\min\{i, 1\}) = \Theta(i + 1)$ . El bucle más externo, de la `i`, hace exactamente  $n$  iteraciones; pero el coste de cada iteración varía, así que el coste es

$$\sum_{i=0}^{n-1} \Theta(i + 1) = \Theta\left(\sum_{i=1}^n i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

**Fragmento 9:** El coste de este algoritmo es proporcional al número de iteraciones del bucle, ya que el cuerpo del bucle tiene coste constante. En cada iteración se dobla el valor de  $i$ , hasta que supera a  $n$ . El número de iteraciones  $k$  es el menor entero tal que  $2^k > n$ , es decir,  $k - 1 \leq \log_2 n < k$ , o dicho de otro modo  $k = \lceil \log_2 n \rceil$ . Por lo tanto el coste es  $\Theta(\log n)$ .

**Solució 1.22:** `// A y B son matrices m × n`  
`void suma_matrices(double A[][], double B[][],`  
`double C[][], int m, int n) {`  
`for (int i = 0; i < m; ++i)`  
`for (int j = 0; j < n; ++j)`  
`C[i][j] = A[i][j]+B[i][j];`  
`}`

El coste del cuerpo del bucle interno (sobre  $j$ ) es constante. El bucle sobre  $j$  hace  $n$  vueltas, así que su coste es  $\Theta(n)$ . El coste del bucle externo (sobre  $i$ ) es  $m$  veces  $\Theta(n)$  puesto que se hacen  $m$  iteraciones. Así el coste del algoritmo es  $\Theta(m \cdot n)$ . Cuando  $m = n$ , el coste es  $\Theta(n^2)$ . Se dice que el algoritmo es lineal puesto que la entrada consiste en  $2n^2$  números. Si llamamos  $N$  al tamaño de la entrada el coste del algoritmo es  $\Theta(N)$ , así que es lineal.

El resultado  $C$  de sumar dos matrices  $n \times n$  consiste en una matriz de  $n^2$  números. El coste mínimo posible vendrá de “pagar” un coste constante para calcular (y almacenar en  $C$ ) cada uno de esos números. Por tanto la complejidad de cualquier algoritmo que sume dos matrices  $n \times n$  es  $\Omega(n^2)$ .

**Solució 1.27:** Aplicamos en todos los casos el teorema maestro para recurrencias substractoras de la forma  $T(n) = a \cdot T(n - c) + \Theta(n^k)$ .

- Como  $a = 1$  la solución es  $\Theta(n^{k+1})$ . En este caso,  $T(n) = \Theta(n)$ .
- Aunque  $c = 2$ , la solución sigue siendo  $T(n) = \Theta(n)$ .
- Tenemos  $a = c = 1$  y  $k = 1$ ; por tanto  $T(n) = \Theta(n^2)$ .
- Tenemos  $a = 2 > 1$  y por lo tanto la solución es  $\Theta(a^{n/c}) = \Theta(2^n)$ .

**Solució 1.28:** Usamos el teorema maestro para recurrencias divide-y-vencerás, es decir, de la forma  $T(n) = a \cdot T(n/b) + \Theta(n^k)$ ; en cada caso calcularemos  $\alpha = \log_b a$ .

- $a = b = 2, k = 0$ . Como  $\alpha = 1 > 0$  el resultado es  $T(n) = \Theta(n^\alpha) = \Theta(n)$ .
- $a = b = 2, k = 2$ ;  $\alpha = 1 < 2$  y la solución es por tanto  $T(n) = \Theta(n^k) = \Theta(n^2)$ .
- Aquí  $a = b = 2, k = 1$  y  $\alpha = 1 = k$ ; además sobre el coste no recursivo sólo tenemos una cota superior  $\mathcal{O}(\cdot)$ . Así que  $T(n) = \mathcal{O}(n \log n)$ .
- $a = 4, b = 2, k = 2$ ; como  $\alpha = \log_2 4 = 2 = k$  tenemos  $T(n) = \Theta(n^2 \log n)$ .
- En este caso  $a = 1$  y  $b = 10/9$  (ojo! no es  $9/10$ !!); además  $k = 1$ . Como  $\alpha = \log_{10/9} 1 = 0 < k$  la solución es  $T(n) = \Theta(n)$ .

**Solució 1.37:** Usando el teorema maestro obtenemos  $T_A(n) = \Theta(n^{\log_2 7})$ . El exponente está entre 2 y 3. En el segundo algoritmo el coste depende de  $x$ : si  $\log_4 x > 2$  entonces  $T_B(n) = \Theta(n^{\log_4 x})$ . Si  $\log_4 x = 2$  entonces el coste es  $\Theta(n^2 \log n)$  y si  $\log_4 x < 2$  entonces el coste es  $\Theta(n^2)$ . Para que los costes de  $A$  y  $B$  se igualen hemos de tener  $\log_2 7 = \log_4 x$ . O equivalentemente  $T_A(n) = \theta(T_B(n))$  si y sólo si  $4^{\log_2 7} = 4^{\log_4 x} = x$ . En otras palabras si  $x = 4^{\log_2 7} = 49$  entonces  $A$  y  $B$  tienen el mismo coste asintótico, si  $x < 49$  entonces  $B$  es más rápido y si  $x > 49$  entonces  $A$  es mejor. Así que  $x = 48$  es el entero más grande para el cual  $B$  es asintóticamente mejor que  $A$ .

**Solució 1.38:** Sea  $F(n)$  el coste en caso peor de la función  $F$  cuando el tamaño del subvector a procesar es  $n = j - i + 1$ . Si  $n = 0$  entonces  $F(0) = \text{cte.}$ ; si  $n > 0$  entonces se hacen cuatro llamadas recursivas cada una de las cuales recibe un subvector con aproximadamente  $n/4$  elementos y algunos cálculos sencillos. Por lo tanto

$$F(n) = \Theta(1) + 4F(n/4), \quad n > 0,$$

cuya solución es  $F(n) = \Theta(n)$ , ya que el coste no recursivo es  $\Theta(n^0)$  y  $\log_4 4 = 1 > 0$ .

**Solució 1.39:** Analicemos en primer lugar el algoritmo **A**. En el caso recursivo se hacen tres llamadas, cada una de las cuales opera sobre un subvector con aproximadamente la mitad de los elementos de la entrada original. El coste no recursivo corresponde a la llamada a  $f$  y algunos cálculos sencillos, de manera que es  $\Theta(1)$ . Por lo tanto, el coste  $T_A(n)$  del algoritmo **A** satisface la siguiente recurrencia

$$T_A(n) = 3T_A(n/2) + \Theta(1).$$

Aplicamos el teorema maestro con  $a = 3, b = 2$  y  $k = 0$ . Como  $\alpha = \log_b a = \log_2 3 > 0$ , la solución es  $T_A(n) = \Theta(n^\alpha) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\dots})$ .

El coste del segundo algoritmo viene descrito por la recurrencia divisora

$$T_B(n) = 3T_B(n/3) + \Theta(n)$$

Aplicando de nuevo el teorema maestro con  $a = 3 = 3^1 = b^k$ , la solución es  $T_B(n) = \Theta(n \log n)$ .

El algoritmo **B** es asintóticamente más eficiente que **A**, ya que  $n \log n = o(n^{\log_2 3})$ .

**Solució 1.46:** L'algorisme  $B$  és més eficient que l'algorisme  $A$ . Si prenem logaritmes de les respectives complexitats tenim  $\sqrt{\log_2 n} \ll \frac{1}{2} \log_2 n$ , és clar que la complexitat de  $B$  és molt menor que la d' $A$ .

**Solució 1.47:** Sigui  $\alpha = \log_4 x$ . Com que el cost no recursiu és  $\Theta(n^{1/2})$ , comparant  $\alpha$  i  $k = 1/2$  tenim tres possibilitats:

1. Si  $\alpha < 1/2$  llavors  $T(n) = \Theta(n^{1/2})$ ; això es produeix quan  $x < 2$ .
2. Si  $\alpha = 1/2$  llavors  $T(n) = \Theta(n^{1/2} \log n)$ ; això es produeix quan  $x = 2$ .
3. Si  $\alpha > 1/2$  llavors  $T(n) = \Theta(n^\alpha) = \Theta(n^{\log_4 x})$ ; això es produeix quan  $x > 2$ .

## Estructures de dades

**Solució 2.1:**

```

1 : 3441, 2001, 128181
2 : 3722, 3202
3 : 1983
5 : 365
7 : 18267
10 : 4830
12 : 3412, 7812
13 : 4893, 3313
14 : 3874
18 : 498
19 : 1299, 999

```

**Solució 2.3:**

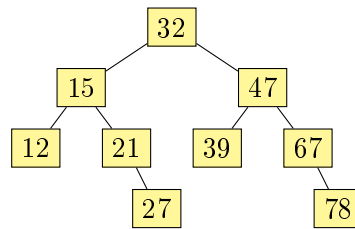
| 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8   | 9    | 10   | 11  | 12    | 13   | 14   | 15  | 16    | 17 | 18 | 19 |
|------|------|------|------|------|------|------|------|-----|------|------|-----|-------|------|------|-----|-------|----|----|----|
| 4830 | 3441 | 3412 | 1983 | 4893 | 3874 | 3722 | 3313 | 498 | 2001 | 3202 | 365 | 12818 | 7812 | 1299 | 999 | 18267 |    |    |    |

**Solució 2.5:** El algoritmo crea una tabla de dispersión  $T$  inicialmente vacía y a continuación procede a recorrer secuencialmente la lista. Para cada elemento  $x$  de la lista, busca a  $x$  en la tabla de dispersión  $T$ . Si el elemento  $x$  ya estaba presente en  $T$ , se termina el recorrido e se informa de que la lista contiene elementos repetidos (concretamente  $x$  sería uno de ellos). En caso contrario, si  $x$  no está en  $T$ , se inserta en ella. Si el recorrido se completa, significa que todos los elementos de la lista eran diferentes.

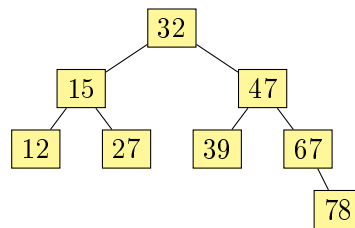
Cada consulta e inserción en una tabla de dispersión tiene coste esperado  $\Theta(1)$  y se efectúan  $n$  iteraciones en caso peor (todos los elementos son distintos), siendo  $n$  la longitud de la lista. Por lo tanto el coste total del algoritmo es  $\Theta(n)$  en caso medio.

**Solució 2.10:**

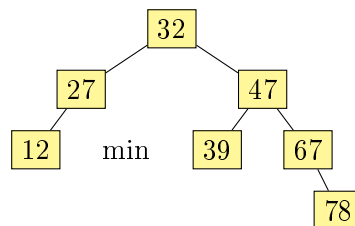
1. Eliminació de la clau 63:



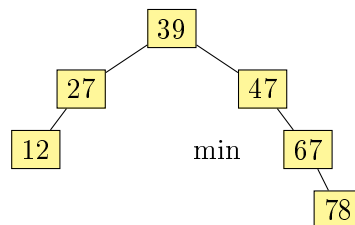
2. Eliminació de la clau 21:



3. Eliminació de la clau 15: la substituïm per la clau mínima del seu subarbre dret



4. Eliminació de la clau 32: substituïm pel mínim del subarbre dret



### Solució 2.18:

```

// a no es buit
Elem maxim(abc a) {
 if (a -> fd == null) // el maxim es situa a l'arrel
 return a -> x;
 else
 // hi ha fill dret, i el maxim d'a sera el
 // maxim del seu fill dret
 return maxim(a -> fd);
}

```

El cost és proporcional a l'alçada de l'arbre  $a$ . En cas pitjor, aquesta alçada serà  $n$  i per tant el cost de la funció, en cas pitjor, és  $\Theta(n)$ .

**Solució 2.20:** Si el BST es vació, la lista a devolver es también vacía. Supongamos que el BST no es vació y que la clave en la raíz es  $x$ . Si  $x < x_1$  entonces todas las claves buscadas deben encontrarse, si las hay, en el subárbol derecho. Análogamente, si  $x_2 < x$  se proseguirá la búsqueda recursivamente en el subárbol izquierdo. Finalmente, si  $x_1 \leq x \leq x_2$  entonces puede haber claves que caen dentro del intervalo tanto en el subárbol izquierdo como en el derecho. Para respetar el orden decreciente en la lista, deberá buscarse recursivamente a la izquierda, luego listar la raíz anteponiéndola a los elementos ya listados y finalmente buscarse recursivamente a la derecha, anteponiendo los elementos encontrados en dicho subárbol a los que ya hubiera en la lista  $L$ .

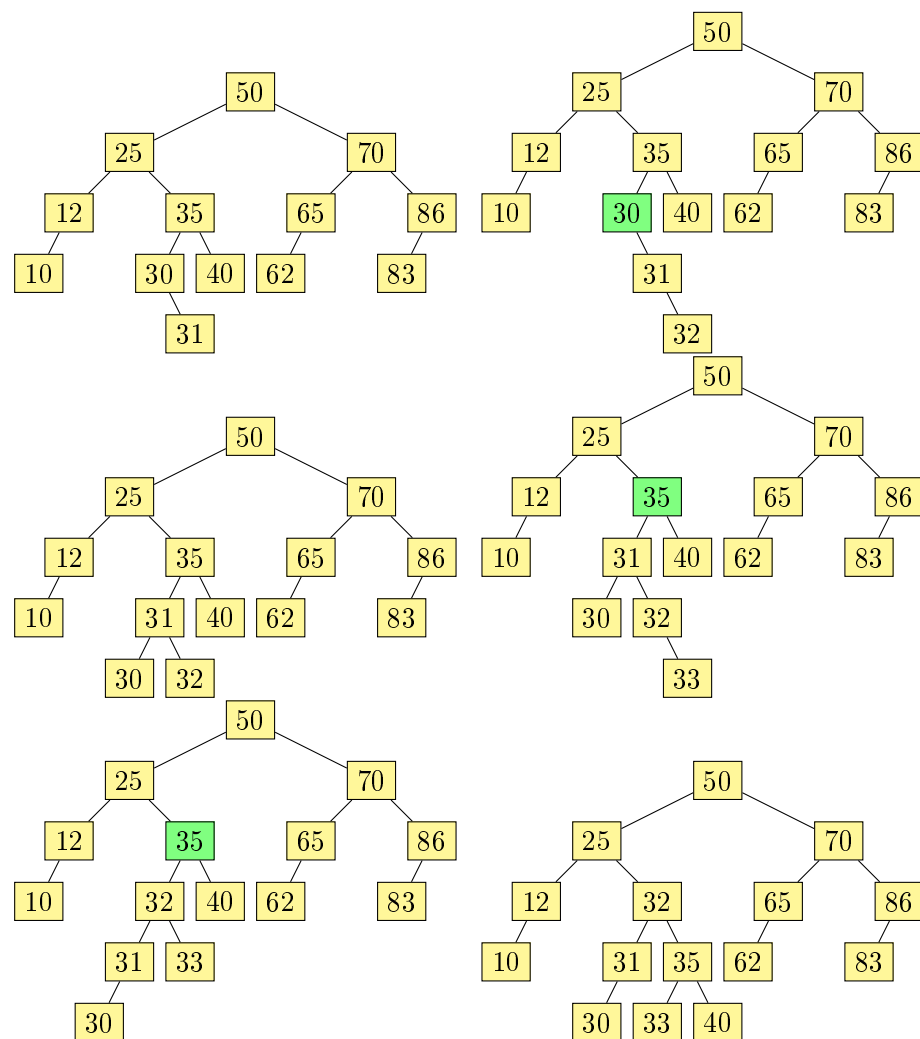
```
template <typename K, typename V>
void BST<K, V>::range_search(nodo* p,
 const K& k1, const K& k2, list<pair<K, V> >& L) {
 if (p == NULL) return;
 if (k1 <= p -> clave_)
 range_search(p -> izq_, k1, k2, L);
 if (k1 <= p -> clave_ && p -> clave_ <= k2)
 L.push_front(make_pair(p -> clave_, p -> valor_));
 if (p -> clave_ <= k2)
 range_search(p -> der_, k1, k2, L);
}
```

**Solució 2.21:** Si l'arbre  $a$  és buit la resposta és 0. Si l'arrel d' $a$  és menor que  $x$ , llavors tots els elements del subarbre esquerre són menors que  $x$ , la pròpia arrel ho és, i a més alguns elements del subarbre dret també. Si l'arrel d' $a$  és més gran o igual que  $x$  llavors només hem de continuar buscant elements menors que  $x$  dins del subarbre esquerre d' $a$ .

```
int menors (abc a, Elem x) {
 if (a == null) return 0;
 if (a->x >= x) return menors(a->fe, x);
 return menors(a->fe, x) + 1 + menors(a->fd, x);
}
```

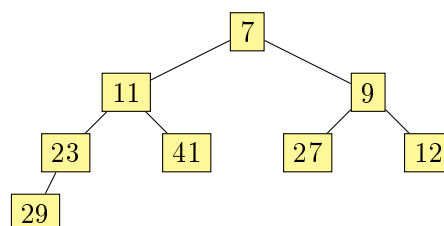
En el cas pitjor, tots els elements de l'arbre són més petits que  $x$  i hem de recórrer l'arbre sencer; per tant en cas pitjor el cost és  $\Theta(n)$ .

**Solució 2.25:** Les successives figures mostren l'evolució de l'AVL. L'inserció de la clau 31 no provoca cap problema. La següent inserció, de la clau 32, provoca una violació de l'invariant dels AVLs al node 30 (marcat en verd) i la següent figura mostra el resultat un cop que fem la corresponent rotació simple. La inserció de la clau 33 requereix una rotació doble per a corregir el desequilibri que hi ha al node 35.



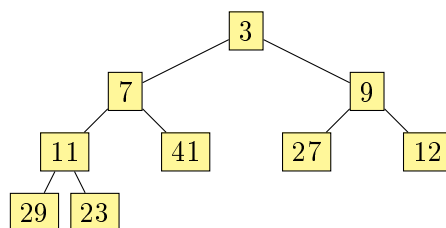
**Solució 2.35:** L'estat inicial és aquest:

|   |    |   |    |    |    |    |    |  |
|---|----|---|----|----|----|----|----|--|
| 7 | 11 | 9 | 23 | 41 | 27 | 12 | 29 |  |
|---|----|---|----|----|----|----|----|--|



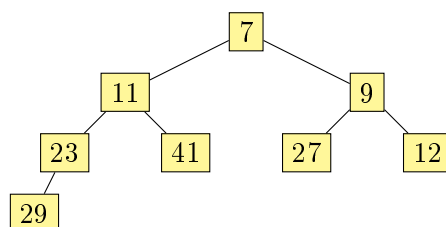
Per a inserir el nou element (el 3) es posa a l'última posició del vector i s'aplica un operació de "surar" que en aquest cas el fa pujar fins l'arrel donat que és mínim. El heap queda així:

|   |   |   |    |    |    |    |    |    |
|---|---|---|----|----|----|----|----|----|
| 3 | 7 | 9 | 11 | 41 | 27 | 12 | 29 | 23 |
|---|---|---|----|----|----|----|----|----|



Finalment, per eliminar el mínim (que és el 3), es substitueix l'arrel per l'últim element del heap (el 23) i s'aplica una operació per a enfonsar-la, fins que es reestableixi la condició de heap. Aquest queda així:

|   |    |   |    |    |    |    |    |  |
|---|----|---|----|----|----|----|----|--|
| 7 | 11 | 9 | 23 | 41 | 27 | 12 | 29 |  |
|---|----|---|----|----|----|----|----|--|



**Solució 2.39:** El algoritmo que aplicamos hunde sucesivamente los elementos  $A[n/2]$ ,  $A[n/2 - 1], \dots, A[1]$ . La representación en vector del *min-heap* resultante es:

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 11 | 21 | 27 | 45 | 53 | 97 | 34 | 78 |
|----|----|----|----|----|----|----|----|

**Solució 2.40:**

1. Cierto. Para toda  $i$ ,  $1 \leq i \leq \lfloor n/2 \rfloor$ , se cumple que  $A[i] \leq A[2i]$  y  $A[i] \leq A[2i+1]$ , ya que la tabla está ordenada ascendentemente. En otras palabras, la tabla satisface el invariante de un *min-heap*.
2. Cierto. En el caso peor, el elemento recién insertado (en  $A[n+1]$ ) es mayor que cualquiera de los que ya hay, de manera que tendremos que “flotarlo” hasta la raíz ( $A[1]$ ), lo que exige realizar  $\Theta(\log n)$  intercambios (y el coste del algoritmo es proporcional a dicho número).
3. Falso. Con toda seguridad, el elemento máximo debe ser una hoja del *min-heap*, pues por definición no puede tener sucesores. Pero un *min-heap* tiene aproximadamente  $n/2$  hojas, y el elemento máximo del *heap* puede estar en cualquiera de ellas. La estructura del *heap* no nos proporciona más información, por lo que no queda más remedio que pagar un coste  $\mathcal{O}(n)$ .

4. Cierto. Supongamos que los elementos del heap son los números del 1 a  $n$  (puesto que nos dicen que son todos diferentes) y que  $n = 2^k - 1$ . Entonces  $n$  ocupa la raíz del heap. Supongamos que  $n - 1$  es la raíz del hijo izquierdo, que el subárbol izquierdo del hijo izquierdo contiene los  $2^{k-2} - 1 = (n + 1)/4 - 1$  elementos más pequeños (p.e., del 1 al  $(n + 1)/4 - 1$ ) y que el subárbol derecho del hijo izquierdo contiene los  $2^{k-2} - 1 = (n + 1)/4 - 1$  elementos siguientes (p.e., del  $(n + 1)/4$  al  $(n + 1)/2 - 2$ ). Supongamos finalmente que desde la raíz hasta la hoja más a la derecha tenemos sucesivamente  $n - 2, n - 3, \dots, n - k = n - \log_2(n + 1)$ . El resto de elementos, del  $(n + 1)/2 - 1$  al  $n - k - 1$  se reparten arbitrariamente (respetando la condición de *max-heap*) en los subárboles izquierdos de  $n - 2, n - 3, \dots, n - k + 1$ . Cuando eliminamos el máximo ponemos la hoja más a la derecha (que es  $n - k$ ) en la raíz, y resulta que es menor que  $n - 2$  y que  $n - 1$ . Se hace el intercambio entre ella y  $n - 1$ , y como  $n - k$  es mayor que  $(n + 1)/4 - 1$  y que  $(n + 1)/2 - 2$ , finalizamos. Ha bastado tiempo  $\Theta(1)$ .

### Solució 2.45:

1. Implementació en C++ de la funció `insert`:

```

arbreprio insert(arbreprio a, int p) {

 if (a == NULL) {
 node* n = new node;
 n -> prio = p;
 n -> left = n -> right = NULL;
 return n;
 };

 // si la prioritat nova és més gran que cap altra
 // passa a ser la nova arrel
 if (a -> prio < p) {
 node* n = new node;
 n -> prio = p;
 n -> right = NULL;
 n -> left = a;
 return n;
 } else {
 // si no hi ha fill esquerre o la prioritat
 // nova és menor que l'arrel del fill esquerre
 // insertem recursivament a l'esquerra; sino
 // ho fem a la dreta
 if (a -> left == NULL ||
 a -> left -> prio > p)
 a -> left = insert(a -> left, p);
 else
 a -> right = inserta(a -> right, p);
 }
}

```



```

 return a;
 }
}
```

2. Recurrència i resolució:

El cas pitjor es dona quan les prioritats s'insereixen en ordre decreixent. Llavors sempre entrarem per la branca `a -> prio >= p\verb`; es a dir, no inserim a l'arrel i el subarbre dret està buit. Per tant el cost vindrà donat per la recurrència:

$$I(n) = \Theta(1) + I(n - 1), I(0) = \Theta(1)$$

la solució de la qual és  $I(n) = \Theta(n)$ .

Nota addicional: Com el cost de fer una inserció en cas pitjor és  $\Theta(i)$  quan l'arbre té  $i$  elements, el cost de fer  $n$  insercions serà  $\Theta(n^2)$ .

**Solució 2.53:** La solució consisteix en posar en un (min-)heap els  $n$  primers elements de les  $n$  taules donades. Es farà servir un `crea_heap`, no  $n$  insercions. El cost d'aques primer pas de l'algorisme és  $\Theta(n)$ . Junt amb cada element del heap guardem addicionalment l'índex de la taula de la qual procedeix. Llavors entrem en un bucle que repetirem  $k_n$  vegades, treiem el mínim element del heap i posant un element de la seva mateixa taula al heap. Si a la taula corresponent ja no queden elements posem al heap l'element màxim global (que es pot calcular prèvia i fàcilment mirant l'últim element de cada taula i quedant-se amb el més gran, amb cost  $\Theta(n)$ ). Això ens donarà el  $k_n$ -èssim element global més petit amb cost  $\Theta(k_n \log n)$ .

Demostrarem la correcció d'aquest algorisme per inducció sobre le nombre d'iteracions realitzades. La nostra hipòtesi d'inducció és que després de  $j$  iteracions, hem extret del heap els  $j$  elements més petits globalment i que el heap conté l'element més petit de cada taula excepte els que ja hem tret.

Per  $j = 0$  l'hipòtesi d'inducció és veritat trivialment. Si la hipòtesi és certa per  $j$  iteracions, llavors l'element mínim del heap és l'element més petit global que encara no s'ha extret. Efectivament, tots els elements que ens queden a les taules i que no em posat al heap són més grans que l'element corresponent de la seva taula que sí hem posat al heap. Com extreiem el mínim del heap i posem un element extret de la taula a la qual l'element extret pertanyia, es restableix l'hipòtesi d'inducció per a la iteració  $j + 1$ .

**Solució 2.54:** Pel primer apartat, es construeix un min-heap amb els  $n$  elements del vector original amb cost  $\Theta(n)$ . A continuació, s'extreuen  $m$  elements posant-los al final del vector, de manera semblant al *heapsort*. Això fa que els  $m$  elements més petits ocupin les  $m$  últimes components del vector, en ordre decreixent. Aquesta fase té cost  $\Theta(m \log n)$  en cas pitjor. Per últim, invertim el vector amb cost  $\Theta(n)$  per a ubicar els  $m$  elements més petits en les primeres  $m$  components del vector, en ordre creixent. El cost total és  $\Theta(n + m \log n)$  en cas pitjor, com es demanava.

Pel segon apartat, construirem un max-heap amb els primers  $m$  elements del vector, amb cost  $\Theta(m)$ . A continuació farem un recorregut sobre els  $n - m$  elements restants

fent el següent: comparem el màxim del heap amb l'element  $x$  en curs; si el màxim és més petit o igual que  $x$ , avançem al següent element del vector; si el màxim és més gran, llavors el màxim i  $x$  s'intercanvien i es fa un “enfosar” sobre l'element  $x$  que ocupa  $A[1]$ . D'aquesta manera el heap contindrà els  $m$  elements més petits vists fins al moment, a cada iteració. Aquesta fase té cost  $\Theta((n - m) \log m) = \Theta(n \log m)$ .

Finalment, el max-heap situat sobre  $A[1..m]$  s'ordena com es fa en el *heapsort*, amb cost en cas pitjor  $\Theta(m \log m)$ . Tot plegat el cost d'aquest altre algorisme és  $\Theta(n \log m)$ .

Comparant l'algorisme 1 amb l'algorisme 2, l'algorisme 1 és millor que l'algorisme 2 excepte si  $m = \Theta(n)$ . Llavors els dos algorismes tenen un cost asimptòticament equivalent.

El segon algorisme és interessant si els elements van arribant d'un en un, i només cal memòria en  $\Theta(m)$ ; l'algorisme 1 necessita que els  $n$  elements vinguin donats des del principi i necessita memòria en  $\Theta(n)$ .

**Solució 2.56:** Per a la representació dels *mf-sets* farem servir un vector  $A$  tal que si  $i$  no és el representant de la seva classe llavors  $A[i]$  és el pare de  $i$  i  $A[i] < 0$  en cas contrari. En l'algorisme bàsic, si  $i$  és una arrel llavors  $A[i] = -1$ . En els algorismes amb unió per talla si  $i$  és una arrel llavors  $-A[i]$  és la talla de l'arbre arrelat a  $i$ . En els algorismes amb unió per alçada, si  $i$  és una arrel  $-A[i]$  és l'alçada de l'arbre arrelat a  $i$ .

És a dir, la partició és representa mitjançant un bosc d'arbres, amb apuntadors al pare. Per a cada cas mostrem els continguts del vector en aplicar la seqüència d'operacions donades.

- Algorisme bàsic:

|    |   |    |   |   |   |   |   |   |   |   |   |   |    |    |
|----|---|----|---|---|---|---|---|---|---|---|---|---|----|----|
| -1 | 1 | 15 | 3 | 3 | 3 | 1 | 1 | 8 | 3 | 3 | 3 | 3 | -1 | 14 |
|----|---|----|---|---|---|---|---|---|---|---|---|---|----|----|

- Amb unió per talla:

|    |   |     |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| -5 | 1 | -10 | 3 | 3 | 3 | 1 | 1 | 8 | 3 | 3 | 3 | 3 | 3 | 3 |
|----|---|-----|---|---|---|---|---|---|---|---|---|---|---|---|

- Amb unió per alçada:

|    |   |    |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|----|---|---|---|---|---|---|---|---|---|---|---|---|
| -3 | 1 | -2 | 3 | 3 | 3 | 1 | 1 | 8 | 3 | 3 | 3 | 3 | 3 | 3 |
|----|---|----|---|---|---|---|---|---|---|---|---|---|---|---|

- Amb unió per talla i compressió de camins:

|    |   |     |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| -5 | 1 | -10 | 3 | 3 | 3 | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 |
|----|---|-----|---|---|---|---|---|---|---|---|---|---|---|---|

- Amb unió per alçada i compressió de camins:

|    |   |    |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|----|---|---|---|---|---|---|---|---|---|---|---|---|
| -3 | 1 | -2 | 3 | 3 | 3 | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 |
|----|---|----|---|---|---|---|---|---|---|---|---|---|---|---|

## Grafs

**Solució 3.7:** En un graf no dirigit  $G = (V, E)$ , cada aresta  $\{v, w\} = e \in E$  contribueix 1 al grau de cadascun dels dos vèrtexos (no necessàriament diferents)  $v, w \in V$  que hi incideixen i per tant, contribueix 2 al sumatori del grau de tots els vèrtexos.

**Solució 3.13:** El resultado del algoritmo se devolverá a través de dos vectores **ge** y **gs**, de tal modo que **ge**[*v*] sea el grado de entrada del vértice *v* y **gs**[*v*] el grado de salida. Nuestra solución se basa en aplicar el esquema de recorrido en profundidad.

```

procedure CALCULAGRADOS(G, ge, gv)
 for all v ∈ G do
 visitado[v] ← false
 ge[v] ← 0; gs[v] ← 0
 end for
 for all v ∈ G do
 if ¬visitado[v] then
 CALCULAGRADOS-REC(G, v, visitado, ge, gs)
 end if
 end for
end procedure

procedure CALCULAGRADOS-REC(G, v, visitado, ge, gv)
 visitado[v] ← true
 for all w ∈ SUCCESORS(v, G) do
 if ¬visitado[w] then
 ge[w] ← ge[w] + 1
 gs[v] ← gs[v] + 1
 CALCULAGRADOS-REC(G, w, visitado, ge, gs)
 end if
 end for
end procedure

```

Puesto que en cada visita de un vértice o un arco se incurre en un coste constante, el coste total del algoritmo es  $\Theta(|V| + |E|)$ , siendo  $|V|$  y  $|E|$  el número de vértices y de arcos de  $G$ , respectivamente.

El problema también se puede resolver, con el mismo coste, simplemente examinando la representación del grafo. Así, el grado de salida de  $v$  es igual a la longitud de la lista de sucesores de  $v$ . Para obtener los grados de entrada se recorre cada lista de sucesores, y para cada elemento se incrementa el grado de entrada del vértice correspondiente.

**Solució 3.17:** La transposició d'un graf dirigit  $G = (V, E)$  d'ordre  $n$  i talla  $m$  representat amb una matriu d'adjacència es pot aconseguir fent la transposició de la matriu d'adjacència, amb cost  $\Theta(n^2)$ . L'algorisme és pràcticament trivial:

```

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

```

```

 int tmp = G[i][j]; G[i][j] = G[j][i]; G[j][i] = tmp;
 }
}

```

La transposició d'un graf dirigit  $G = (V, E)$  d'ordre  $n$  i talla  $m$  representat amb llistes d'adjacència es pot aconseguir fent un recorregut de les llistes d'adjacència, inserint l'arc  $(v, w)$  al graf dirigit  $G^T = (V, E^T)$  en visitar el vèrtex  $v$  a la llista d'adjacència del vèrtex  $w$ , amb cost temporal i espacial  $\Theta(n + m)$ .

```

for $w \leftarrow 1, |V|$ do
 $G^T[w] \leftarrow \emptyset$
end for
for $v \leftarrow 1, |V|$ do
 $L \leftarrow G[v]$
 ▷ $G[v]$ = llista d'adjacents a v
 for $w \in L$ do
 $G^T[w].\text{APPEND}(v)$
 end for
end for

```

**Solució 3.19:** Un algorisme trivial per a aquest problema té cost temporal  $\Theta(n!/(n-4)!) = \Theta(n^4)$  si el graf és representat amb una matriu d'adjacència.

```

function QUADRE(V, E)
 $n \leftarrow |V|$
 for $v \leftarrow 1, n$ do
 for $w \leftarrow 1, n$ do
 for $x \leftarrow 1, n$ do
 for $y \leftarrow 1, n$ do
 if $\{v, w\}, \{w, x\}, \{x, y\}, \{y, v\} \in E$ then
 return true
 end if
 end for
 end for
 end for
 end for
 return false
end function

```

Una altre algorisme trivial per a aquest problema també té cost temporal  $\Theta(n^4)$  si el graf és representat amb una matriu d'adjacència.

```

function QUADRE(V, E)
 for all vèrtex $v \in V$ do
 for all vèrtex w adjacent amb el vèrtex v do
 if $v \neq w$ then
 for all vèrtex x adjacent amb el vèrtex w do
 if $v \neq x$ and $w \neq x$ then
 for all vèrtex y adjacent amb el vèrtex x do

```

```

 if $v \neq y$ and $w \neq y$ and $x \neq y$ then
 if $\{v, y\} \in E$ then
 return true
 end if
 end if
 end for
end if
end for
end for
return false
end function

```

Un algorisme més eficient consisteix a buscar, per a cada parell  $\{v, w\}$  de vèrtexos diferents, un altre parell  $\{x, y\}$  de vèrtexos diferents (entre sí i també dels anteriors) tal que tant  $x$  com  $y$  siguin adjacents amb  $v$  i amb  $w$ .

```

function QUADRE(V, E)
 for all parell de vèrtexos $\{v, w\}$ de V amb $v \neq w$ do
 $X := \{x \in V \mid \{v, x\} \in E\}$
 $Y := \{y \in V \mid \{w, y\} \in E\}$
 if $|X \cap Y \setminus \{v, w\}| \geq 2$ then
 return true
 end if
 end for
 return false
end function

```

Aquest algorisme es pot implementar amb cost temporal  $\Theta(n^3)$  si el graf està representat amb una matriu d'adjacència i amb cost temporal  $O(n^3)$  si el graf està representat amb llistes d'adjacència.

**Solució 3.20:** El problema és pot modelar amb un graf dirigit  $G = (V, E)$  ón  $V$  correspón al grup d' $n$  persones i hi haurà un arc  $(i, j) \in E$  si la persona  $i$  coneix la persona  $j$ . Una celebritat és un vèrtex amb grau d'entrada  $n - 1$  i grau de sortida 0. Si fessim un recorregut del graf per a calcular els graus dels vèrtexos podríem resoldre el problema però el cost seria en cas pitjor  $\Theta(n + m)$  i  $m$  pot ser molt gran si el graf fos dens.

Es pot trobar una solució molt més eficient si tenim en compte les dues següents observacions: 1) un graf només pot tenir una celebritat o cap; 2) si  $(i, j) \in E$  llavors  $i$  no pot ser una celebritat ( $i$  coneix algú) i si  $(i, j) \notin E$  llavors  $j$  no pot ser una celebritat (algú no coneix  $j$ ).

Apliquem ara un raonament per inducció: agafem una  $x$  i una  $y$  diferents qualsevols, p.e.  $x = 1$  i  $y = 2$ , i mirem si  $(x, y) \in E$  o no. Si  $(x, y) \in E$ , descartem  $x$  i mirem de trobar la celebritat en el graf  $G'$  de  $n - 1$  persones amb  $V(G') = V - \{x\}$ . Si, pel contrari,  $(x, y) \notin E$  llavors descartem  $y$  i mirem de trobar la celebritat en el graf  $G''$  amb  $V(G'') = V - \{y\}$ . En la solució en pseudocodi que donem més avall mantenim en el

primer bucle el candidat  $i$  a ser una celebritat.

Cada pas elimina una persona i per tant, en  $n - 1$  passos tindrem un graf amb un únic candidat  $i$  a celebritat.

Serà suficient comprovar que per a tot  $j \in V$ ,  $j \neq i$ , tenim  $(j, i) \in E$  i  $(i, j) \notin E$ .

```

function UNIVERSAL SINK(V, E)
 $i \leftarrow 1$
 for all $j \leftarrow 2, n$ do
 if $(i, j) \in E$ then
 $i \leftarrow j$
 end if
 end for
 for all j from 1 to n do
 if $(i, j) \in E$ or $((j, i) \notin E$ and $i \neq j)$ then
 return false ▷ no hi ha celebritat
 end if
 end for
 return true ▷ i és una celebritat
end function

```

Podeu consultar les següents referències per a més informació:

- Udi Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989, §5.5.
- Gabriel Valiente, *Trading Uninitialized Space for Time*, Information Processing Letters, 92(1):9–13, 2004.

**Solució 3.22:** Si el graf és connex, basta calcular-ne la talla  $m$  i comparar-la amb el seu ordre  $n$ .

```

function ACÍCLIC(V, E)
 $n \leftarrow m \leftarrow 0$
 for all vèrtex $v \in V$ do
 $n \leftarrow n + 1$
 end for
 for all vèrtex $v \in V$ do
 for all vèrtex w adjacent amb el vèrtex v do
 $m \leftarrow m + 1$
 if $m > n$ then
 return false
 end if
 end for
 end for
 return $m = n - 1$

```

**end function**

Si el graf no és connex, cal repetir aquest càlcul sobre cada component connex.

**Solució 3.24:** Durant el recorregut en profunditat d'un graf dirigit i acíclic  $G = (V, E)$ , en acabar de tractar cada vèrtex, se l'afegeix al final d'una seqüència  $L$ , que quedarà doncs ordenada per *finishing time* (també anomenada *numeració inversa del recorregut en profunditat*) dels vèrtexos.

Aquesta seqüència  $L$  és una ordenació topològica inversa de  $G$ .

```

procedure ORDENACIÓINVERSA(G, L)
 for all $v \in G$ do
 $vis[v] \leftarrow false$
 end for
 $L \leftarrow []$
 for all $v \in G$ do
 if $\neg vis[v]$ then
 ORDENACIÓINVERSA-REC(G, v, vis, L)
 end if
 end for
end procedure
procedure ORDENACIÓINVERSA-REC(G, v, vis, L)
 $vis[v] \leftarrow true$
 $\triangleright G[v] =$ llista de succesors del vèrtex v
 for all $w \in G[v]$ do
 if $\neg vis[w]$ then
 ORDENACIÓINVERSA-REC(G, w, vis, L)
 end if
 end for
 $L \leftarrow L.APPEND(v)$
end procedure

```

L'algorisme es pot implementar amb cost  $\Theta(n + m)$  si el graf està representat mitjançant llistes d'adjacència.

**Solució 3.30:** Tot graf  $G = (V, E)$  no dirigit i connex admet un arbre d'expansió  $T = (V, E')$ . Sigui  $v \in V$  una fulla qualssevol de  $T$ , i sigui  $\{v, w\} \in E'$ . Com que  $T'' = (V \setminus \{v\}, E' \setminus \{\{v, w\}\})$  és un arbre,  $T''$  és arbre d'expansió del subgraf  $G'$  de  $G$  induït per  $V \setminus \{v\}$ , per la qual cosa  $G'$  és connex.

En particular, l'última fulla de l'arbre d'expansió que s'obté mitjançant un recorregut en profunditat del graf compleix amb l'enunciat. Així doncs, el cost temporal de l'algorisme és  $\Theta(n + m)$ , on  $n$  és l'ordre i  $m$  la talla del graf.

```

procedure RECORREGUT(G : graf)
 for all vèrtex v de G do
 $visitat[v] \leftarrow false$
 end for

```

```

 $f \leftarrow \text{null}$
 for all vèrtex v de G do
 recorregut(G, v, f)
 end for return f
end procedure
procedure RECORREGUT(G : graf; s, f : vèrtex)
 for all vèrtex v de G do
 visitat[v] \leftarrow false
 end for
 empilar(P, s)
 repeat
 $v \leftarrow \text{cim}(P)$
 desempilar(P)
 if $\neg \text{visitat}[v]$ then
 $f \leftarrow v$
 visitat[v] \leftarrow true
 for all vèrtex w adjacent amb el vèrtex v do
 if $\neg \text{visitat}[w]$ then
 empilar(P, w)
 end if
 end for
 end if
 until buit(P)
end procedure

```

De fet, qualssevol fulla de l'arbre d'expansió que s'obté mitjançant un recorregut en profunditat del graf compleix amb l'enunciat. La primera fulla, per exemple, és el primer vèrtex descobert que té tots els seus vèrtexos adjacents ja descoberts.

**Solució 3.40:**

```

void arrels_i_fulles(const graf& G,
 int& nr_arrels, int& nr_fulles) {

 // Creem un vector amb n = |V| components; inicialment
 // grau_entr[v] == 0 per a tot vertex v, 0 <= v < n
 vector<int> grau_entr(G.size(), 0);

 forall_ver(v, G)
 forall_adj(w, G[v])
 ++grau_entr[*w];

 // aqui grau_entr[v] == grau d'entrada de v, per a tot v

 // calculem quants vertexos tenen grau_entr == 0 (nr_arrels)
 // i quants vertexos no tenen successors (nr_fulles)
 nr_arrels = nr_fulles = 0;
}

```



```

forall_ver(v, G) {
 if (grau_entr[v] == 0) ++nr_arrels;
 if (G[v].empty()) ++nr_fulles;
 // G[v].size() pot tenir cost lineal
 // s'ha d'usar el mètode 'empty' o una alternativa semblant
 // amb cost Theta(1)
}
}

```

El cost de l'algorisme és  $\Theta(n + m)$ , que correspon al bucle central en el qual calculem **grau\_entr** per a tot vèrtex  $v$ . La creïó del vector i el bucle final ténen cost  $\Theta(n)$ . La correctesa és immediata: cada vèrtex  $v$  és predecessor dels seus successors, així doncs cada vèrtex  $v$  contribueix en una unitat al grau d'entrada dels seus successors i això és exactament el que fa el bucle central.

**Solució 3.41:** El nostre algorisme és una aplicació del recorregut en profunditat. La funció **CC\_monocolor** inicialitza un vector de visitats i entra en una iteració per a arrencar recorreguts en profunditat des de cada vèrtex no visitat; cada cop que s'inicia un nou recorregut estarem explorant una nova component connexa. La variable  $ncc$  ens dona l'índex de la component connexa a explorar. En les crides recursives per a fer recorregut d'una CC també passem el color tentatiu que tenen les arestes vistes fins al moment de la CC; inicialment donem a aquest color un valor fictici, doncs encara no s'ha visitat cap aresta.

```

function CC_MONOCOLOR(G, H, CC)
 for all $v \in V(G)$ do
 visitat[v] ← false
 end for
 $ncc \leftarrow 0$; $color \leftarrow -1$
 for all $v \in V(G)$ do
 if ¬visitat[v] then
 $H[ncc] \leftarrow CC_MONOCOLOR_REC(G, v, ncc, CC, color, visitat)$
 $ncc \leftarrow ncc + 1$
 end if
 end for
end function

```

La funció que visita una component connexa, assigna  $ncc$  com a component connexa del vèrtex  $v$  des d'on començem i el marca com a visitat. A continuació, per a tot vèrtex  $w$  adjacent a  $v$  i no visitat, farà el recorregut en profunditat des de  $w$  i ens dirà si totes les arestes visitades són del mateix color, i quin és aquest color si totes fossin del mateix color. Si la aresta  $(v, w)$  no fos del mateix color llavors la CC no és homogènia, es a dir, no totes les seves arestes són del mateix color. Si  $color = -1$  podem assignar a  $color$  el color de la aresta  $(v, w)$ .

```

function CC_MONOCOLOR_REC($G, v, ncc, CC, color, visitat$)
 $CC[v] \leftarrow ncc$
 visitat[v] ← true

```

```

 monocolor ← true
 for all $w \in \text{Adj}(G, v)$ do
 if $\neg \text{visitat}[w]$ then
 monocolor ← CC_MONOCOLOR_REC($G, v, ncc, CC, color, \text{visitat}$)
 if $color = -1$ then
 $color = G.\text{COLOR}(v, w)$
 end if
 monocolor ← monocolor \wedge $color = G.\text{COLOR}(v, w)$
 end if
 end for
 return monocolor
end function

```

Donat que els còmputos associats a la visita de cada vèrtex i aresta del graf són de cost  $\Theta(1)$  el cost d'aquest algorisme és el del recorregut en profunditat bàsic:  $\Theta(n + m)$ , on  $n$  és el número de vèrtexos i  $m$  el número d'arestes.

## Algorismes de dividir i vèncer

**Solució 4.1:** Primero escribiremos una solución recursiva que busque el elemento  $x$  en el subvector  $T[i..j]$ . Si  $i > j$  el subvector es vacío y retornamos -1. Si  $i = j$  entonces el subvector contiene un único elemento y una comparación de éste con  $x$  nos permitirá determinar si  $x$  está presente o no. En el caso general, tenemos  $i < j$  y calcularemos el punto medio del subvector:  $m = \lfloor (i + j)/2 \rfloor$ . Si  $T[m] = x$  entonces  $m$  es la posición buscada. Si  $x < T[m]$  entonces  $x$  se encontrará en  $T[i..m - 1]$  o bien no está en  $T$ ; si  $T[m] < x$  entonces  $x$  se habrá de buscar en  $T[m + 1..j]$ .

```

int localizar(const vector<int>& T, int i, int j, int x) {
 if (i > j) return -1;
 if (i == j) return (T[i] == x) ? i : -1;
 // i > j
 int m = (i + j) / 2;
 if (x < T[m]) return localizar(T, i, m - 1, x);
 else if (x > T[m]) return localizar(T, m + 1, j, x);
 else return m;
}

```

La llamada inicial es `localizar(T, 0, T.size() - 1, x)`. El coste en caso peor del algoritmo viene dado por la recurrencia

$$B(n) = \Theta(1) + B(n/2), \quad n > 1,$$

puesto que la parte no recursiva tiene coste constante ( $\Theta(1)$ ) y sólo hacemos una llamada recursiva (en el caso peor) sobre un subvector que contiene  $n/2$  elementos aproximadamente, siendo  $n$  el número de elementos del vector original. La solución de esta recurrencia es  $B(n) = \Theta(\log n)$ , tal como se pide en el enunciado.

La conversión de este algoritmo a uno iterativo es extremadamente simple, ya que sólo se hace una llamada recursiva y además es lo último que se hace (a esto se le llama *recursividad final*). Básicamente sólo tendremos que “envolver” el cuerpo del algoritmo en un bucle que itera hasta llegar a un caso base:

```
int localizar(const vector<int>& T, int x) {
 int i = 0; int j = T.size() - 1;
 while (i < j) {
 int m = (i + j) / 2;
 if (x < T[m]) j = m - 1;
 else if (x > T[m]) i = m + 1;
 else i = j = m;
 }
 if (i > j) return -1;
 if (i == j) return (T[i] == x) ? i : -1;
}
```

El coste en caso peor sigue siendo  $\Theta(\log n)$  puesto que el bucle realiza  $\Theta(\log n)$  iteraciones en caso peor.

#### Solució 4.2:

- Fragment 1: No funciona si la taula només té una posició i un element que és més gran que el que es busca. Per exemple quan  $T = \langle 1 \rangle$  i  $x = 0$ , es penja.
- Fragment 2: No funciona si la taula només té una posició i un element que és més petit que el que es busca. Per exemple quan  $T = \langle 0 \rangle$  i  $x = 1$ , accedeix a una posició que no existeix. Tampoc funciona quan la taula és buida.
- Fragment 3: No funciona quan quan  $T = \langle 10, 20 \rangle$  i  $x = 30$ , accedeix a una posició que no existeix a la segona iteració.

**Solució 4.3:** La solución de este problema es casi idéntica a la del problema 4. La diferencia está en que cuando encontramos un elemento  $T[m]$  mayor o igual que  $x$  entonces puede haber otras ocurrencias de  $x$  en posiciones inferiores y habremos de hacer una llamada recursiva sobre  $T[i..m]$ .

```
int localizar(const vector<int>& T, int i, int j, int x) {
 if (i > j) return -1;
 if (i == j) return (T[i] == x) ? i : -1;
 // i > j
 int m = (i + j) / 2;
 if (x <= T[m]) return localizar(T, i, m, x);
 else return localizar(T, m + 1, j, x);
}
```

**Solució 4.4:** Modificaremos levemente el algoritmo del apartado anterior para que nos devuelva la menor posición  $p$  tal que  $T[p] \geq x$ . De forma similar, haremos otra variante

del algoritmo que nos devuelva la mayor posición  $q$  tal que  $T[q] \leq y$ . El coste de cada búsqueda es  $\Theta(\log n)$ . El número de elementos pedido es  $q - p + 1$  y el coste total del algoritmo es  $\Theta(\log n)$ .

**Solució 4.5:** Para que la búsqueda ternaria pueda funcionar el vector debe estar ordenado. Supondremos que está ordenado crecientemente. Nuestro algoritmo recursivo busca un elemento dado  $x$  en el subvector de  $v$  delimitado por los índices  $i$  y  $j$ . Es decir, en todo momento mantendremos como invariante que  $v[i] \leq x < v[j]$ .

```
// Pre: $v[i] \leq x < v[j]$, v ordenado creciente
// Post: $k = \text{busqueda_ternaria}(v, i, j, x) \implies v[k] \leq x < v[k+1]$
template <typename Elem>
int busqueda_ternaria(const vector<Elem>& v, int i, int j,
 const Elem& x) {
 int n = j - i + 1;
 if (n < 3) return caso_base(v, i, j, x);

 // n >= 3
 int d = n / 3;
 if (x < v[i + d]) return busqueda_ternaria(v, i, i+d, x);
 if (x < v[j - d]) return busqueda_ternaria(v, i+d, j-d, x);
 return busqueda_ternaria(v, j-d, j, x);
}

// Pre: $j - i + 1 \leq 2$, $v[i] \leq x < v[j]$
template <typename Elem>
int caso_base(const vector<Elem>& v, int i, int j,
 const Elem& x) {
 // Caso especial: x es mayor que cualquier elemento de v
 if (v[v.size() - 1] <= x) return v.size() - 1;

 int k = i;
 while (k < j && v[k+1] <= x)
 ++k;
 return k;
}
```

La llamada inicial es  $k = \text{busqueda\_ternaria}(v, -1, v.size(), x)$ , y podemos asumir que hay dos elementos ficticios  $v[-1] = -\infty$  y  $v[v.size()] = +\infty$  para que el invariante se cumpla desde el principio. Después de la llamada, si  $k = -1$  entonces  $x$  es menor que cualquier elemento del vector y si lo insertáramos debería ser el primero; si  $k \geq 0$  entonces comparamos  $v[k]$  con  $x$ . Si son iguales entonces  $x$  está en  $v$ , en la posición  $k$ ; si son diferentes entonces  $x$  no está en  $v$  y si lo insertáramos deberíamos hacerlo en la posición  $k + 1$ .

El coste de este algoritmo satisface la siguiente recurrencia:

$$T(n) = T(n/3) + \Theta(1), \quad n \geq 3,$$

de manera que aplicando el *master theorem* para recurrencias de divide y vencerás obtenemos que  $T(n) = \Theta(\log n)$ . Es decir, asintóticamente idéntico al coste de la búsqueda

binaria. Si bien el número de llamadas recursivas que efectúa la búsqueda ternaria es menor ( $\approx \log_3 n$ , frente a las  $\approx \log_2 n$  de la búsqueda binaria), el coste de cada llamada recursiva es mayor (se hace una comparación más que en la búsqueda binaria en cada llamada recursiva), por lo que no resulta más interesante que la búsqueda binaria.

**Solució 4.6:** No. La búsqueda binaria requiere disponer de acceso directo en tiempo constante, es decir, poder consultar el elemento  $i$ -ésimo, dado  $i$ , en tiempo constante. Esto no es posible en una lista, donde el acceso es puramente secuencial.

**Solució 4.8:** El algoritmo es estable si utilizamos un algoritmo de fusión de listas que lo sea. Es decir, cuando el algoritmo de fusión de listas examina dos valores idénticos en las listas a fusionar debe preservar el orden relativo, colocando en la lista resultado el valor proveniente de la primera lista, no el de la segunda lista.

**Solució 4.9:** El coste del algoritmo de ordenación por fusión (*mergesort*) no depende de la entrada particular, ya que actúa siempre igual. Divide la lista original en dos partes aproximadamente de igual longitud (con coste  $\Theta(n)$ ), ordena cada una de ellas separadamente, y las fusiona; el coste de la fusión también es  $\Theta(n)$ , independientemente de cómo se “entrelazan” las dos listas dadas. Por todo ello el coste es  $\Theta(n \log n)$  en cualquier caso.

**Solució 4.10:** La profundidad máxima que alcanzará la pila es igual a la altura del árbol de recursión, es decir, la distancia máxima entre la raíz (llamada inicial) y una hoja (casos de base). Esto se aplica en genral a todo algoritmo recursivo, no sólo a *mergesort*. Puesto que en *mergesort* cada llamada recursiva divide por 2 (aproximadamente) el tamaño de la lista a ordenar, el árbol de recursión tiene altura  $\Theta(\log n)$ .

**Solució 4.15:** No, no lo es. En una etapa de partición dada, podemos intercambiar un elemento menor que el pivote, digamos  $a$ , con uno mayor que el pivote de tal modo que dicho elemento  $a$  quede por delante de uno igual a él, y esto no necesariamente se “corrije” en las siguientes etapas. De modo que ya no es cierto que en caso de empate se respete el orden original de aparición en el vector. Por ejemplo, si tenemos el vector  $v = [6, 8, 4, 4', 7]$ , tomamos como pivote al 6 y la partición intercambiará el segundo 4 (lo hemos marcado para distinguirlo) con el 7. El vector quedará así:  $v = [4', 4, 6, 7]$ . Las siguientes llamadas recursivas no modifican nada.

**Solució 4.16:** En el primer caso, el coste en caso promedio será  $\Theta(n \log n)$ , pues el elemento inicial del vector será el  $j$ -ésimo con probabilidad  $1/n$ . En el segundo caso, el pivote elegido es el menor elemento del vector y tras la partición (con coste  $\Theta(n)$ ) sólo haremos una llamada recursiva sobre el subvector de la derecha que contiene  $n - 1$  elementos. El coste resultante es  $\Theta(n^2)$ . Pasa otro tanto en el tercer caso, ya que el pivote elegido es el mayor elemento del vector y sólo haremos una llamada recursiva sobre el subvector de la izquierda que contiene  $n - 1$  elementos. Finalmente, en el cuarto caso, la partición “barrerá” el vector haciendo  $\approx n/2$  intercambios inútiles, pero dejará el pivote en la posición media del vector, de manera que el coste será  $\Theta(n \log n)$ . Una modificación

astuta del algoritmo de partición que haga **tres** partes (elementos menores, elementos iguales, elementos mayores que el pivote) tendría coste  $\Theta(n)$  en este caso, puesto que el algoritmo no hace una llamada recursiva sobre la parte de los elementos iguales y las otras dos partes son vacías en el caso de que todos los elementos son iguales.

**Solució 4.18:** Una solució consisteix a dividir la taula en dues meitats, vèncer comptant recursivament el nombre d'inversions en cada meitat per separat, i combinar comptant el nombre d'inversions entre elements de meitats diferents, retornant la suma del tres nombres d'inversions. Òbviament,  $T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2)$ , per la qual cosa cal trobar un algorisme de combinació més eficient.

El nombre d'inversions entre elements de meitats diferents es pot comptar fàcilment en temps  $\Theta(n)$  una vegada hem ordenat cadascuna de les dues meitats, amb la qual cosa tenim que  $T(n) = 2T(n/2) + \Theta(n \log n) = \Theta(n \log^2 n)$ .

Suposant, però, que les dues meitats es troben ordenades, podem comptar en temps  $\Theta(n)$  el nombre d'inversions entre elements de meitats diferents mentre fusionem les dues meitats en una taula ordenada, també en temps  $\Theta(n)$ , amb la qual cosa tenim que  $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ .

```

procedure COMPTARINVERSIONS(A, num_inv)
 $num_inv \leftarrow 0$
 COMPTARINVERSIONS($A, 0, A.SIZE() - 1, num_inv$)
end procedure
procedure COMPTARINVERSIONS(A, i, j, num_inv)
 if $i < j$ then
 $m \leftarrow (i + j)/2$;
 COMPTARINVERSIONS(A, i, m, num_inv)
 COMPTARINVERSIONS($A, m + 1, j, num_inv$)
 FUSIONACOMPTANT($A, i, m, m + 1, j, num_inv$)
 end if
end procedure
procedure FUSIONACOMPTANT(A, i, i', j, j', num_inv)
 $k \leftarrow 0$
 while $i \leq i' \wedge j \leq j'$ do
 if $A[i] \leq A[j]$ then
 $B[k] \leftarrow A[i]; i \leftarrow i + 1$
 else
 $B[k] \leftarrow A[j]; j \leftarrow j + 1$
 $num_inv \leftarrow num_inv + i' - i + 1$
 end if
 $k \leftarrow k + 1$
 end while
 while $i \leq i'$ do
 $B[k] \leftarrow A[i]; i \leftarrow i + 1; k \leftarrow k + 1$
 end while

```

```

while $j \leq j'$ do
 $B[k] \leftarrow A[j]; j \leftarrow j + 1; k \leftarrow k + 1$
end while
for $k \leftarrow 0$ to $j - i$ do
 $A[i + k] \leftarrow B[k]$
end for
end procedure

```

**Solució 4.19:** Para resolver la primera cuestión, ordenaremos en primer lugar el vector  $t$  crecientemente, y mediante un bucle que lo recorre una vez ordenado, obtenemos sus sumas parciales:

```

sort(t.begin(), t.end());
vector<int> sumparc(t.size() + 1);
sumparc[0] = 0;
for (int i = 1; i < sumparc.size(); ++i)
 sumparc[i] = sumparc[i-1] + t[i-1];

```

Llamemos  $t'$  al vector  $t$  una vez ordenado. Entonces

$$\text{sumparc}[i] = \sum_{j < i} t'[j] = \sum_{t[j] < t'[i]} t[j].$$

Para encontrar la respuesta que nos piden bastará recorrer el vector de sumas parciales y contabilizar cada elemento tal que  $\text{sumparc}[i] < t'[i]$ . El algoritmo queda así:

```

int contar(vector<int>& t) {
 sort(t.begin(), t.end());
 vector<int> sumparc(t.size() + 1);
 sumparc[0] = 0;
 for (int i = 1; i < sumparc.size(); ++i)
 sumparc[i] = sumparc[i-1] + t[i-1];

 int cont = 0;
 for (int i = 0; i < t.size(); ++i)
 if (sumparc[i] < t[i]) ++cont;
 return cont;
}

```

El coste del algoritmo es  $\mathcal{O}(n \log n)$ , que corresponde al coste de la ordenación; los dos bucles posteriores tiene coste  $\Theta(n)$ .

Una solución alternativa combina todos estos pasos y consiste en una variación de quicksort en la que la partición calcula la suma de los elementos del subvector a la izquierda del pivote, y si dicha suma es menor que el pivote entonces se incrementa el contador de elementos del conjunto. El procedimiento recibe adicionalmente la suma de los elementos por debajo del subvector  $A[i..j]$  procesado.

```

// sumprec = sum(a[k], k=0..i-1
// 0 <= i <= j < n = a.size()

```

```

int contar(const vector<int>& a, int i, int j,
 int sumprec) {
 if (i == j) {
 if (sumprec < a[i]) return 1;
 else return 0;
 } else { // i < j
 int k; int sumparc = 0;
 // reorganiza a de modo que
 // a[i..k-1] < a[k] < a[k+1..j]
 // y sumparc == sum(a[r], r = i..k-1)
 particionar(a, i, j, k, sumparc);
 int contizq = contar(a, i, k-1, sumprec);
 int contder = contar(a, k+1, j, sumprec + sumparc + a[k]);
 if (sumprec + sumparc < a[k])
 return contizq + contder + 1;
 else
 return contizq + contder;
 }
}

```

**Solució 4.23:** Siguin  $N$  i  $M$  les longituds respectives de les taules  $A$  i  $B$ . Comparem el punt mig  $x$  de la taula  $A$  amb el punt mig  $y$  de la taula  $B$ .

Si  $x > y$  llavors  $A[N/2..N]$  conté elements més grans que els elements dels subvectors  $A[1..N/2]$  i  $B[1..M/2]$ . Això vol dir que els elements d' $A[N/2..N]$  queden per sobre de la mitjana de la “taula” combinada (ténen més de  $N/2 + M/2 = (N + M)/2$  per sota), i  $B[1..M/2]$  queda per sota de la mitjana de l'array combinat (els elements d'aquest subvector ténen més de  $(M + N)/2$  elements per sobre). Si  $k < (M + N)/2$  (es a dir, queda per sota de la mitjana de l'array combinat) llavors podem descartar  $A[N/2..N]$ . Si  $k > (M + N)/2$  podem descartar  $B[1..M/2]$ . S'aplica un raonament similar si  $x \leq y$ .

A cada pas,  $M$  o  $N$  es redueixen a la meitat. Per tant, el cost de l'algorisme és  $\Theta(\log M + \log N)$ . Inicialment  $M = N = n$ , amb la qual cosa arribem a la conclusió de que el cost de l'algorisme és  $\Theta(\log n)$ .

**procedure** FINDKTHGLOBAL( $A, B, i, j, i', j', k$ )

**if**  $j \leq i \vee j' \leq i'$  **then**

    Resolem el problema trivialment

**else**

    ▷ El  $k$ -èssim global es troba a  $A[i..j] + B[i'..j']$

$m \leftarrow (i + j)/2; m' \leftarrow (i' + j')/2$

**if**  $x > y$  **then**

**if**  $k \geq j' + j - m' - m + 1$  **then**

        ▷ L'element buscat no pot estar a  $B[i'..m']$

        FINDKTHGLOBAL( $A, B, i, j, m' + 1, j', k - j' - j + m' + m$ )

**else**

        ▷ L'element buscat no pot estar a  $A[m + 1..j]$

        FINDKTHGLOBAL( $A, B, i, m, i', j', k$ )

**end if**



```

else
 if $k \geq j' + j - m' - m + 2$ then
 ▷ L'element buscat no pot estar a $A[i..m-1]$
 FINDKTHGLOBAL($A, B, i, m-1, i', j', k-j'-j+m'+m-1$)
 else
 ▷ L'element buscat no pot estar a $B[m'..j']$
 FINDKTHGLOBAL($A, B, i, m, i', m'-1, k$)
 end if
end if
end if
end procedure

```

**Solució 4.26:** Supongamos que sabemos que el índice buscado se encuentra, en todo caso, en el subvector  $a[i..j]$ . Para fijar ideas, haremos que el algoritmo retorne -1 si no existe el índice buscado.

Si  $i = j$  entonces el subvector contiene un único elemento, y entonces bastará comprobar si  $a[i] = i$  o no, y retornar  $i$  o -1 según el caso.

Si  $i < j$  hay más de un elemento y entonces examinamos el punto medio  $a[m]$  del subvector, con  $m = (i + j)/2$ . Si  $a[m] = m$  entonces  $m$  es un índice que cumple la condición buscada. Si  $a[m] < m$  entonces, como todos los elementos son distintos y el vector está en orden creciente, ningún índice  $k$  a la izquierda, es decir,  $i \leq k < m$ , puede cumplir que  $a[k] = k$ . En efecto, si  $a[k] = k$  entonces  $a[k+1] \geq k+1$ ,  $a[k+2] \geq k+2$ , ... y  $a[m] \geq m$ , lo que está en contradicción con el supuesto de que  $a[m] < m$ . Por lo tanto si  $a[m] < m$  podemos proseguir la búsqueda en  $a[m+1..j]$ , descartando  $a[i..m]$ . Análogamente, si  $a[m] > m$  entonces se descarta el subvector de la derecha y seguimos la búsqueda recursivamente en  $a[i..m-1]$ .

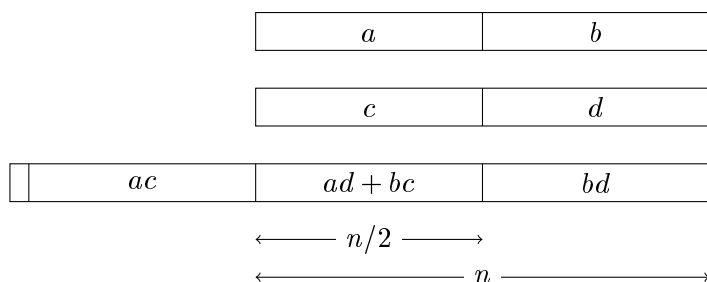
```

int punto_fijo(const vector<int>& a, int i, int j) {
 if (i == j) return ((a[i] == i) ? i : -1);
 // i < j
 int m = (i + j) / 2;
 if (a[m] == m)
 return m;
 else if (a[m] < m)
 return punto_fijo(a, m+1, j);
 else // a[m] > m
 return punto_fijo(a, i, m-1);
}

```

El coste del algoritmo es  $\Theta(\log n)$  pues funciona de manera idéntica a una búsqueda dicotómica. Podemos llegar a la misma conclusión resolviendo la recurrencia satisfecha por el coste  $C(n)$  del algoritmo. Puesto que sólo se hace una llamada recursiva en caso peor con la mitad de los elementos del subvector original,  $C(n) = \Theta(1) + C(n/2)$ , cuya solución es  $C(n) = \Theta(\log n)$ .

**Solució 4.27:** El següent diagrama mostra les dades d'entrada  $x$  i  $y$  i el seu resultat, i com es poden descomposar:

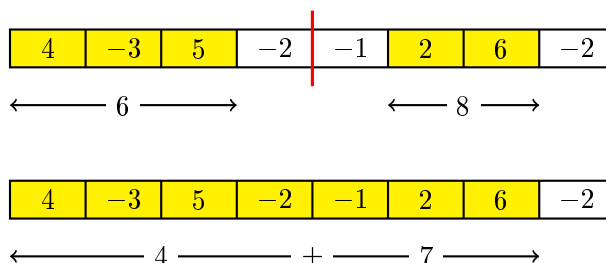


- a. Cal fer 4 crides recursives per a calcular els productes  $a \cdot c$ ,  $b \cdot d$ ,  $a \cdot d$  i  $b \cdot c$ . A més s'hauran de fer decalatges per a multiplicar per  $2^n$  i  $2^{n/2}$ , i la suma  $a \cdot d + b \cdot c$ . Aquestes operacions tenen totes cost  $\Theta(n)$ . Així doncs el cost de l'algorisme de multiplicació satisfà:  $T(n) = 4T(n/2) + \Theta(n)$ , i la solució és  $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$ .
- b. Podem estalviar una crida recursiva fent més operacions de suma (o resta) a la part no recursiva. Només caldrà calcular els productes  $U = a \cdot c$ ,  $V = b \cdot d$  i  $W = (a + b) \cdot (c + d)$ . Llavors  $ad + bc = (a + b)(c + d) - ac - bd = W - U - V$ . El cost ve donat per la recurrència:  $T(n) = 3T(n/2) + \Theta(n)$ , la solució de la qual és  $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$ .
- c. Omplint de zeros a l'esquerra fins arribar a una potència de 2. Només passem de  $n$  a menys de  $2n$  bits.

#### Solució 4.28:

- a) Aquest algorisme requereix 7 crides recursives al producte de matrius i 17 sumes/restes de matrius. La part no recursiva de l'algorisme té cost  $\Theta(n^2)$ ; llavors el cost de l'algorisme de Strassen és  $M(n) = \Theta(n^2) + 7M(n/2)$ . Donat que  $\alpha = \log_2 7 \approx 2.807 > 2$ , obtenim que  $M(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$ .
- b) Igual que a l'exercici anterior és suficient afegir files i columnes amb zeros fins que les noves dimensions  $n' \times n'$  siguin potència de 2. Com que  $n \leq n' < 2n$ , el cost de l'algorisme seguirà sent del mateix ordre:  $M(n) = \Theta((n')^{\log_2 7}) = \Theta(n^{\log_2 7})$ .
- c) Tot algorisme per multiplicar dues matrius  $n \times n$  té cost  $\Omega(n^2)$ , doncs cal donar valor a  $n^2$  elements de la matriu resultat.

**Solució 4.29:** Damos en primer lugar, en forma muy compacta, una solución basada en divide-y-vencerás que tiene coste  $\Theta(n \log n)$ :



```

1: function MAXSUM(A, ℓ, r)
2: if $\ell > r$ then
3: return 0
4: else if $\ell = r$ then
5: return ($A[\ell] > 0$)? $A[\ell]$: 0
6: else
7: $m := \lfloor (\ell + r)/2 \rfloor$
8: $lmax := 0$
9: $sum := 0$
10: for $i := m$ downto ℓ do
11: $sum := sum + A[i]$
12: if $sum > lmax$ then $lmax := sum$
13: end if
14: end for
15: $rmax := 0$
16: $sum := 0$
17: for $i := m + 1$ to r do
18: $sum := sum + A[i]$
19: if $sum > rmax$ then $rmax := sum$
20: end if
21: end for
22: return $\max(\text{maxsum}(\ell, m), \text{maxsum}(m + 1, r), \quad lmax + rmax)$
23: end if
24: end function

```

Vamos a ver a continuación una serie de soluciones, progresivamente más eficientes, con explicaciones detalladas sobre su funcionamiento y coste.

Definimos  $\sigma_{i,j} = \sum_{i \leq k \leq j} A[k]$ . Buscamos pues  $\sigma = \max(\sigma_{i,j} \mid 0 \leq i \leq j < n)$ .

Una primera solución ingenua tiene coste  $\Theta(n^3)$ :

```

imax = 0; jmax = -1; sigma = 0;
for (int i = 0; i < n; ++i)
 for (int j = i; j < n; ++j) {
 int sum = 0;
 for (int k = i; k <= j; ++k)
 sum += A[k];
 // sum = $\sigma_{i,j}$
 if (sum > sigma) {
 imax = i; jmax = j; sigma = sum;
 }
 }

```

Se puede obtener una mejora sustancial sobre la solución previa observando que

$$\sigma_{i,j+1} = \sigma_{i,j} + A[j+1]$$

```

imax = 0; jmax = -1; sigma = 0;
for (int i = 0; i < n; ++i) {
 int sum = 0;
 for (int j = i; j < n; ++j) {
 // sum = $\sigma_{i,j-1}$
 sum += A[j];
 // sum = $\sigma_{i,j}$
 if (sum > sigma) {
 imax = i; jmax = j; sigma = sum;
 }
 }
}

```

Esta nueva solución tiene coste  $\Theta(n^2)$ . Usando divide y vencerás podemos obtener la solución de un modo más eficiente. Supongamos que dividimos el segmento en curso  $A[\ell..u]$  (con más de un elemento, en otro caso podemos dar la solución directa) en dos mitades (o casi) y resolvemos el problema de manera independiente para cada una de ellas. El segmento de suma máxima en  $A[\ell..u]$  será uno de los dos segmentos obtenidos recursivamente **o bien** un segmento no vacío que contiene elementos de las dos mitades en que se dividió  $A[\ell..u]$ , “atravesando” el punto de corte.

Para determinar el segmento de suma máxima en  $A[\ell..u]$ , siendo el punto de corte  $m$ ,  $\ell \leq m < u$ , hemos de averiguar cuál es el segmento de la forma  $A[r..m]$ , es decir, que necesariamente acaba en la posición  $m$ , de suma máxima en  $A[\ell..m]$ . Sea  $s_1$  su suma y  $\ell'$  su extremo inferior (su otro extremo es, por definición,  $m$ ).

Análogamente, deberemos averiguar cuál es el segmento de la forma  $A[m+1..s]$  de suma máxima en  $A[m+1..u]$ , siendo su extremo superior  $u'$  y suma  $s_2$ .

Entonces el segmento de suma máxima en  $A[\ell..u]$  es el de mayor suma entre el segmento de suma máxima en  $A[\ell..m]$ , el segmento de suma máxima en  $A[m+1..u]$ , y el segmento  $A[\ell'..u']$  (su suma es  $s_1 + s_2$ ).

Tanto  $s_1$  y  $s_2$  como sus extremos se pueden obtener mediante dos sencillos bucles, pues  $A[\ell'..m]$  tiene suma máxima entre todos los segmentos de la forma  $A[r..m]$ ,  $\ell \leq r \leq m$ , y  $A[m+1..u']$  tiene suma máxima entre todos los que son de la forma  $A[m+1..s]$ ,  $m+1 \leq s \leq u$ .

```

void ssm(const vector<T>& A, int l, int u,
 int& imax, int& jmax, int& summax) {

 // el segmento A[l..u] contiene 0 ó 1 elemento
 if (u - l + 1 <= 1) {
 if (u - l + 1 == 0) {
 imax = l; jmax = u; summax = 0;
 } else if (A[l] >= 0) {
 imax = l; jmax = u; summax = A[l];
 } else {
 imax = l; jmax = l - 1; summax = 0;
 }
 return;
 }
}

```

```

// el segmento A[l..u] contiene 2 ó más elementos
int m = (l + u) / 2;
int imax1, jmax1, imax2, jmax2, summax1, summax2;
ssm(A, l, m, imax1, jmax1, summax1);
ssm(A, m + 1, u, imax2, jmax2, summax2);

int lprime = m; int s1 = 0;
for (int sum = 0, int r = m; r >= l; --r) {
 sum += A[r];
 if (sum > s1) { lprime = r; s1 = sum; }
}

int uprime = m; int s2 = 0;
for (int sum = 0, int s = m + 1; s <= u; ++s) {
 sum += A[s];
 if (sum > s2) { uprime = s; s2 = sum; }
}

summax = max(summax1, summax2, s1 + s2);
imax = ...; // imax1, imax2 ó lprime
jmax = ...; // jmax1, jmax2 ó uprime
}

```

Puesto que el coste de dividir y combinar es  $\Theta(n)$ , el coste del algoritmo divide y vencerás es

$$S(n) = \Theta(n) + 2 \cdot S(n/2),$$

es decir,  $S(n) = \Theta(n \log n)$ .

Pero podemos mejorar la eficiencia hasta conseguir que sea óptima, utilizando una *inmersión de eficiencia*. En concreto, la nueva versión de **ssm**, aplicada a un segmento  $A[l..u]$  retornará:

1. Los extremos del segmento de suma máxima *imax* y *jmax* y su valor *summax*;
2. El extremo *ider* del segmento de suma máxima de entre aquellos que incluyen al extremo superior *u*, y su suma *sder*;
3. El extremo *jizq* del segmento de suma máxima de entre aquellos que incluyen al extremo inferior *l*, y su suma *sizq*;
4. La suma *sum* de todos los elementos del segmento.

Si efectuamos las llamadas

```

m = (l + u) / 2;
ssm(A, l, m, imax1, jmax1, summax1,
 ider1, sder1, jizq1, sizq1, sum1);
ssm(A, m+1, u, imax2, jmax2, summax2,
 ider2, sder2, jizq2, sizq2, sum2);

```

podemos calcular los valores correspondientes a  $A[l..u]$  teniendo en cuenta lo siguiente:

- El segmento de suma máxima será el correspondiente a

$$\max(\text{summax1}, \text{summax2}, \text{sder1} + \text{sizq2})$$

- El segmento de suma máxima con extremo superior en  $u$  será el correspondiente a

$$\max(\text{sder2}, \text{sder1} + \text{sum2})$$

- El segmento de suma máxima con extremo inferior en  $\ell$  será el correspondiente a

$$\max(\text{sizq1}, \text{sum1} + \text{sizq2})$$

- La suma del segmento es  $\text{sum1} + \text{sum2}$ .

Los valores de los extremos  $\text{imax}$ ,  $\text{jmax}$ ,  $\text{ider}$  y  $\text{jizq}$  se calculan en consonancia: p.e. si  $\max(\text{sizq1}, \text{sum1} + \text{sizq2})$  es  $\text{sizq1}$  entonces  $\text{jizq} = \text{jizq1}$ , sino  $\max(\text{sizq1}, \text{sum1} + \text{sizq2}) = \text{sum1} + \text{sizq2}$  y por tanto  $\text{jizq} = \text{jizq2}$ .

Gracias a la inmersión de eficiencia, el coste no recursivo es ahora  $\Theta(1)$  y ya que

$$S(n) = \Theta(1) + 2 \cdot S(n/2),$$

el coste del nuevo algoritmo es  $S(n) = \Theta(n)$  (usar el caso 3 del teorema maestro).

**Solució 4.35:** Existen varias formas de solucionar este problema, pero todas ellas tienen al menos coste  $\Omega(n^2)$  pues tenemos como mínimo que dar valor a  $(n^2 - n)/2$  componentes de la matriz resultado. Una posible solución usa el esquema de divide-y-vencerás para resolver el problema. La solución propuesta usa una función auxiliar, también basada en divide-y-vencerás, para organizar un torneo entre los jugadores de dos conjuntos  $A$  y  $B$ . Cada conjunto tiene  $m$  jugadores y cada jugador juega un partido por día contra cada uno de los jugadores del otro conjunto; los jugadores de un conjunto no juegan partidos entre ellos y el torneo se completa en  $m$  días.

```
// Organizamos un torneo para el subconjunto de jugadores
// {i, ..., j} comenzando el día d; al menos tiene que haber
// dos jugadores; i <= j-1; el torneo dura n - 1 días si el
// subconjunto contiene n = j - i + 1 jugadores
//
// Llamada inicial: torneo(M, 1, n, 1);
//
void torneo(int M[][], int i, int j, int d) {
 if (j - i == 1) // solo hay dos jugadores
 M[i][j] = d;
 else {
 // dividimos el subconjunto {i, ..., j} en dos
 // subconjuntos de igual talla; para cada uno de los subconjuntos
 // organizamos un torneo independiente y en paralelo,
 // y luego hacemos que
 // jueguen partidos entre los jugadores de uno y otro subconjunto
 int m = (i + j) / 2;
```

```

 torneo(M, i, m, d);
 torneo(M, m+1, j, d);
 // estos torneos acaban el dia d + m - i - 1
 // hacemos los cruces comenzando el dia d + m - i
 cruces(M, i, m, m+1, j, d + m - i);
}
}

// Organizamos un torneo entre los conjuntos de jugadores
// {i, ..., j} y {a, ..., b} de igual talla (j - i + 1 == b - a + 1),
// comenzando el dia d; al menos tiene que haber
// un jugador en cada conjunto
// y el torneo resultante dura tantos dias como jugadores hay en
// un conjunto (los dos conjuntos tienen igual talla)

void cruces(int M[][], int i, int j, int a, int b, int d) {
 if (i == j) // solo hay un jugador en cada conjunto
 M[i][a] = d;
 else {
 // dividimos cada conjunto A y B en dos partes, obteniendo
 // cuatro subconjuntos de jugadores: A1, A2, B1, B2;
 // hacemos los cruces A1 - B1, A2 - B2 en paralelo y luego
 // los cruces A1 - B2 y A2 - B1 en paralelo
 int m = (i + j) / 2;
 int c = (a + b) / 2;
 cruces(M, i, m, a, c, d);
 cruces(M, m+1, j, c+1, b, d);
 cruces(M, i, m, c+1, b, d + (m - i) + 1);
 cruces(M, m+1, j, a, c, d + (m - i) + 1);
 }
}

```

Sea  $C(n)$  el coste del algoritmo que organiza los cruces entre dos conjuntos de  $n$  jugadores cada uno. Entonces

$$C(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 4C(n/2) + \Theta(1) & \text{si } n > 1. \end{cases}$$

La solución de esta recurrencia, que obtenemos aplicando el *master theorem*, es  $C(n) = \Theta(n^2)$  ( $a = 4, b = 2, k = 0, \alpha = \log_b a = 2 > k = 0$ ). El coste del algoritmo que organiza el torneo para un conjunto de  $n$  jugadores viene dado por la siguiente recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 2, \\ 2T(n/2) + C(n/2) + \Theta(1) & \text{si } n > 2. \end{cases}$$

La solución es  $T(n) = \Theta(n^2)$  ya que el coste no recursivo es  $C(n/2) + \Theta(1) = \Theta(n^2)$  ( $a = 2, b = 2, k = 2, \alpha = \log_b a = 1 < k = 2$ ).

**Solució 4.39:** L'algorisme que proposem és molt similar al de cerca binaria. Si el vector té tres elements llavors el pic és el màxim dels tres elements. Altrament, trobem el punt

intermig  $a_m$ . Si  $a_{m-1} < a_m$  llavors el pic necessàriament es troba en la subseqüència  $(a_m, a_{m+1}, \dots)$  a la dreta; si  $a_{m-1} > a_m$  llavors el pic es troba necessàriament a la subseqüència  $(a_1, \dots, a_{m-1})$ . El cas  $a_m = a_{m-1}$  no és possible per la definició del que és una seqüència unimodal. L'algorisme queda així:

```
// el vector té 3 ó més elements: i + 2 <= j

template <typename T>
T pic_unimodal(const vector<T>& A, int i, int j) {
 // maxim(a, b, c) retorna el més gran dels tres valors
 if (i+2 == j) return maxim(A[i], A[i+1], A[i+2]);

 int m = (i + j) / 2;
 if (A[m-1] < A[m])
 return pic_unimodal(A, m, j);
 else
 return pic_unimodal(A, i, m-1);
}
```

El cost satisfà la recurrència  $U(n) = \Theta(1) + U(n/2)$ , i la solució és  $U(n) = \Theta(\log n)$ .

## Programació dinàmica

**Solució 5.1:** Dada la definició recursiva anterior es immediato implementar una funció recursiva que calcule el valor de  $F(n)$  para  $n$  dada.

```
int Fib(int n) {
 if (n <= 1) return n;
 return Fib(n-2) + Fib(n-1);
}
```

Sin embargo, si analizamos el coste de la función anterior podemos observar fácilmente que es exponencial con respecto a  $n$ . Más específicamente, es  $\Theta(\phi^n)$ , donde  $\phi = (1 + \sqrt{5})/2 \approx 1.618\dots$ ; es sumamente ineficiente.

También es fácil observar en la implementación anterior que se repite el cálculo de subproblemas idénticos y que se puede mejorar la eficiencia del algoritmo guardando los valores ya calculados. Para ello podemos utilizar un vector que nos permita memorizar los resultados.

```
class Fibonacci {
 vector<int> V;

public:

 Fibonacci() {
 V.push_back(0);
 V.push_back(1);
 }
};
```



```

 }

 int operator()(int n) {
 int m = V.size();
 for (int i = m; i <= n; ++i)
 V.push_back(V[i-2] + V[i-1]);
 return V[n];
 }
};

int main() {
 Fibonacci Fib;
 ...
 long Fn = Fib(n);
 ...
}

```

Procediendo de esta manera obtenemos un algoritmo de coste  $\mathcal{O}(n)$  que requiere una cantidad de memoria  $\mathcal{O}(n)$ . Si ahora aplicamos la técnica de inmersión a la implementación anterior obtenemos una función que no requiere este espacio y tiene un coste  $\mathcal{O}(n)$ .

```

// Post: a = F(n), b = F(n+1)
void Fib(int n, int &a, int &b) {
 if (n == 0) {
 a = 0;
 b = 1;
 } else if (n == 1) a = b = 1;
 else {
 Fib(n-2, a, b);
 a = a + b;
 b = a + b;
 }
}

```

**Solució 5.2:** Los números combinatorios (o coeficientes binomiales)  $\binom{n}{k}$  se definen como el número de maneras que hay de escoger  $k$  elementos de un conjunto de  $n$ . Por tanto tenemos que:

$$\binom{n}{k} = \begin{cases} 0 & k < 0 \text{ o } k > n \\ 1 & n = 0 \\ \frac{n!}{(n-k)!k!} & \end{cases}$$

Es fácil demostrar a partir de la definición anterior que los números combinatorios cumplen con la siguiente igualdad:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Y a partir de esta igualdad es inmediato implementar una función recursiva que calcule  $\binom{n}{k}$ .

```
int Bin(int n, int k) {
 if (k < 0 or k > n) return 0;
 if (n == 0) return 1;
 return Bin(n-1, k) + Bin(n-1, k-1);
}
```

Sin embargo, el coste de esta función es exponencial ya que se hacen dos llamadas recursivas para resolver subproblemas idénticos de manera independiente. Esto sugiere guardar las soluciones de los problemas ya resueltos para no tener que recalcularlos en la segunda llamada recursiva. Como estructura de datos para guardar estos valores utilizaremos una matriz triangular inferior, conocida como *Triángulo de Tartaglia* o *Triángulo de Pascal*. Una posible implementación es la siguiente.

```
class Binomial {

 vector<vector<int> > M;

 int consulta (int i, int j) {
 return j < 0 or j > i ? 0 : M[i][j];
 }

public:

 Binomial() {
 M = vector<vector<int> >(1, vector<int>(1, 1));
 }

 int operator() (int n, int k) {
 int m = M.size();
 for (int i = m; i <= n; ++i) {
 M.push_back(vector<int>(i+1));
 for (int j = 0; j <= i; ++j)
 M[i][j] = consulta(i-1, j) + consulta(i-1, j-1);
 }
 return consulta(n, k);
 }
};

int main() {
 Binomial Bin;
 int n, k;
 ...
 long bin_nk = Bin(n, k);
 ...
}
```

Ahora el coste  $C(n)$  del algoritmo anterior para calcular  $\binom{n}{k}$  es el coste de rellenar

la matriz, es decir  $\Theta(n^2)$ , lo que mejora en mucho el coste exponencial de la primera función.

Una implementación recursiva eficiente para el cálculo de números combinatorios se basa en la siguiente propiedad.

$$\binom{n}{k} = \binom{n}{k-1} \frac{n-k+1}{k}$$

```
int Bin(int n, int k) {
 if (k < 0 or k > n) return 0;
 if (k == 0) return 1;
 return Bin(n, k-1) * (n-k+1) / k;
}
```

El algoritmo recursivo resultante tiene coste  $\mathcal{O}(n)$ , lo que incluso mejora el algoritmo iterativo anterior. Sin embargo, en aplicaciones en las que no hay restricciones de memoria y se requiere calcular de manera muy frecuente números combinatorios para  $ns$  y  $ks$  variadas (más de  $n$  veces) puede resultar más eficiente tener la matriz guardada y acceder a cada una de sus posiciones en tiempo constante.

**Solució 5.8:** Sea  $C(i, w)$  el número de formas diferentes en que podemos conseguir el valor  $w \geq 0$  usando sólo los primeros  $i$  tipos de sellos. Entonces la respuesta buscada es  $C(n, V)$ . Tomaremos como base de la recursión  $C(0, 0) = 1$  y  $C(0, w) = 0$  si  $w > 0$ , pues hay exactamente una forma de conseguir sumar valor 0 sin usar ningún tipo de sello, y no podemos sumar un valor estrictamente positivo si no usamos ningún tipo de sello. Por otro lado, si  $i \geq 0$ ,  $C(i, 0) = 1$  y  $C(i, w) = 0$  para todo  $w < 0$ ; no podemos conseguir un valor negativo cualquiera que sea el número de tipos de sellos usados y sólo hay una forma de conseguir el valor 0, cualquiera que sea el número de sellos del que dispongamos.

Si sabemos  $C(n-1, w)$  para cualquier valor de  $w \geq 0$  y  $n > 0$  entonces

$$C(n, w) = C(n-1, w) + C(n-1, w-v_n) + C(n-1, w-2v_n) + \dots,$$

donde  $C(n-1, w-k \cdot v_n)$  corresponde a usar  $k$  sellos del tipo  $n$  y sumar valor  $w-k \cdot v_n$  con los restantes. Sea  $m = \lfloor \frac{w}{v_n} \rfloor$ . Entonces

$$C(n, w) = \sum_{k=0}^m C(n-1, w-k \cdot v_n).$$

Nuestro algoritmo de programación dinámica se fundamenta en la expresión recursiva recién obtenida. Para ello dispondremos de dos vectores  $C$  y  $Cprev$  de  $V+1$  componentes (tanto  $V$  como los  $v_i$ 's son enteros). Nos bastan dos vectores puesto que  $C(n, w)$  sólo depende de los  $C(n-1, w')$ 's con  $w' \leq w$ . Durante la iteración  $i$ ,  $C[w] = C(i, w)$  y  $Cprev[w] = C(i-1, w)$ .

En primer lugar inicializaremos dichos vectores de manera que  $C[w] = C(0, w)$  y  $Cprev[w] = C(0, w)$ . Luego entraremos en el proceso iterativo de modo que al terminar la iteración  $i$ -ésima  $C[w] = C(i, w)$ .

```

int sellos(const vector<int>& v, int V) {
 vector<int> C(V + 1, 0), Cprev(V + 1, 0);
 // C, Cprev son vectores de V + 1 componentes
 // inicializadas a 0
 C[0] = Cprev[0] = 1;

 for (int i = 0; i < v.size(); ++i) {
 for (int w = v[i]; w <= V; ++w) {
 C[w] = Cprev[w];
 for (int k = 1; k <= w / v[i]; ++k)
 C[w] += Cprev[w - k * v[i]];
 };
 Cprev = C;
 }
 return C[V];
}

```

El coste en caso peor es obviamente  $O(nV^2)$ .

Alternativamente, para  $C(n, w)$  podemos plantear la siguiente recurrencia para  $n > 0$  y  $w \geq v_n$ :

$$C(n, w) = C(n - 1, w) + C(n, w - v_n),$$

mientras que si  $w < v_n$  entonces  $C(n, w) = C(n - 1, w)$ ; es decir, el número de formas de conseguir el valor  $w$  con  $n$  tipos de sellos es igual al número de formas de conseguirlo con  $n - 1$  tipos si  $w < v_n$ , puesto que no podemos usar el tipo  $n$ ; si  $w > v_n$  entonces podemos conseguir el valor  $w$  con  $n - 1$  tipos de sellos sin usar ninguno del tipo  $n$ , o podemos usar un sello del tipo  $n$  y conseguir el valor  $w - v_n$  con los restantes tipos de sellos, o podemos usar un sello del tipo  $n$  y conseguir alcanzar el valor  $w - v_n$  usando de nuevo todos los tipos de sellos. El algoritmo resultante es muy similar al anterior, aunque basta un sólo vector, pues si  $C[w] = C(n - 1, w)$  dicho componente no será alterado al calcular  $C[w'] = C(n, w')$  para  $w' < w$ . Esta es la mejor solución en tiempo y espacio.

```

int sellos(const vector<int>& v, int V) {
 vector<int> C(V + 1, 0);
 // C es un vector de V + 1 componentes
 // inicializadas a 0

 C[0] = 1;
 for (int i = 0; i < v.size(); ++i) {
 for (int w = v[i]; w <= V; ++w)
 C[w] += C[w - v[i]];
 }
 return C[V];
}

```

El coste temporal de este algoritmo es claramente  $O(nV)$  y su coste espacial  $\Theta(V)$ .

**Solució 5.11:** Consulteu la secció 8.7 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>).

**Solució 5.16:** Anomenem  $V_{i,j}$  el valor més gran que es pot assolir si només usem els primers  $i$  objectes,  $1 \leq i \leq n$  i la capacitat de la motxila fos  $j$ ,  $0 \leq j \leq C$ . Llavors, si definim  $V_{i,j} = -\infty$  per  $j < 0$  i  $V_{0,j} = 0$  per tota  $j \geq 0$ ,

$$V_{i,j} = \max\{V_{i-1,j}, V_{i-1,j-p[i]} + v[i]\}, \quad i > 0, j \geq 0.$$

La primera alternativa consisteix en no fer servir l'objecte  $i$  i omplir òptimament la motxilla de capacitat  $j$  amb els primers  $i - 1$  objectes; la segona opció consisteix en posar l'objecte  $i$  a la motxilla i omplir òptimament la motxilla de capacitat  $j - p[i]$  amb els primers  $i - 1$  objectes. Fixeu-vos que si  $j < p[i]$  llavors  $V_{i-1,j-p[i]} = -\infty$  i per tant  $V_{i,j} = V_{i-1,j}$ , es a dir, no posarem l'objecte  $i$  a la motxilla.

Amb la recurrència prèvia, l'algorisme és immediat. Escriurem directament la versió iterativa, amb memoització. La matriu  $C$  contindrà els valors  $V_{i,j}$ . Per a omplir una determinada casella de  $C$  necessitem tenir prèviament calculada la fila precedent. Només cal aquest requisit i per tant podrem omplir la matriu amb un senzill recorregut de dalt abaix, i d'esquerra a dreta.

```

function MOTXILA-ENTERA(p, v, n, C)
 for $j \leftarrow 0$ to C do
 $V[0, j] \leftarrow 0$
 end for
 for $i \leftarrow 1$ to n do
 for $j \leftarrow 0$ to C do
 if $j < p[i]$ then
 $V[i, j] \leftarrow V[i - 1, j]$
 else
 $V[i, j] \leftarrow \max\{V[i - 1, j], V[i - 1, j - p[i]] + v[i]\}$
 end if
 end for
 end for
 return $V[n, C]$
end function

```

El cost temporal i en espai d'aquesta solució és  $\Theta(n \cdot C)$ . Això pot arribar a ser molt elevat, doncs  $C$  pot arribar a ser exponencialment gran respecte a  $n$ , en particular, el cost és  $O(n \cdot 2^n)$ . Si  $C$  no és molt gran aquesta solució serà bastant ràpida i és força senzilla.

D'altra banda, si volem esbrinar quins objectes posem i quins no en la solució òptima, podem reconstruir aquesta informació amb la matriu  $V$ . Si  $V[n, C] = V[n - 1, C]$  llavors vol dir que l'objecte  $n$  no s'ha de posar. Si  $V[n, C] > V[n - 1, C]$  llavors vol dir que sí que posem l'objecte  $n$ . Després examinem  $V[n - 1, C]$  ó  $V[n - 1, C - p[n]]$  segons que l'objecte  $n$  formi part o no de la motxila òptima, raonant de la mateixa manera, “desfent” el camí fins a arribar a  $V[1, ??]$ . El cost d'aquest algorisme de reconstrucció de la solució òptima és  $\Theta(n + C)$ .

**Solució 5.17:** El algoritmo de Floyd (1962) para los caminos mínimos entre todos los pares de vértices de un grafo dirigido  $G$  se basa en el esquema de PD. En realidad

estamos resolviendo  $n^2$  problemas de optimización, uno por cada par de vértices, pero están relacionados unos con otros y los resolveremos simultáneamente.

Supongamos que  $\delta_{k-1}(u, v)$  es la distancia del camino mínimo entre dos vértices  $u$  y  $v$  cuando se usan exclusivamente vértices en  $\{1, \dots, k-1\}$  como vértices intermedios. Entonces podremos calcular todos los caminos mínimos que usan vértices en  $\{1, \dots, k\}$  simplemente comparando si es mejor el camino entre  $u$  y  $v$  que no usa  $k$  ( $\delta_{k-1}(u, v)$ ) y el que usa a  $k$  ( $\delta_{k-1}(u, k) + \delta_{k-1}(k, v)$ ).

Observemos que se da el principio de optimalidad, de manera bastante intuitiva en este problema: si  $\pi$  es un camino óptimo entre  $u$  y  $v$  que pasa por un vértice  $w$  entonces los subcaminos que van de  $u$  a  $w$  y de  $w$  a  $v$  son necesariamente mínimos. Si no lo fueran, entonces  $\pi$  no sería el mínimo!

El caso de base es sencillo:  $\delta_0(u, v) = +\infty$  si entre  $u$  y  $v$  no hay ninguna arista, y  $\delta_0(u, v) = \omega(u, v)$  si  $(u, v)$  es una arista del grafo y  $\omega(u, v)$  denota su peso.

A priori parece necesitarse una tabla de  $\Theta(n^3)$  entradas para resolver el problema, pero puesto que un estado  $\delta_k(u, v)$  depende exclusivamente de valores  $\delta_{k-1}$  podemos mantener exclusivamente dos tablas con  $n^2$  entradas que mantengan los costes  $\delta_{k-1}$  y los costes en curso de la “fase”  $k$ . El orden en que se resolviesen los  $n^2$  subproblemas de la fase  $k$  sería irrelevante.

Pero podemos solucionar el problema con una sola tabla  $n \times n$  ya que durante la fase  $k$  los únicos valores que necesitamos para evaluar el coste  $\delta_k(u, v)$  son el valor previo ( $\delta_{k-1}(u, v)$ ) y los de la fila y columna  $k$  ( $\delta_{k-1}(u, k)$  y  $\delta_{k-1}(k, v)$ ). Pero durante la fase  $k$  no puede cambiar ni la fila ni la columna  $k$  ya que no sirve de nada usar  $k$  como vértice intermedio en un camino que nos lleve desde o hacia  $k$ .

El coste en espacio del algoritmo es  $\Theta(n^2)$  y el coste en tiempo es  $\Theta(n^3)$  (por los tres bucles anidados de  $k, i$  y  $j$ ).

```

procedure FLOYD(g, D)
 $n \leftarrow g.\text{NUM-VERTICES}()$
 Inicializar $D[i, j]$
 ▷ $D[i, j] = \omega(i, j)$ si $(i, j) \in E$; $D[i, j] = +\infty$ en caso contrario
 ▷ Coste: $\Theta(n^2)$
 for $k \leftarrow 1$ to n do
 for $i \leftarrow 1$ to n do
 for $j \leftarrow 1$ to n do
 if $D[i, k] + D[k, j] < D[i, j]$ then
 $D[i, j] \leftarrow D[i, k] + D[k, j]$
 end if
 end for
 end for
 end for
end procedure

```

**Solució 5.20:** Comencem definint  $p_i = p(x_i)$  i  $p_{i,j} = p_i + \dots + p_j = p(x_i) + \dots + p(x_j)$ ,  $1 \leq i \leq j \leq n$ .

Sigui  $T_{i,j}$  un BST òptim pel conjunt de claus  $\{x_i, \dots, x_j\}$  amb probabilitats d'accés normalitzades

$$p'_k = \frac{p_k}{p_{i,j}}, \quad i \leq k \leq j.$$

Suposem que  $x_m$  fos l'arrel de  $T_{i,j}$ . Llavors el subarbre esquerre de  $T_{i,j}$  ha de ser òptim per al conjunt de claus  $\{x_i, \dots, x_{m-1}\}$  amb probabilitats  $p_k^{(L)} = p'_k/p'_{i,m-1} = p_k/p_{i,m-1}$ . Si aquest subarbre no fos òptim, llavors  $T_{i,j}$  no seria òptim. De la mateixa manera, el subarbre dret de  $T_{i,j}$  ha de ser òptim pel conjunt de claus  $\{x_{m+1}, \dots, x_j\}$  amb probabilitats d'accés  $p_k^{(R)} = p_k/p_{m+1,j}$ .

Sigui  $C_{i,j}$  el cost esperat de  $T_{i,j}$ :

$$C_{i,j} = \sum_{i \leq k \leq j} p'_k d(T_{i,j}, x_k).$$

Sigui  $x_m$  l'element arrel de  $T_{i,j}$ , i que per tant, fa que  $C_{i,j}$  sigui mínim. Llavors

$$\begin{aligned} C_{i,j} &= \sum_{i \leq k \leq j} \frac{p_k}{p_{i,j}} (d(T_{i,j}, x_k) + 1) \\ &= \sum_{i \leq k < m} \frac{p_k}{p_{i,j}} (d(T_{i,m-1}, x_k) + 1 + 1) + \sum_{m < k \leq j} \frac{p_k}{p_{i,j}} (d(T_{m+1,j}, x_k) + 1 + 1) \\ &= \sum_{i \leq k \leq j} \frac{p_k}{p_{i,j}} \sum_{i \leq k < m} \frac{p_k}{p_{i,j}} (d(T_{i,m-1}, x_k) + 1) + \sum_{m < k \leq j} \frac{p_k}{p_{i,j}} (d(T_{m+1,j}, x_k) + 1) \\ &= 1 + \sum_{i \leq k < m} \frac{p_k}{p_{i,j}} d(T_{i,m-1}, x_k) + \sum_{m < k \leq j} \frac{p_k}{p_{i,j}} d(T_{m+1,j}, x_k). \end{aligned}$$

Si multipliquem els dos costats d'aquesta equació per  $p_{i,j}$  i definim  $\Delta_{i,j} = p_{i,j} C_{i,j}$  llavors tenim

$$\begin{aligned} \Delta_{i,j} &= p_{i,j} \sum_{i \leq k < m} p_k d(T_{i,m-1}, x_k) + \sum_{m < k \leq j} p_k d(T_{m+1,j}, x_k) \\ &= p_{i,j} + \Delta_{i,m-1} + \Delta_{m+1,j}. \end{aligned}$$

Com que  $m$  és l'índex que minimitza  $\Delta_{i,j}$  en general tindrem:

$$\Delta_{i,j} = p_{i,j} + \min_{i \leq m \leq j} \{\Delta_{i,m-1} + \Delta_{m+1,j}\}.$$

per a completar la recurrència necessitem els casos de base: per a tot  $i$ ,  $1 \leq i \leq n$ ,  $\Delta_{i,i} = p_{i,i} C_{i,i} = p_{i,i}$ . A més convé definir  $\Delta_{i,i-1} = C_{i,i-1} = 0$  (cost d'un BST buit).

Per a fer una versió iterativa eficient, amb memoització, necessitarem una matriu  $D$  per a guardar els valors  $\Delta_{i,j}$ . També convé guardar els "pesos"  $p_{i,j}$  en una altra matriu  $P$  de talla  $n \times n$ . Com que no només ens interessa el cost del BST òptim, tindrem una matriu  $R$  per a emmagatzemar les arrels: així,  $R[1, n]$  serà l'índex de l'element arrel del BST i en general  $R[i, j] = m$  serà l'índex de l'element a l'arrel de  $T_{i,j}$ .

Per a calcular un element  $\Delta_{i,j}$  necessitem tots els elements de la mateixa fila  $i$  en columnes precedents ( $m - 1 < j$ ) i tots els elements de la mateixa columna ( $j$ ) de les files per sota ( $i < m + 1$ ). Per tant la matriu  $D$  s'ha d'omplir començant per abaix i d'esquerra a dreta, amb  $j \geq i$ .

```

procedure BST- $\hat{\text{OPTIM}}(p, n, D, R)$
 ▷ Omplim la matriu P ; $P[i, j] = p_{i,j}$
 for $i \leftarrow 1$ to n do
 $P[i, i - 1] \leftarrow 0$
 for $j \leftarrow i$ to n do
 $P[i, j] \leftarrow P[i, j - 1] + p[j]$
 end for
 end for
 ▷ I ara les matrius D i R ...
 $D[n + 1, n] \leftarrow 0$
 for $i \leftarrow n$ to 1 do
 $D[i, i - 1] \leftarrow 0$
 for $j \leftarrow i$ to n do
 $min \leftarrow \infty$
 for $m \leftarrow i$ to j do
 if $min \geq D[i, m - 1] + D[m + 1, j]$ then
 $min \leftarrow D[i, m - 1] + D[m + 1, j]$; $R[i, j] \leftarrow m$
 end if
 end for
 $D[i, j] \leftarrow P[i, j] + min$
 end for
 end for
end procedure

```

Aquesta solució té cost en espai  $\Theta(n^2)$  i cost temporal  $\Theta(n^3)$ . El cost del BST òptim és  $C_{1,n} = \Delta_{1,n}$  i la matriu  $R$  ens permet reconstruir el BST en temps linial:

```

function RECONSTRUIR-BST- $\hat{\text{OPTIM}}(i, j, R, X)$
 if $i > j$ then
 return arbre buit
 else
 $m \leftarrow R[i, j]$
 $T \leftarrow \text{NewNode}$; $T \rightarrow info \leftarrow X[m]$
 $T \rightarrow fill_esq \leftarrow \text{RECONSTRUIR-BST-}\hat{\text{OPTIM}}(i, m - 1, R, X)$
 $T \rightarrow fill_dret \leftarrow \text{RECONSTRUIR-BST-}\hat{\text{OPTIM}}(m + 1, j, R, X)$
 end if
 return T
end function

```

Es pot aconseguir una millora molt significativa en el cost un cop que ens adonem que  $R[i, j - 1] \leq R[i, j] \leq R[i + 1, j]$ . El bucle que calcula  $D[i, j]$  no ha de considerar tots els valors de  $m$  entre  $i$  i  $j$ :



```

procedure BST- $\hat{\text{OPTIM}}(p, n, D, R)$
 ...
 $\min \leftarrow \infty$
 for $m \leftarrow R[i, j - 1]$ to $R[i + 1, j]$ do
 if $\min \geq D[i, m - 1] + D[m + 1, j]$ then
 $\min \leftarrow D[i, m - 1] + D[m + 1, j]$; $R[i, j] \leftarrow m$
 end if
 end for
 $D[i, j] \leftarrow P[i, j] + \min$
 ...
end procedure

```

El cost de l'algorisme es redueix a  $\Theta(n^2)$ : en efecte, el cost es  $\Theta(\sum_{1 \leq i \leq j \leq n} 1 + R[i + 1, j] - R[i, j - 1])$  i els termes amb les  $R[i, j]$ 's es cancel·len.

**Solució 5.22:** Nuestra solución es muy similar a la solución de Floyd para el problema de caminos mínimos en un grafo. Necesitamos generalizar la noción de arbitraje entre dos divisas  $i$  y  $j$  cualesquiera. Sea  $\delta_k(i, j)$  el valor del mejor arbitraje entre la divisa  $i$  y la divisa  $j$  utilizando exclusivamente las divisas 1 a  $k$  como divisas intermedias. El valor que buscamos es  $\delta_n(i, i)$ . Si no podemos usar ninguna divisa intermedia, el mejor arbitraje se obtiene por conversión directa:  $\delta_0(i, j) = C[i, j]$ . Por otro lado, si  $k > 0$

$$\delta_k(i, j) = \max\{\delta_{k-1}(i, k) \cdot \delta_{k-1}(k, j), \delta_{k-1}(i, j)\}.$$

Para nuestro algoritmo de programación dinámica usaremos matrices de tamaño  $n \times n$  para guardar los valores de las  $\delta$ 's.

Puesto que  $\delta_k(i, j)$  depende exclusivamente de valores de  $\delta_{k-1}$  sólo necesitamos almacenar dos matrices con los valores de  $\delta_k$  y  $\delta_{k-1}$ . No podemos evitar el uso de las dos matrices como en el algoritmo de Floyd ya que puede ocurrir que  $\delta_{k-1}(k, k) \neq 1$ . En cambio en el algoritmo de Floyd se usa el hecho de que el camino mínimo entre  $k$  y  $k$  es 0, para cualquier  $k$ , y sean cuales sean los vértices intermedios que se pueden usar.

El resultado final que nos queda es:

```

procedure MEJORARBITRAJE($C[1..n, 1..n], i$)
 $Dprev \leftarrow C$
 for $k \leftarrow 1$ to n do
 for $i \leftarrow 1$ to n do
 for $j \leftarrow 1$ to n do
 $D[i, j] \leftarrow \text{MAX}(Dprev[i, k] * Dprev[k, j], Dprev[i, j])$
 end for
 end for
 $Dprev \leftarrow D$
 end for
 return $D[i, i]$
end procedure

```

Esta solución requiere obviamente espacio en  $\Theta(n^2)$  y tiene coste  $\Theta(n^3)$ .

**Solució 5.24:** Sea  $d_{i,j}$  la distancia de edición entre las secuencias  $s_1, \dots, s_i$  y  $s'_1, \dots, s'_j$  con  $0 \leq i \leq n$  y  $0 \leq j \leq m$ . Es fácil observar que si ambas secuencias son vacías (es decir,  $i = 0$  y  $j = 0$ ) su distancia de edición es 0 ( $d_{0,0} = 0$ ). También es fácil ver que si una de las dos secuencias es vacía y la otra no, su distancia de edición es la longitud de la secuencia no vacía, es decir,  $d_{i,0} = i$  y  $d_{0,j} = j$ .

En general (cuando  $i > 0$  y  $j > 0$ ) dadas las subsecuencias  $s_1, \dots, s_i$  y  $s'_1, \dots, s'_j$  para transformar la primera subsecuencia en la segunda podemos realizar cualquiera de las siguientes operaciones:

- Borrar  $s_i$  y transformar la subsecuencia  $s_1, \dots, s_{i-1}$  en la subsecuencia  $s'_1, \dots, s'_j$ . En este proceso intervienen una operación de edición (borrar  $s_i$ ) más el número de operaciones de edición que requiera la transformación.
- Insertar  $s'_j$  al final de  $s_1, \dots, s_i$  y transformar la subsecuencia  $s_1, \dots, s_i$  en la subsecuencia  $s'_1, \dots, s'_{j-1}$ . Se realiza una operación de edición (insertar  $s'_j$ ) y el número de operaciones de edición que requiera la transformación.
- Remplazar a  $s_i$  por  $s'_j$  y transformar la subsecuencia  $s_1, \dots, s_{i-1}$  en la subsecuencia  $s'_1, \dots, s'_{j-1}$ . El número de operaciones que se realiza en este caso depende de cómo son los caracteres  $s_i$  y  $s'_j$  entre ellos. Si son iguales, el número de operaciones es el que requiera la transformación, si son diferentes es el número anterior más uno donde el uno corresponde al remplazo.

Por tanto, para calcular la distancia de edición entre dos subcadenas cualesquiera debemos escoger de las tres opciones anteriores la que realice el menor número de operaciones. Es decir,

$$d_{i,j} = \min\{1 + d_{i-1,j}, 1 + d_{i,j-1}, c + d_{i-1,j-1}\}, \quad i, j > 0$$

con  $c = 0$  si  $s_i = s'_j$  y  $c = 1$  en caso contrario.

Si juntamos todas las observaciones anteriores obtenemos la siguiente recurrencia para  $d_{i,j}$ :

$$d_{i,j} = \begin{cases} 0 & i = 0 \text{ y } j = 0 \\ i & i > 0 \text{ y } j = 0 \\ j & i = 0 \text{ y } j > 0 \\ \min\{1 + d_{i-1,j}, 1 + d_{i,j-1}, \delta(s_i, s'_j) + d_{i-1,j-1}\} & i > 0 \text{ y } j > 0 \end{cases}$$

donde  $\delta(x, y) = 0$  si  $x = y$  y  $\delta(x, y) = 1$  si  $x \neq y$ .

Es fácil ver a partir de la recurrencia anterior que la implementación recursiva directa de la solución incurrirá en la solución múltiple de subproblemas idénticos, pero utilizando una matriz de tamaño  $(|s| + 1) \times (|s'| + 1)$  donde  $|s|$  denota la longitud de la cadena podemos guardar los resultados de los cálculos intermedios y acceder a ellos en tiempo constante cada vez que sea necesario.

A continuación se muestra una posible implementación de la solución propuesta.

```

#include <iostream>
#include <vector>
#include <string>

using namespace std;

int minim(int a, int b, int c) {
 int min = a;
 if (b < min) min = b;
 if (c < min) min = c;
 return min;
}

int edit_dist(const string& s, const string& z) {
 int n = s.size();
 int m = z.size();
 vector<vector<int>> > M(n+1,m+1);

 // Casos base: inicializacio de 1eras fila y columna
 M[0][0] = 0;
 for (int i = 1; i < n+1; ++i) M[i][0] = i;
 for (int j = 1; j < m+1; ++j) M[0][j] = j;

 // Caso general: calculo del valor de cada posici3n de la matriz
 for (int i = 1; i < n+1; ++i) {
 for (int j = 1; j < m+1; ++j) {
 int c;
 if (s[i-1] == z[j-1]) c = 0;
 else c = 1;
 M[i][j] = minim(M[i-1][j-1]+c, M[i-1][j]+1, M[i][j-1]+1);
 }
 }
 return M[n][m];
}

```

**Soluci3 5.26:** Sigui  $L(i, j)$  el nombre m3nim de l3nies necessari per a escriure un text de longitud  $m = A[j+1] - A[i-1]$  amb salts de l3nia autoritzats pel subarray  $A[i..j]$ ,  $1 \leq i \leq n+1$ ,  $0 \leq j \leq n$ ,  $i \leq j+1$ . El valor que ens interessa 3s  $L(1, n)$ . Si  $m \leq x$ , llavors  $L(i, j) = 1$  i no caldr3 insertar cap salt de l3nia. Altrament, si  $i = j+1$  i  $A[j+1] - A[j] > x$  llavors  $L(i, j) = +\infty$ , doncs no hi ha cap salt de l3nia que puguem inserir i el text que volem subdividir en l3nies t3 longitud  $> x$ . Finalment, si  $i \leq j$  i  $A[j+1] - A[i-1] > x$ , tenim la recurr3ncia

$$L(i, j) = \min_{i \leq k \leq j} \{L(i, k-1) + L(k+1, j)\}.$$

En general, l'element  $L[i, j] = L(i, j)$  dep3n dels valors de la mateixa fila en columnes a la seva esquerra i d'elements en la mateixa columna en files per sota. Aix3 vol

dir que la manera correcta d'omplir la matriu és començant des d'abaix ( $i = n$ ) cap dalt ( $i = 1$ ), i d'esquerra a dreta.

▷ Inicialització de la matriu  $L$

```

for $i \leftarrow 1$ to $n + 1$ do
 if $A[i] - A[i - 1] \leq x$ then
 $L[i, i - 1] \leftarrow 1$
 else
 $L[i, i - 1] \leftarrow +\infty$
 end if
end for

for $i \leftarrow n$ downto 1 do
 for $j \leftarrow i$ to n do
 if $A[j + 1] - A[i - 1] \leq x$ then
 $L[i, j] \leftarrow 1$
 else
 ▷ $\min L \leftarrow \min_{i \leq k \leq j} \{L[i, k - 1] + L[k + 1, j]\}$
 $\min L \leftarrow +\infty$
 for $k \leftarrow i$ to j do
 if $L[i, k - 1] + L[k + 1, j] < \min L$ then
 $\min L \leftarrow L[i, k - 1] + L[k + 1, j]$
 end if
 end for
 end if
 end for
end for

```

El cost en espai d'aquest algorisme és  $\Theta(n^2)$ . El cost en temps és  $\Theta(n^3)$ . Tenint en compte certes propietats del problema es pot aconseguir reduir aquest cost a  $\Theta(n^2)$ , però la solució de cost  $\Theta(n^3)$  presentada aquí es considera perfectament vàlida en aquest contexte.

## Algorismes voraços

**Solució 6.1:** Hem demostrat el principi d'optimalitat local per a les denominacions  $25 > 10 > 5 > 1$  i també el principi d'estructura òptima per a tota combinació de denominacions. Sigui ara  $50 > 20 > 10 > 5 > 2 > 1$  les denominacions disponibles. Volem mostrar que sempre existeix una solució òptima al problema de la màquina expenedora per a  $n$  cèntims que inclou una moneda de denominació  $d$ , on  $d$  és la denominació més gran disponible tal que  $d \leq n$ .

Suposem, en sentit contrari, una solució òptima al problema de la màquina expenedora per a  $n$  cèntims que no inclou cap moneda de denominació  $d$ . Si  $1 \leq n < 2$ , aleshores  $d = 1$  i la solució òptima consisteix en una moneda de 1 cèntim.

Si  $2 \leq n < 5$ , aleshores  $d = 2$  i com que aquesta solució òptima no conté cap

moneda de 2 cèntims, només conté monedes de 1 cèntim i canviant-ne dues per una moneda de 2 cèntims obtenim una solució amb una moneda menys.

Si  $5 \leq n < 10$ , aleshores  $d = 5$  i com que aquesta solució òptima no conté cap moneda de 5 cèntims, només conté monedes de 2 i 1 cèntims. En particular, conté monedes de 2 i 1 cèntim que sumen 5 cèntims, i canviant-les per una moneda de 5 cèntims obtenim una solució amb menys monedes.

Si  $10 \leq n < 20$ , aleshores  $d = 10$  i com que aquesta solució òptima no conté cap moneda de 10 cèntims, només conté monedes de 5, 2 i 1 cèntim. En particular, conté monedes de 5, 2 i 1 cèntim que sumen 10 cèntims, i canviant-les per una moneda de 10 cèntims obtenim una solució amb menys monedes.

Si  $20 \leq n < 50$ , aleshores  $d = 20$  i com que aquesta solució òptima no conté cap moneda de 20 cèntims, només conté monedes de 10, 5, 2 i 1 cèntim. En particular, conté monedes de 10, 5, 2 i 1 cèntim que sumen 20 cèntims, i canviant-les per una moneda de 20 cèntims obtenim una solució amb menys monedes.

Finalment, si  $50 \leq n$ , aleshores  $d = 50$  i com que aquesta solució òptima no conté cap moneda de 50 cèntims, només conté monedes de 20, 10, 5, 2 i 1 cèntim. Si conté tres monedes de 20 cèntims, les podem canviar per una moneda de 50 cèntims més una de 10 cèntims, obtenint una solució amb una moneda menys. Si conté com a molt dues monedes de 20 cèntims, aleshores conté monedes de 10, 5 i 1 cèntim que sumen 10 cèntims, i canviant-les per una moneda de 50 cèntims obtenim una solució amb menys monedes.

Siguin ara  $25 > 10 > 1$  les denominacions disponibles. No sempre existeix una solució òptima al problema de la màquina expenedora per a  $n$  cèntims que inclou una moneda de denominació  $d$ , on  $d$  és la denominació més gran disponible tal que  $d \leq n$ . De fet, l'algorisme voraç per a  $n = 30$  dóna una solució amb 6 monedes ( $30 = 25 + 1 + 1 + 1 + 1 + 1$ ) mentre que hi ha una solució no voraç amb només 3 monedes ( $30 = 10 + 10 + 10$ ).

**Solució 6.3:** El algoritmo voraz retrasará en cada ocasión, lo máximo posible, la siguiente parada. Es decir,

$$S_0 = \max\{i \leq n + 1 \mid D[i] \leq Z\},$$

$$S_j = \max\{i \leq n + 1 \mid D[i] - D[S_{j-1}] \leq Z\}$$

Un sencillo algoritmo de coste lineal  $O(n)$  hace el correspondiente cálculo:

```
int i = 0; double d = D[0];
while (i < D.size()) {
 while (i < D.size() && D[i] - d <= Z)
 ++i;
 d = D[i - 1];
 sol.push_back(i - 1);
}
```

La optimalidad de la solución es inmediata. Llamemos  $S_g$  a la solución hallada por el voraz. Sea  $S'$  una solución alternativa. Dicha solución tendrá al menos el mismo número de paradas que  $S_g$ . Si  $S' \neq S_g$  entonces tiene que haber una etapa  $j$  del camino tal que  $S_g[i] = S'[i]$  para  $0 \leq i < j$  y  $S'[j] < S_g[j]$ , pues el voraz para lo más lejos posible. Pero

si  $S'[j] < S_g[j]$  eso significa que en las sucesivas etapas la solución alternativa tendrá parada antes o en el mismo lugar que la solución voraz, y en cualquier caso no puede hacer menos paradas que el voraz.

**Solució 6.4:** Una estratègia voraç per resoldre aquest problema consisteix a ordenar el conjunt de punts i repetir el procés d'estreure-hi el primer punt, afegir un interval unitari amb aquest punt com a extrem esquerre, i eliminar els punts continguts dins aquest interval unitari, fins que el conjunt sigui buit.

Aquest algorisme es pot implementar amb cost temporal  $\Theta(n \log n)$ .

Per exemple, donat  $X = \{7/4, 7/2, 1/2, 3, 3/2, 0\}$ , aquesta estratègia voraç dona  $I = \{[0, 1], [3/2, 5/2], [7/2, 9/2]\}$ , on  $0, 1/2 \in [0, 1]$ ,  $3/2, 7/4, 2 \in [3/2, 5/2]$ ,  $7/2 \in [7/2, 9/2]$ .

```

function UNIT_INTERVAL_COVER(X)
 $I := \emptyset$
 ordenar X ascendentment
 while X no és buit do
 $x :=$ primer element de X
 $X := X \setminus \{x\}$
 $I := I \cup \{[x, x + 1]\}$
 $y := x + 1$
 while X no és buit and $x \leq y$ do
 $X := X \setminus \{x\}$
 if X no és buit then
 $x :=$ primer element de X
 end if
 end while
 end while
 return I
end function

```

Principi d'optimalitat: A cada iteració, existeix un conjunt òptim d'interval·ls per als punts  $P = \{z \mid z \geq x\}$  que conté l'interval  $[x, x + 1]$ . De fet, un conjunt òptim d'interval·ls per a  $P$  ha d'incloure qualsevol interval  $i_x$  que cobreix  $x$ . Com que cap element de  $P$  és menor que  $x$ , es pot desplaçar  $i_x$  cap a la dreta fins que el seu extrem esquerre sigui igual a  $x$ . Substituint l'interval  $i_x$  per aquest interval desplaçat  $i'_x$ , el conjunt d'interval·ls encara cobreix  $P$  i segueix sent òptim. Però,  $i'_x = [x, x + 1]$ , amb la qual cosa existeix un conjunt òptim d'interval·ls per a  $P$  que conté l'interval  $[x, x + 1]$ .

Principi d'estructura òptima: Sigui  $I$  solució òptima de  $X$ , sigui  $I' \subseteq I$  i  $X' = \{x \in X \mid \exists i \in I', x \in i'\}$ . Aleshores,  $I'$  és solució òptima de  $X'$ . De fet, si existeix una solució de  $X'$  amb menys de  $|I'|$  interval·ls, es pot substituir dins la solució  $I$  de  $X$ , la qual cosa contradiu la hipòtesi que aquesta solució és òptima.

Una solució alternativa:

Un algorisme voraç per resoldre aquest problema pot establir la selecció del nou interval com el primer necessari per a cada punt dels no inclosos en cap altre interval.

En aquesta representació, cal interpretar el conjunt  $I$  d'interval solució com aquells formats per cada un dels elements del vector  $I$ , i el mateix valor més 1.

```
void intervals(vector<double>& X, vector<double>& I) {
 sort(X.begin(), X.end());
 I.push_back(X[0]);
 for (int i = 1; i < X.size(); ++i) {
 if (X[i] > I[I.size()-1] + 1) {
 I.push_back(i);
 }
 }
}
```

L'eficiència d'aquest algorisme és clar que pertany a  $\Theta(n \log n)$ , sent  $n$  el número de punts. El cost més gran correspon a ordenar el vector  $X$ ; l'iteració que ve a continuació té cost  $\Theta(n)$ . Aquesta solució troba l'òptim, ja que qualsevol altra solució que fos òptima hauria de seleccionar un interval inicial que contingues  $X[0]$ , que si no fos aquest, hauria d'acabar abans i per tant no podria contenir més valors que aquest. Així doncs, aquesta selecció voraz és òptima. Afegint que el problema manté el principi d'optimalitat, és a dir, que les subsolucions a subproblemes també són òptimes, sembla demostrada l'optimalitat d'aquesta solució.

**Solució 6.5:** La solución voraz consiste en seleccionar los archivos por orden creciente de talla.

Alternativamente, se puede ordenar los archivos por orden creciente de tamaño e ir seleccionando los archivos en dicho orden en tanto su tamaño conjunto no exceda  $L$ .

**procedure** SELECCIONAARCHIVOS( $\ell[1..n]$ ,  $L$ ,  $result$ )

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$F[i].ind \leftarrow i$

$F[i].\ell \leftarrow \ell[i]$

**end for**

  Ordenar el vector  $F$  de pares  $\langle i, \ell[i] \rangle$  por orden creciente de  $\ell$ 's

$c \leftarrow 0$ ;  $result \leftarrow \emptyset$

$j \leftarrow 1$

**while**  $c + F[j].\ell \leq L$  **do**

$c \leftarrow c + F[j].\ell$

$result \leftarrow result \cup \{F[j].ind\}$

$j \leftarrow j + 1$

**end while**

**end procedure**

El coste del algoritmo de arriba viene dominado por el coste de la ordenación inicial del vector ( $\Theta(n \log n)$  si usamos un algoritmo como *mergesort* o *heapsort*, por ejemplo). El bucle posterior tiene coste  $\mathcal{O}(n)$ , pues como mucho haremos  $n - 1$  iteraciones y cada una tiene coste constante.

Una implementación mejor consiste en crear inicialmente un *min-heap* con el vector  $F$  de pares, tomando la talla de cada archivo como prioridad. Este paso tiene coste  $\Theta(n)$ .

Después se hace una iteración similar a la de más arriba, pero en cada paso el archivo seleccionado es el que obtenemos haciendo una extracción del mínimo del *heap*. Este bucle tendría coste  $\mathcal{O}(n \log n)$ . En caso peor será  $\Theta(n \log n)$ , pero en muchos casos, el coste será inferior. Por ejemplo, si el número de archivos que se pueden almacenar en el dispositivo es mucho menor que  $n$ , el número de iteraciones del bucle que hace la selección será mucho menor que  $n$ , y el coste total del algoritmo será más cercano a  $\Theta(n)$  que a  $\Theta(n \log n)$ .

```

procedure SELECCIONAARCHIVOS($\ell[1..n]$, L , $result$)
 for $i \leftarrow 1$ to n do
 $F[i].ind \leftarrow i$
 $F[i].\ell \leftarrow \ell[i]$
 end for
 ▷ Crea un min-heap sobre F usando ℓ como prioridad
 MAKEHEAP(F)
 $c \leftarrow 0$; $result \leftarrow \emptyset$
 $x \leftarrow \text{EXTRACTMIN}(F)$
 while $c + x.\ell \leq L$ do
 $c \leftarrow c + x.\ell$
 $result \leftarrow result \cup \{x.ind\}$
 $x \leftarrow \text{EXTRACTMIN}(F)$
 end while
end procedure

```

La corrección del algoritmo, en cualquiera de las dos variantes, es inmediata. Supongamos que el conjunto retornado por el voraz es  $S$  y que existe un conjunto  $S' \neq S$  que tiene tamaño total  $\leq L$  pero que contiene más archivos que  $S$ . Entonces  $S'$  contendrá un cierto archivo  $x$  de talla más grande que cualquiera que los que hay en  $S$ , y alguno de los que están en  $S$  no estará en  $S'$  (porque sino los archivos contenidos en  $S'$  tendrían tamaño total  $> L$ ). Pero entonces podemos reemplazar el archivo  $x$  por uno de los pequeños que estaban en  $S$  y no en  $S'$  y la solución obtenida tendría menor tamaño total y el mismo número de archivos. Este razonamiento se puede aplicar repetidamente y finalmente llegaríamos a la conclusión de que  $S \subset S'$ , lo cual es una contradicción, a menos que  $S = S'$ .

**Solució 6.7:** La secuencia a construir es una permutación de las  $n$  tareas de modo que cada una de ellas acabe como muy tarde en su  $t[i]$ . La función de selección que de forma natural se desprende del enunciado es la de elegir la tarea que acaba antes de entre todas las no consideradas. Por tanto, es suficiente con ordenar las  $n$  tareas por orden creciente de  $t[i]$  y comprobar si esa secuencia es factible, es decir, comprobar si de esa forma todas las tareas se pueden ejecutar y acaban en, como mucho, su  $t[i]$ . Para cada una de las tareas debe suceder que la suma de los tiempos de las tareas que le preceden más el suyo propio ha de ser menor o igual a su instante de finalización. Esa es, por definición, una secuencia factible.

El problema es determinar si siempre que exista una secuencia factible para las  $n$  tareas de partida, el algoritmo voraz propuesto va a encontrar siempre una solución.



Para demostrarlo vamos a comprobar que cualquier solución factible (se ejecutan todas y antes o en su  $t[i]$ ) se puede transformar en una secuencia ordenada por  $t[i]$  que es la que encuentra el voraz. Supongamos que existe una secuencia factible,  $SF$ , en la que las  $n$  tareas no se encuentran en orden creciente de instante de finalización. Recorremos  $SF$  de derecha a izquierda y nos detenemos sobre el primer par que cumpla la condición de estar desordenado, es decir, en la posición  $k$  encontramos una tarea  $r$  cuyo  $t[r]$  es mayor que el  $t[s]$  que corresponde a la tarea  $s$  colocada en la posición  $k + 1$ .

Podemos intercambiar las dos tareas sin ningún problema: Si la tarea  $s$  se podía ejecutar en el lugar  $k + 1$ , la tarea  $r$  que tiene un  $t[r]$  todavía mayor que  $t[s]$  seguro que se puede retrasar y, por supuesto, no hay ningún problema en ejecutar antes  $s$  de lo que lo hace ahora (el problema puede ser retrasar porque las tareas tienen un instante máximo de finalización pero nunca será adelantar porque lo que se hace es acabar todavía antes de lo planificado). Las tareas programadas por delante de  $r$ , en las posiciones desde la 1 a la  $k - 1$  no se ven afectadas por el intercambio y las tareas programadas por detrás de la  $s$ , desde la posición  $k + 2$  a la  $n$ , tampoco ya todas pueden empezar en el mismo instante en que tenían previsto. Lo que sí puede suceder es que al intercambiar  $r$  y  $s$ , la secuencia desde  $k + 1$  hasta  $n$  deje de estar ordenada. En concreto lo que puede suceder es que ahora la tarea que ha ido a parar a  $k + 1$  tenga un instante de terminación superior a la que estaba en  $k + 2$ . No pasa nada, simplemente repetimos el proceso de “ordenar” pero ahora desde  $k + 1$  hasta  $n$ . Con todo esto habremos conseguido que desde  $k$  hasta  $n$  las tareas estén ordenadas en orden creciente de instante de finalización. Basta con repetir el proceso las veces que sea necesario (hasta que  $k = 1$ ) y obtendremos una secuencia factible  $SF'$  obtenida a partir de  $SF$  en la que las tareas están en orden creciente de  $t[i]$  y esta  $SF'$  es precisamente la solución obtenida por el voraz.

**function** SECUENCIA-FACTIBLE( $v[1..n]$ ,  $L$ )

- ▷  $v$  es un vector de tuplas:  $v[i].l = l[i]$  y  $v[i].t = t[i]$
- ▷ En  $L$  devolveremos la secuencia de las  $n$  tareas en
- ▷ el orden de proceso si es factible, y el resultado de la
- ▷ función será cierto. En caso contrario  $L$  estará vacía.

Ordenar  $v$  por orden creciente del atributo  $t$

$L \leftarrow []$ ;

$es\_factible \leftarrow \text{cierto}$

$acum \leftarrow 0; i \leftarrow 1$

**while**  $i \leq n \wedge es\_factible$  **do**

$acum \leftarrow acum + v[i].l$

**if**  $acum \leq v[i].t$  **then**

$i \leftarrow i + 1$

$L \leftarrow L.AÑADIR(i)$

**else**

$es\_factible \leftarrow \text{false}$

$L \leftarrow []$

**end if**

**end while**

```

 return es_factible
end function

```

El coste es  $\Theta(n \log n)$  debido a la ordenación porque el bucle tiene coste  $\Theta(n)$  en el caso peor.

**Solució 6.12:** Cada ficha ha de hacer como mínimo tantos movimientos verticales como su distancia a la fila  $i$  dada. Por tanto, el total de movimientos verticales a efectuar es

$$\sum_{j=0}^{n-1} |X_j - i|$$

Si sólo efectuamos movimientos verticales, esto “apilaría” las fichas que se encontrasen en la misma columna. Para hallar los movimientos horizontales seguimos una estrategia voraz: cada ficha se mueve a la casilla libre más cercana a su columna original. Para ello consideramos las fichas por orden creciente de columna, y vamos recorriendo las columnas una a una: en la columna 0 ponemos a la ficha que esté en la columna más proxima (será la primera ficha, al ser consideradas en orden creciente de columnas), en la columna 1 pondremos la segunda ficha, y así sucesivamente. Los movimientos horizontales efectuados son:

$$\sum_{j=0}^{n-1} |\bar{Y}_j - j|,$$

siendo  $\bar{Y}$  el vector que nos da las columnas de las fichas en orden creciente.

```

int moviments (int n, int i, vector<int>& X, vector<int>& Y) {
 int m = 0;
 for (int x = 0; x < n; x++) {
 m += abs(X[x] - i);
 }
 sort(Y.begin(), Y.end()); // destructive
 for (int y = 0; y < n; y++) {
 m += abs(Y[y] - y);
 }
 return m;
}

```

**Solució 6.17:** L'algorisme de Dijkstra a alt nivell és:

**procedure** DIJKSTRA( $G, s, D$ )

- ▷  $G = \langle V, E \rangle$  és un graf dirigit ponderat,  $s$  un dels seus vèrtexos
- ▷ En acabar,  $D[u]$  és la distància mínima entre  $s$  i  $u$ , per
- ▷ a tot vèrtex  $u \in V$ .

**for all**  $v \in V(G)$  **do**

$D[v] \leftarrow +\infty$

**end for**

```

 $D[s] \leftarrow 0$
 $CAND \leftarrow V(G)$
while $CAND \neq \emptyset$ do
 $u \leftarrow$ el vèrtex de $CAND$ amb D mínima
 $CAND \leftarrow CAND \setminus \{u\}$
 for all $v \in G.SUCCESORS(u)$ do
 $d \leftarrow D[u] + G.PES(u, v)$
 if $d < D[v]$ then
 $D[v] \leftarrow d$
 end if
 end for
end while
end procedure

```

El que farem serà tenir una taula  $LC$  tal que  $LC[u]$  sigui el nombre d'arcs mínim possible en un camí mínim entre  $s$  i  $u$ , per a tot vèrtex  $u \in V$ . Durant l'iteració se satisfà que  $LC[u]$  és el nombre mínim d'arcs possible en un camí mínim entre  $s$  i  $u$  que passi només per vèrtexos del conjunt  $V \setminus CAND$ . Inicialment,  $LC[u] = +\infty$  per a tot  $u \neq s$  i  $LC[s] = 0$ . Quan trobem un nou camí mínim entre  $s$  i  $v$  que passa per  $u$  i tal que  $D[v] = D[u] + G.PES(u, v)$  llavors tenim un empat i haurem d'escollir el que tingui un nombre d'arcs més petit. Específicament, haurem de comparar  $LC[v]$  i  $LC[u] + 1$ , i actualitzar com correspongui.

```

procedure DIJKSTRA-MOD(G, s, D, LC)
 for all $v \in V(G)$ do
 $D[v] \leftarrow +\infty$; $LC[v] \leftarrow +\infty$
 end for
 $D[s] \leftarrow 0$; $LC[s] \leftarrow 0$
 $CAND \leftarrow V(G)$
 while $CAND \neq \emptyset$ do
 $u \leftarrow$ el vèrtex de $CAND$ amb D mínima
 $CAND \leftarrow CAND \setminus \{u\}$
 for all $v \in G.SUCCESORS(u)$ do
 $d \leftarrow D[u] + G.PES(u, v)$
 if $d < D[v]$ then
 $D[v] \leftarrow d$; $LC[v] \leftarrow LC[u] + 1$
 else if $d = D[v] \wedge LC[u] + 1 < LC[v]$ then
 $LC[v] \leftarrow LC[u] + 1$
 end if
 end for
 end while
end procedure

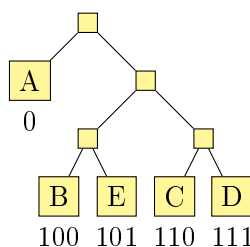
```

**Solució 6.29:** L'únic que necessitem per a resoldre aquest problem és modificar un algorisme eficient de càlcul d'arbres d'expansió mínims, per exemple, Kruskal o Prim, de tal manera que es faci servir una i només aresta que tanca un cicle.

La modificació de l'algorisme de Kruskal és més senzilla i resulta més intuïtiva. Afegim un booleà `uniciclic` i l'inicialitzem a `false`. Durant l'iteració principal, quan estem considerant una aresta  $e$  de pes mínim entre les arestes encara no visitades, afegirem  $e$  a la solució si `uniciclic` és fals o  $e$  no tanca un cicle en el conjunt d'arestes ja seleccionades. En cas contrari, l'aresta  $e$  s'ha de rebutjar. Si en posar l'aresta  $e$  en el conjunt d'arestes seleccionades tanquem un cicle llavors fem `uniciclic=true`. Com tot graf unicíclic d'expansió d'un graf amb  $n$  vèrtexos (que en tingui algún graf unicíclic d'expansió) té necessàriament  $n$  arestes, el bucle principal finalitza quan el conjunt d'arestes seleccionades té  $n$ , no  $n - 1$  que es el que fem en l'algorisme de Kruskal original. Abans d'aplicar el nostre algorisme convindrà doncs comprovar amb temps  $O(|V| + |E|)$  que el graf és connex i comprovar que  $|E| \geq |V|$ . El cost de l'algorisme resultant és asimptòticament el mateix que el de l'algorisme de Kruskal. En cas pitjor  $\Theta(|E| \log |V|)$ .

### Solució 6.30:

- Un arbre de Huffman que minimitzi la llargada d'aquest fitxer és



- Un cop codificat, la llargada del fitxer és  $35.000 \cdot 1 + 1000 \cdot 3 + 2000 \cdot 3 + 2000 \cdot 3 + 1500 \cdot 3 = 54500$  bits

## Algorismes de cerca exhaustiva

**Solució 7.2:** Consulteu les seccions 7.1, 7.2 i 7.3 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>). L'únic que cal fer és substituir les crides al procediment `escriure()` per un increment `++compt` d'un atribut `compt` de la classe `NReines`. La constructora de la classe haurà d'inicialitzar aquest atribut a 0, i s'haurà d'afegir un mètode consultor que permeti obtenir el valor.

**Solució 7.3:** Un cop s'ha trobat la primera solució cal abandonar la cerca. Per això fem que el procediment `recursiu` ens retorni un booleà; `recursiu(i)` retorna cert si  $i$  només si des del node on es fa la invocació hi ha cap solució que sigui accessible. La iteració del nivell  $i$  s'abandona així que s'ha trobat una solució. Per exemple, la solució al problema presentada en la secció 7.1 del document *Algorismes en C++* s'ha de modificar de la següent manera:

```

bool recursiu(int i) {
 if (i == n) {
 escriure();
 return true;
 } else {
 bool trobada_solucio = false;
 for (int j = 0; j < n && !trobada_solucio; ++j) {
 T[i] = j;
 if (legal(i)) {
 trobada_solucio = recursiu(i + 1);
 }
 }
 return trobada_solucio;
 }
}

```

Les solucions millorades que s'expliquen a les seccions 7.2 i 7.3 es modificarien de manera semblant.

**Solució 7.5:** Consulteu la secció 7.4 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>).

**Solució 7.6:** Consulteu la secció 7.7 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>).

**Solució 7.9:** Consulteu les seccions 7.9 i 7.10 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>) per a la solució amb l'esquema de *backtracking*. Consulteu la secció 7.14 per a la solució amb l'esquema de *branch & bound*.

**Solució 7.12:** Consulteu la secció 7.8 del document *Algorismes en C++* (accessible a <http://www.lsi.upc.es/~ada/apunts/programes-apunts.pdf>) per a la solució amb l'esquema de *backtracking*. Consulteu la secció 7.13 per a la solució amb l'esquema de *branch & bound*.

**Solució 7.26:** Para resolver el problema definiremos una clase `concurso` cuya constructora recibe el número  $N$  de bolas y un vector con las  $M$  apuestas. Cada apuesta consiste en un vector que especifica 4 bolas, y una bola es un par  $\langle n, c \rangle$ , siendo  $n$  el número de la bola sobre la que se efectúa la apuesta y  $c$  su color.

La constructora toma nota de los datos del problema, realizará algunas inicializaciones adicionales y finalmente invoca al método recursivo `backtrack`. Al finalizar éste habrá dejado en el atributo booleano `_hi_ha_solucio` el valor cierto o falso, según que el problema tenga o no solución; si hay solución entonces el vector `_solucio` contendrá una solución. En particular, `_solucio[i]` será el color que damos a la bola  $i$  en la solución (la componente 0 del vector no se utiliza).

La estructura general es la siguiente:

```

enum { undef, blanca, negra } color;
struct bola {
 int n;
 color c;
};

// una aposta és un vector que defineix quatre boles
typedef vector<bola> aposta;

class concurs {
public:
 // la constructora rep el vector amb M apostes (correctes)
 concurs(int N, vector<aposta>& apostes) {
 _N = N;
 _apostes = apostes;
 _hi_ha_solucio = false;
 _solucio = vector<color>(_N + 1, undef);
 // otras incializaciones
 backtrack(1);
 }

 bool hi_ha_solucio() { return _hi_ha_solucio; }

 vector<color> solucio() {
 if (_hi_ha_solucio) return _solucio;
 else return vector<color>(_N, undef);
 }

private:
 int _N;
 vector<aposta> _apostes;
 bool _hi_ha_solucio;
 vector<color> _solucio;
 void backtracking(int k);
 // otros atributos y metodos privados
};

```

El algoritmo de **backtrack** se sirve de un método privado **factible(k)** que devuelve cierto si y sólo si ningún apostante acierta su apuesta (las cuatro bolas) con las asignaciones hechas hasta el nivel  $k$ . Por otro lado, tan pronto como encontremos una solución debemos finalizar la ejecución del *backtracking*, por eso sólo se probará a darle el color negro a la bola  $k$  si no se ha encontrado ninguna solución dándole el color blanco a la bola  $k$ . Tendremos la solución buscada cuando  $k = N + 1$  porque eso significa que hemos asignado color a todas las bolas y la combinación es factible.

```

void concurs::backtrack(int k) {
 if (k == _N + 1) {
 _hi_ha_solucio = true;
 }
}

```

```

 return;
 }
 _solucio[k] = blanca;
 if (factible(k))
 backtrack(k + 1);
 if (!_hi_ha_solucio) {
 _solucio[k] = negra;
 if (factible(k))
 backtrack(k + 1);
 }
}

```

Para completar nuestra solución debemos definir cómo es el método **factible**. Obviamente podemos comprobar, para cada uno de los  $M$  apostantes, si alguno de ellos ha acertado su apuesta. Pero podemos hacer que esta función sea mucho más eficiente si observamos que sólo alguien que hizo apuesta sobre la bola  $k$  podría acertar. Por esa razón, nos convendrá saber, para cada bola, de la 1 a la  $N$ , quiénes apostaron por esa bola. En muchos casos, especialmente si  $M \ll N$ , habrá muchas bolas sobre las que nadie o sólo unos pocos apostantes han hecho apuesta, y un recorrido sobre una lista de apostantes de esa bola hará bastante menos de  $M$  iteraciones.

Por otro lado, si para cada jugador tenemos almacenado cuántos aciertos lleva hasta el momento, la función **factible** será todavía más simple y eficiente. Bastará saber si uno de los apostantes de la bola  $k$  ha logrado 4 aciertos. Por otro lado, cuando asignamos a la bola  $k$  un color, sólo los apostantes sobre esa bola tienen un acierto más, los restantes jugadores (que no apostaron nada sobre la bola  $k$ ) se quedan igual.

Ahora bien, la información sobre quién ha apostado a una determinada bola se puede calcular al principio y no cambia a lo largo de la ejecución del algoritmo, mientras que la información sobre el número de aciertos de cada jugador va cambiando según las decisiones que toma sucesivamente el *backtracking* y eso nos obligará a incluir en el algoritmo los pertinentes desmarcajes.

Incluimos en la parte privada la definición de los atributos y métodos que vamos a necesitar.

```

// _aposta_blanca[i] es la lista de jugadores que han apostado que
// la bola i es blanca; análogamente, _aposta_negra[i]
// es la lista de apostantes que la bola i es negra
vector<list<int>> > _aposta_blanca;
vector<list<int>> > _aposta_negra;

// número de aciertos de cada jugador
vector<int> _aciertos(_apostes.size(), 0);

bool factible(int k) const;

// recalcula el numero de aciertos de los jugadores
// con la nueva bola asignada en el nivel k
void marcar(int k);

```

```
// deshace el calculo previo
void desmarcar(int k);
```

Y reescribimos el algoritmo de backtracking para añadir los marcajes y desmarcajes:

```
void concurs::backtrack(int k) {
 if (k == _N + 1) {
 _hi_ha_solucio = true;
 return;
 }
 _solucio[k] = blanca;
 marcar(k);
 if (factible(k))
 backtrack(k + 1);
 desmarcar(k);
 if (!_hi_ha_solucio) {
 _solucio[k] = negra;
 marcar(k);
 if (factible(k))
 backtrack(k + 1);
 desmarcar(k);
 }
}
```

En la constructora hemos de inicializar los vectores con las listas de apostantes, antes de lanzar el backtracking:

```
concurs(int N, vector<aposta>& apostes) {
 ...

 for (int i = 0; i < apostes.size(); ++i) {
 for (int b = 0; b < 4; ++b) {
 bola B = apostes[i][b];
 if (B.c == blanca)
 _aposta_blanca[B.n].push_back(i);
 if (B.c == negra)
 _aposta_negra[B.n].push_back(i);
 }
 }

 backtrack(1);
}
```

Ahora ya sólo nos queda por implementar los tres métodos privados, que casi se escriben solos, dadas las definiciones de las estructuras de datos que utilizamos para los marcajes:

```
void concurs::marcar(int k) {
 list<int>::iterator ap;
 switch (_solucio[k]) {
 case blanca : {
 // solo los apostantes que la bola k era blanca tienen un
 // acierto mas
```



```
 for (ap = _aposta_blanca[k].begin();
 ap != _aposta_blanca[k].end(); ++ap)
 ++_aciertos[*ap];
 break;
 }
 case negra : {
 // solo los apostantes que la bola k era negra tienen un
 // acierto mas
 for (ap = _aposta_negra[k].begin();
 ap != _aposta_negra[k].end(); ++ap)
 ++_aciertos[*ap];
 break;
 }
}

bool concurs::factible(int k) {
// miramos si alguno de los apostantes de la bola k tiene 4 aciertos
list<int>::iterator ap;
switch (_solucio[k]) {
case blanca : {
 for (ap = _aposta_blanca[k].begin();
 ap != _aposta_blanca[k].end(); ++ap)
 if (_aciertos[*ap] == 4) return false;
 // nadie tiene 4 aciertos todavia
 return true;
 break;
}
case negra : {
 for (ap = _aposta_negra[k].begin();
 ap != _aposta_negra[k].end(); ++ap)
 if (_aciertos[*ap] == 4) return false;
 // nadie tiene 4 aciertos todavia
 return true;
 break;
}
}
}

void concurs::desmarcar(int k) {
list<int>::iterator ap;
switch (_solucio[k]) {
case blanca : {
 // solo los apostantes que la bola k era blanca tienen un
 // acierto menos
 for (ap = _aposta_blanca[k].begin();
 ap != _aposta_blanca[k].end(); ++ap)
 --_aciertos[*ap];
 break;
}
```

```

 }
 case negra : {
 // solo los apostantes que la bola k era negra tienen un
 // acierto menos
 for (ap = _aposta_negra[k].begin();
 ap != _aposta_negra[k].end(); ++ap)
 --_aciertos[*ap];
 break;
 }
}
}

```

**Solució 7.28:** La nostra solució de tornada enrera trobarà el subconjunt de  $n$  valors de les  $D_j$ 's que constitueixin una solució, si n'hi ha. Ordenarem el vector de  $D$ 's amb aquesta finalitat. Farem servir marcatge: d'una banda, el paràmetre  $l$  de la funció recursiva `backtrack` serà l'índex de la primera distància a provar en el nivell  $k$ . Així doncs, al nivell  $k$  provarem les decisions  $x_k = D_l$ ,  $x_k = D_{l+1}$ , ... Donat que el vector està ordenat, les  $x$ 's sortiran ordenades també. D'altra banda, farem servir un vector de booleans `usat`, de manera que `usat[j]` serà cert en el nivell  $k$  si la distància  $D_j$  correspon a un parell de  $x$ 's ja definides, es a dir,  $D_j = x_m - x_p$ , amb  $0 \leq p \leq m < k$ . Per saber si una decisió  $x_k := D_i$  n'hi haurà prou amb comprovar que es marquen  $k$  distàncies com a "usades". I quan  $k = n$ , ja tindrem una solució.

```

void distancias(int k, int n, vector<int>& Dorig,
 vector<int>& x, bool& hi_ha_sol) {

 // fem una còpia del vector original i l'ordenem
 // ascendenment i definim el vector de marcatge
 vector<int> D(Dorig);
 sort(D.begin(), D.end());
 vector<bool> usat(D.size(), false);

 x = vector<int>(n); x[0] = 0;

 hi_ha_sol = backtrack(1, 0, n, D, x, usat);
}

// k és el nivell d'exploració: estem prenent decisió sobre x[k]
// l és l'índex de la primera distància disponible,
// i.e., haurem de provar $x[k] = D[l], D[l+1], \dots$
// usat[j] == true ssi D[j] és una distància entre les x's ja definides

bool backtrack(int k, int l,
 int n, const vector<int>& D,
 vector<int>& x,
 vector<bool>& usat) {

 if (k == n) // es solucio

```

```

 return true;
else
 for (int i = 1; i < D.size(); ++i) {
 x[k] = D[i];

 // marcatge
 // la nova decissió ha de marcar com usades k distàncies
 int cont = 0;
 for (int p = 0; p < k; ++p) {
 int d = x[k] - x[p];
 bool found = false;
 for (int j = 0; j < D.size() && !found; ++j)
 found = D[j] == d && !usat[j];
 if (found) {
 usat[j] = true;
 ++cont;
 } else {
 break;
 }
 }

 if (cont == k) // és factible
 if (backtrack(k + 1, i + 1, n, D, x, usat));
 return true;

 // desmarcatge
 for (int p = 0; p < k; ++p) {
 int d = x[k] - x[p];
 bool found = false;
 for (int j = 0; j < D.size() && !found; ++j)
 found = D[j] == d && !usat[j];
 if (found)
 usat[j] = false;
 }
 }
 return false;
}

```

**Solució 7.30:** `typedef vector<vector<bool> > matriu_adj;`  
`class Antenes {`  
`private:`  
 `matriu_adj M; // matriu d'adjacències`  
 `int n; // núm. de ciutats`  
  
 `vector<bool> x; // solució en curs`  
 `// x[i] == true ssi la ciutat 'i' té antena`

```

int talla_vertex_cov; // núm. d'antenes posades fins al moment
vector<bool> cov; // cov[i] == true ssi la ciutat 'i' està
 // coberta per la solució en curs
 // cov és part del "marcatge"

vector<bool> min_vertex_cov; // millor solució
int talla_min_vertex_cov; // núm. de ciutats en la millor
 // solució

void backtrack(int k);
int totes_cobertes() const;
public:
Antenes(const matriu_adj& _M) {
 M = _M;
 n = _M.size();

 // comencem amb una solució parcial sense cap antena
 x = vector<bool>(n, false);
 cov = vector<bool>(n, false);
 talla_vertex_cov = 0;

 // posar antenes a totes les ciutats és una solució
 // possible, probablement lluny de l'òptim
 min_vertex_cov = vector<bool>(n, true);
 talla_min_vertex_cov = n;

 backtrack(0);
}
...
}

```

El nostre backtracking explora un arbre binari, on a cada nivell decidim si a la ciutat  $k$ -èssima posem ( $x[k] == \text{true}$ ) o no ( $x[k] == \text{false}$ ) una antena. El criteri per a saber si una solució parcial és o no solució és senzill: només cal comprovar si totes les  $n$  ciutats estan cobertes o no. L'atribut `cov` és part del marcatge i és molt útil per evitar càlculs redundants. Només quan prenem la decissió  $x[k] = \text{true}$  cal marcar les veïnes de la ciutat  $k$  com a cobertes (i la pròpia  $k$ , és clar). Qualsevol subconjunt de vèrtexos és factible, però podem podar el backtracking si la solució en curs conté més antenes que la millor solució que haguem trobat fins al moment.

```

void Antenes::backtrack(int k) {
 if (k < n) {
 vector<bool> cov_copy = cov;

 x[k] = true;
 cov[k] = true;
 ++talla_vertex_cov;
 for (int i = 0; i < n; ++i)
 cov[i] = cov[i] or M[k][i];
 if (totes_cobertes() &&

```

```
 talla_vertex_cov < talla_min_vertex_cov) {
 talla_min_vertex_cov = talla_vertex_cov;
 min_vertex_cov = x;
} else if (talla_vertex_cov < talla_min_vertex_cov) {
 backtrack(k + 1);
}

x[k] = false;
cov = cov_copy;
--talla_vertex_cov;
// segur que no es cobreix tot doncs ara no
// hem afegit cap antena a la solució en curs
if (talla_vertex_cov < talla_min_vertex_cov)
 backtrack(k + 1);
} }

// retorna cert ssi totes les n ciutats estan cobertes
int Antenes::totes_cobertes() {
 for (int i = 0; i < n && cov[i]; ++i);
 return i == n;
}
```

## Introducció a la NP-completesa

### **Solució 8.1:** Escena 1: graf-hamiltonià

- els vèrtexos són les persones
- les arestes són les parelles de persones no enfadades mútuament

### **Escena 2:** 3-colorabilitat

- els vèrtexos són les persones
- les arestes són les parelles de persones enfadades mútuament

### **Escena 3:** conjunt-dominador

- els vèrtexos són les persones
- les arestes són les parelles de persones no enfadades mútuament
- $k$  ve donat

### **Escena 4:** graf-hamiltonià

- els vèrtexos són les places on hi ha bons restaurants
- les arestes són els carrers que les uneixen

### **Escena 5:** conjunt-dominador

- els vèrtexos són les places on hi ha bons restaurants
- les arestes són els carrers que les uneixen i els que uneixen tres places consecutives
- $k = m$

**Escena 6:** bin-packing

- les  $n$  peces són les persones, on la mida de cada peça és el temps que triga la persona a sopar
- els  $k$  contenidors són les cadires
- $m$  és el temps que resta fins que tanquin el restaurant

**Escena 7:** tripartite-matching

- els conjunts són les persones, les persones i els colors de les taules
- el subconjunt ve donat per les preferències

**Escena 8:** graf-hamiltonià

- els vèrtexos són les persones
- hi ha un arc  $(i, j)$  si i només si la persona  $i$ -èssima pot acabar de sopar abans no hagi de començar a sopar la persona  $j$ -èssima

**Solució 8.2:** Sigui  $C_1, \dots, C_n$  una col·lecció de  $n$  nombres enters positius i sigui  $S$  un nombre enter positiu. Sigui també  $C = C_1 + \dots + C_n$  i considerem la instància del problema de la PARTICIÓ  $C_1, \dots, C_n, A, B$ , on  $A = 2C - S$  i  $B = C + S$ . La suma total dels elements d'aquesta instància és  $C + 2C - S + C + S = 4C$  i existeix una partició de  $C_1, \dots, C_n, A, B$  en dos subconjunts  $S_1$  i  $S_2$  que sumen exactament  $2C$  si i només si existeix un subconjunt de  $C_1, \dots, C_n$  que suma exactament  $S$ .

Per mostrar que PARTICIÓ pertany a NP, basta considerar com a certificat els dos subconjunts  $S_1$  i  $S_2$ , que són de talla polinòmica en la talla  $n$  del conjunt  $C_1, \dots, C_n$  donat i aquest certificat es pot verificar en temps polinòmic: només cal comprovar que  $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ .

**Solució 8.3:** Sigui  $G = (V, E)$  un graf no dirigit i  $k$  un natural. Construïm un  $k$ -clique  $G_1$  i un altre graf no dirigit  $G_2 = G$ . Aleshores,  $G$  conté un clique de talla  $k$  si i només si  $G_1$  és un subgraf induït de  $G_2$ .

Per mostrar que SUBGRAF-INDUÏT pertany a NP, siguin  $G_1 = (V_1, E_1)$  i  $G_2 = (V_2, E_2)$  i considerem com a certificat una funció  $f : V_1 \rightarrow V_2$ . Aquest certificat és de talla polinòmica en la talla de  $G_1$  i es pot verificar en temps polinòmic: només cal comprovar que per a tot  $u_1, v_1 \in V_1$ , es compleix  $\{u_1, v_1\} \in E_1$  si i només si  $\{f(u_1), f(v_1)\} \in E_2$ .

**Solució 8.14:** • Ja que  $\text{SAT} \leq_m X$  l'únic que podem deduir és que  $X$  és tan difícil o més que SAT, però no sabem si  $X$  està a la classe NP o no. La resposta és (c).

- Com que  $X$  es redueix a SAT i ja que aquest es redueix a  $X$ , tots dos problemes són equivalents. La resposta correcta és (ii).

**Solució 8.15:** La resposta correcta és la (b).