Anàlisi i Disseny d'Algorismes (ADA)

Col·lecció de Problemes

18 d'octubre de 2007

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 Algorismes de dividir i vèncer **11**

3 Estructures de dades **19**

4 Grafs **33**

5 Solucions **41**

6 Solucions (professors) **63**

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 es presentarà en classe (però, intenteu resoldre'ls vosaltres mateixos abans de mirar les solucions!).

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

Capítol 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 $x \in \{0, ..., 100\}$ i per a $x \in \{0, ..., 10000\}$.

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 $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 n^0$ 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 $\mathrm{H}_n = \sum_{i=1}^{n} 1/i$. Mostreu que $\mathrm{H}_n = \Theta(\log n)$ aproximant per integrals contínues.

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

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 novela "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 novela?

11. Existeixen diferents versions sobre l'orígen 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 contingués el doble de grans que l'anterior. A la primera casella va demanar que hi col·loquéssin 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 $f(n)$ μs 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 $\Theta(n)$. Suposeu 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 dígits,
    - Multiplicar dos naturals de n dígits.

✎ 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.

Eratóstenes

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'Eratóstenes 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 subtractores 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: Suposeu 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. Suposeu 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?

    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 Fibonnaci 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
};
```

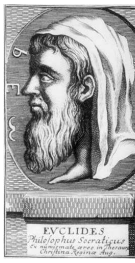
```
int comptar_as (cadena& c) {
    int as = 0;
    for (int i=0; i<c.n; ++i) {
        if (c.T[i]=='A') {
            ++as;
    }   }
    return as;
}
```

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.

Capítol 2

Algorismes de dividir i vèncer

1. Escriviu un algorisme de cost $\Theta(\log n)$ que, donada una taula ordena 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 ordena 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 es 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

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

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

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

☆☆ 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:

a) $\{ i \in 1 \ldots n \ : \ |\{ j \ : \ j \neq i \wedge t[j] < t[i] \}| \ = \ i \ \}$

b) $\{ i \in 1 \ldots n \ : \ |\{ j \ : \ j \neq i \wedge t[j] < t[i] \}| \ = \ t[i] \ \}$

c) $\{i \in 1 \ldots n \ : \ |\{j \ : \ j < i \wedge t[j] < t[i]\}| \ = \ i \ \}$
d) $\{i \in 1 \ldots n \ : \ |\{j \ : \ j < i \wedge t[j] < t[i]\}| \ = \ t[i] \ \}$
e) $\{i \in 1 \ldots n \ : \ |\{j \ : \ j < i \wedge t[j] < t[i]\}| \ = \ |\{j \ : \ j > i \wedge t[j] < t[i]\}| \ \}$
f) $\{i \in 1 \ldots n \ : \ |\{j \ : \ j < i \wedge t[j] < t[i]\}| \ = \ |\{j \ : \ j < i \wedge t[j] > t[i]\}| \ \}$

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 estrictament creixent. Dissenyeu un algorisme $\mathcal{O}(\log n)$ que retorni un índex i tal que $V[i] = i$ si aquest existeix, -1 altrament.

    Repetiu el mateix problema suposant que el vector pot contenir elements repetits.

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 analitzeune 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 2×2 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

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

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 aplicaríeu 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 $A[2 \,..\, 6]$. Formalment, es vol calcular

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

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.

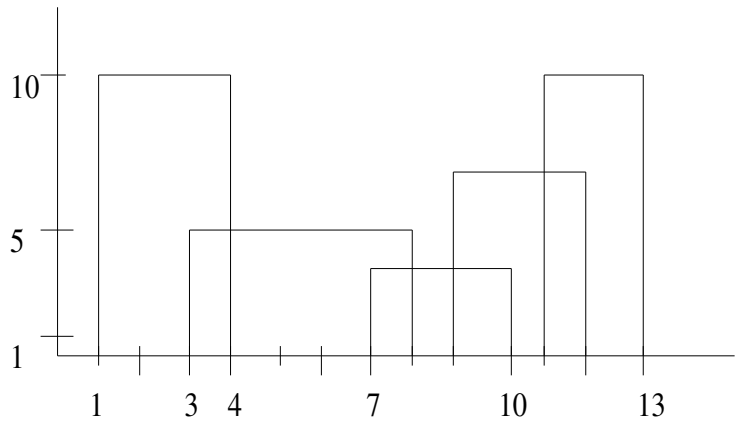
☆☆ Escriviu un algorisme $\Theta(n)$ per resoldre el mateix problema.

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

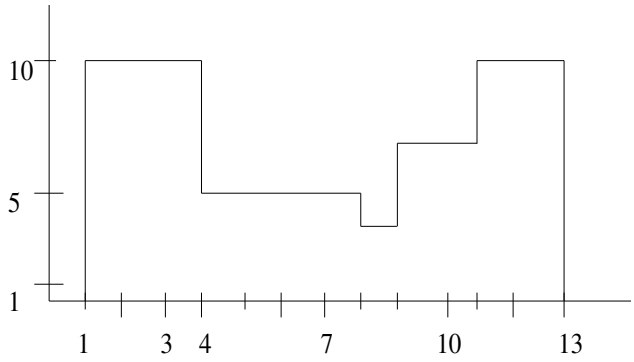
$$d = \min_{\substack{1 \le i,j \le n \\ i \ne j}} \{|v_i - v_j|\}.$$

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*, linea 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 $v = (v_0, \ldots, v_{n-1})$ amb n valors reals, dissenyeu un algorisme que calculi quant val la diferència d entre els dos valors més propers:

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

✎ 33. Sigui $a[0..n-1]$, $n > 0$, un vector d'enters diferents i ordenats creixentment. Dissenyeu un algorisme que retorni un índex k, $0 \leq k < n$, tal que $a[k] = k$ si existeix un índex que satisfà la condició. El cost del vostre algorisme ha de ser $O(\log n)$ en cas pitjor.

☆ 34. Dissenyeu 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 $1 \leq j[1] < j[2] < \ldots < j[p] \leq n$, cal trobar els elements $j[1]$-èsim, $j[2]$-èsim, $\ldots$, $j[p]$-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 $(2, 7, 14)$.

35. 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 procedimient a dissenyar, un cop feta la inmersió de par àmetres, 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 vuestre algorisme sigui molt detallat i que es dissenyin totes les funcions auxiliars que s'usin.

✎ 36. Cal organitzar l'horari d'un campeonat 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 campeonat en $n - 1$ dies. Analitzeu el cost de l'algorisme.

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

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

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 elementos més grans o iguals que x és $\ge 1/2$. 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.

38. 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. Suposeu 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:

    (a) Primer ordenem V amb *quicksort* usant el primer cognom, després amb *mergesort* usant el segon cognom.

    (b) Primer ordenem V amb *mergesort* usant el primer cognom, després amb *quicksort* usant el segon cognom.

    (c) Primer ordenem V amb *quicksort* usant el segon cognom, després amb *mergesort* usant el primer cognom.

    (d) Primer ordenem V amb *mergesort* usant el segon cognom, després amb *quicksort* usant el primer cognom.

39. Recordeu que una inversió en una taula $T[1 \ldots n]$ 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 $T[1 \ldots n]$ que només té una inversió i un element x, digui si x és a T.

Capítol 3

Estructures de dades

Taules de dispersió

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

   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 $h(x) = x \mod 11$.

   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 $h(x) = x \mod 10$.

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 $h(x) = x \mod 11$.

   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
```

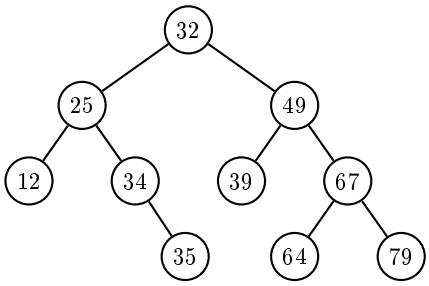
La instrucció `RENUM` renumera les instruccions del programa de forma que les linies vagin de 10 en 10. Per exemple, després de renumerar el programa anterior queda:

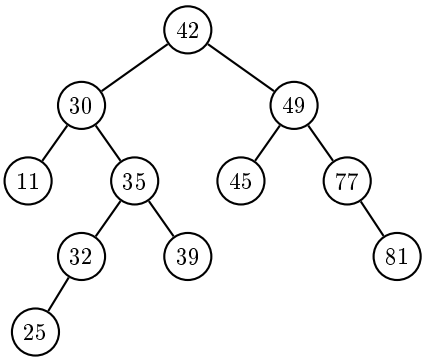
```
10 INPUT N
20 LET F = 1
30 LET I = 1
40 IF I=N THEN GOTO 80
50 LET F = F*I
60 LET I = I+1
70 GOTO 40
80 PRINT F
```

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

Arbres binaris de cerca

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

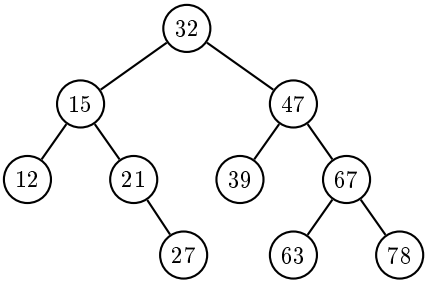




8. Partint d'un arbre binari de cerca buit, inseriu amb l'algorisme clàssic, l'una rera l'altra, la seqüència de claus $32, 15, 47, 67, 78, 39, 63, 21, 12, 27$.

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 arr
                         // (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 afegeixi 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?

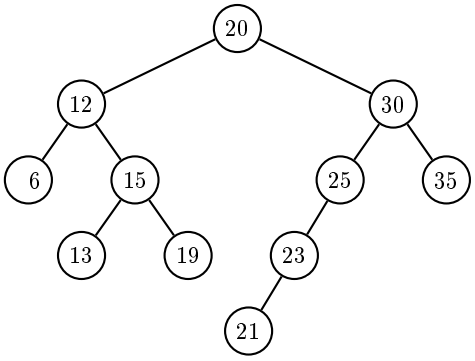
17. Implementeu una operació `Elem minim (abc a)` que retorni l'element més petit de l'arbre binari de cerca no buit `a`. Quin és el seu cost?

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.

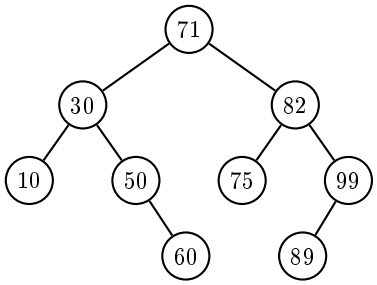
✎ 21. Implementeu una funció `int menors (abc a, Elem x)` que retorni el nombre d'elements de l'arbre binari de cerca `a` que són estrictament inferiors a `x`. Quin és el seu cost?

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.

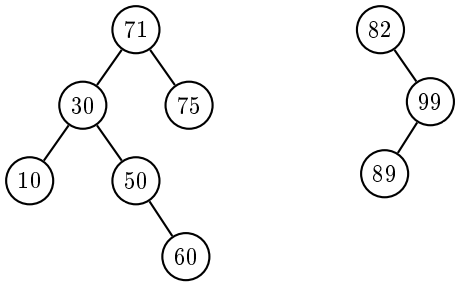
23. Implementeu la funció `abc reconstrueix (vector<Elem> pre)` que reconstrueix un arbre binari de cerca a partir del seu recorregut en preordre emmagatzemat en un vector `pre`. Quin és el seu cost?

24. Implementeu una funció `void separa (abc& a, Elem x, abc& le, abc& gt)` que donat una 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

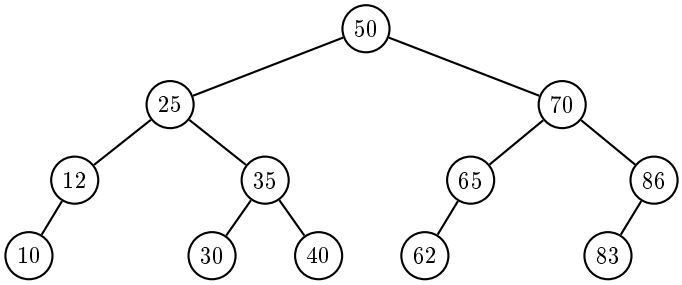


i l'element $x = 75$, els arbres `le` i `gt` són els següents:

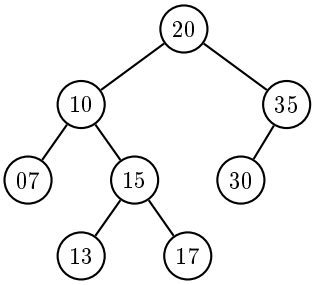


Arbres AVL

✎ 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):



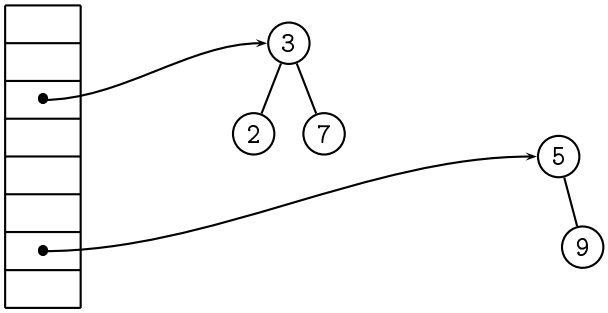
27. Dibuixeu un arbre AVL d'alçada màxima que tingui les claus 1, 2, . . . , 12.

Altres diccionaris

28. Suposeu 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 usarieu?

    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. Suposeu 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:

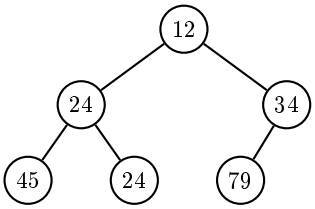


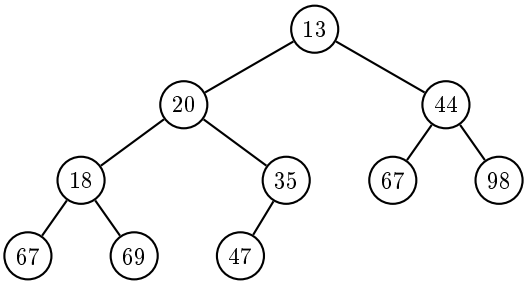
Suposeu que la taula té M posicions i n claus emmagatzemades, amb $M = \Theta(n)$. Considereu el cost de buscar una clau.

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

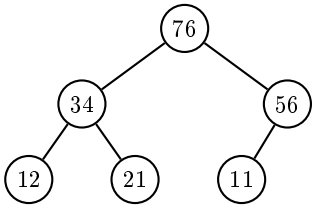
Heaps

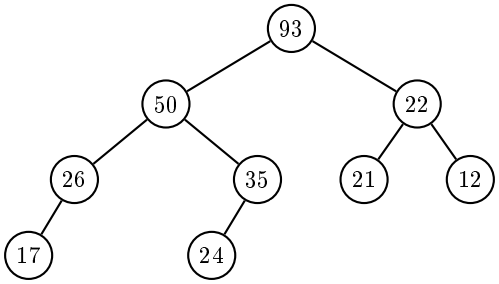
30. Digueu si els arbres binaris següents són min-heaps o no i perquè.





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

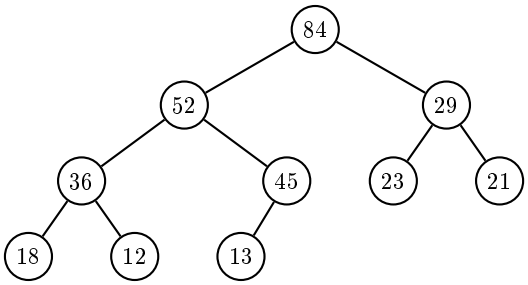




32. Partint d'un min-heap buit, inseriu successivament les claus 45, 67, 23, 46, 89, 65, 12, 34, 98, 76.

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.

Cues de prioritat

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; // △
            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␣plena.");
        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 seqüencia d'intercanvis de la línia △.

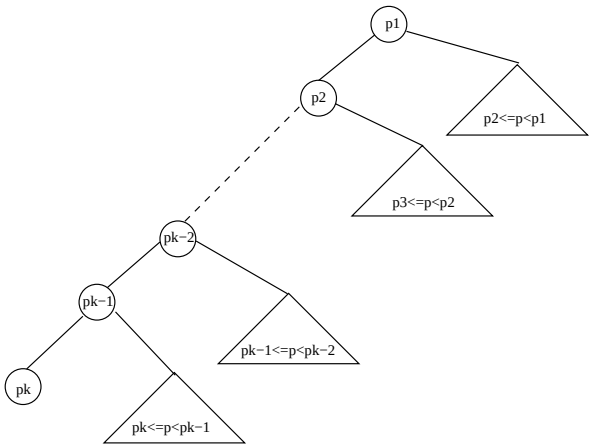
(c) Utilitzeu la posició 0 de la taula com a sentinella per eliminar la condició `i!=1` de la cerca.

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

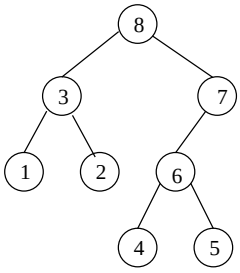
```
priority_queue<int> CP;
int x;
while (cin >> x) {
    CP.push(x);
}
while (not CP.empty()) {
    cout << CP.top() << endl;
    CP.pop();
}
```

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 estàn 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) Escriviu 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.

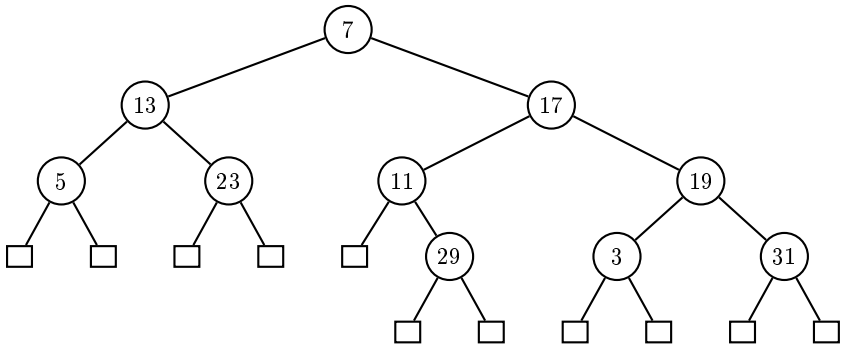
46. Dissenyeu una estructura de dades per al conjunt dels enters amb les operacions i els costos temporals en cas pitjor següents. Podeu suposar que es coneix una fita superior M del nombre d'elements del conjunt.

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

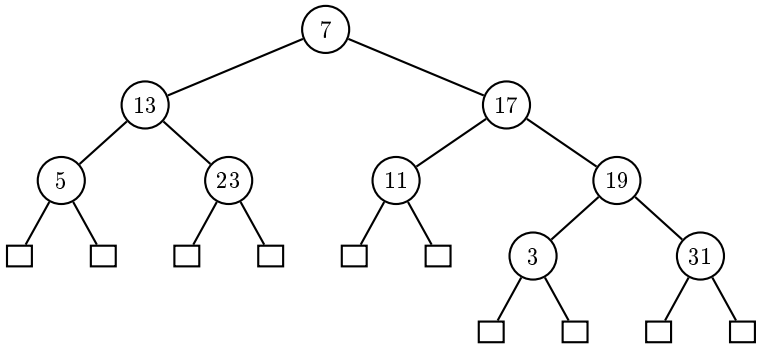
47. L'Édgar Gonzàlez (Ex-becari d'EDA, classificat 15è millor programador del món al Concurs de Programació ACM-IBM 2004) és un *freaky* a qui li agrada usar l'expressió "arbre caducifoli".

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



obtindrem 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 precondició 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.

Heapsort

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?

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.

57. Repetiu el problema anterior per a la seqüencia següent:

    `crear(10)`, `unir(1,2)`, `unir(2,3)`, `unir(3,4)`, `unir(4,5)`, `unir(5,6)`, `unir(6,7)`, `unir(7,8)`, `unir(8,9)`, `unir(8,10)`, `representant(1)`.

58. Considereu la classe següent per a particions amb claus entre 0 i $n-1$.

```
class Particio {
public:
    explicit Particio (int n);
      // Constructor. Crea una partició per als elements de 0 a n.
      // Cada enter es troba en una classe d'equivalència que
      // només el conté a ell.

    Particio (Particio& p);              // Constructor de còpia
    Particio& operator= (Particio& p);   // Operador d'assignació
    ~Particio ();                        // Destructor

    void unir (int x, int y);     // Uneix les classes de x i de y
    int representant (int x);     // Ret. representant canònic de
                                  // la classe de x
    int talla ();                 // Ret. la talla de la partició
    int operator[] (int x);       // Embelliment de representant
};
```

Implementeu la classe anterior tractant tots els possibles errors i analitzeu el cost de les operacions públiques. Feu-ne cinc versions:

- Implementant l'algorisme bàsic.
- Implementant l'algorisme amb unió per talla.
- Implementant l'algorisme amb unió per alçada.
- Implementant l'algorisme amb unió per talla i compressió de camins.
- Implementant l'algorisme amb unió per alçada i compressió de camins.

59. Implementeu de la forma el més eficient possible en el cas pitjor la classe següent per a particions amb cadenes com a claus.

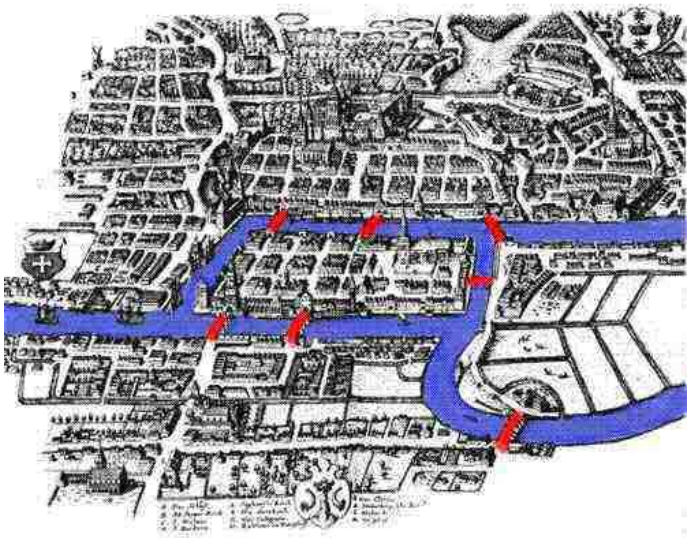
```
class Particio {
public:
    explicit Particio ();                  // Constructor
    Particio (Particio& p);                // Constructor de còpia
    Particio& operator= (Particio& p);     // Operador d'assignació
    ~Particio ();                          // Destructor

    void unir (string x, string  y);       // Uneix les classes de x i de
    string representant (string x);        // Representant canònic de
                                           // la classe de x
    int talla ();                          // Retorna la talla de la parti
                                           //     (nombre de cadenes)
};
```

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

Capítol 4

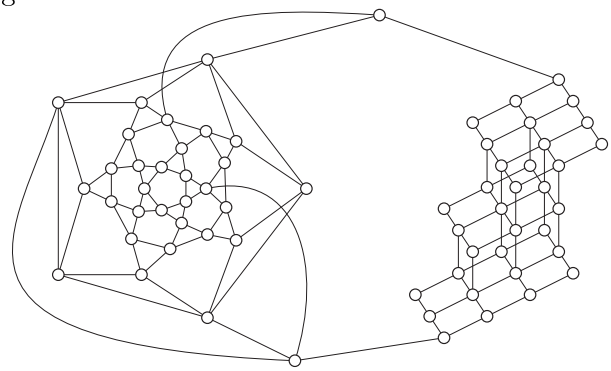
Grafs

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:

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

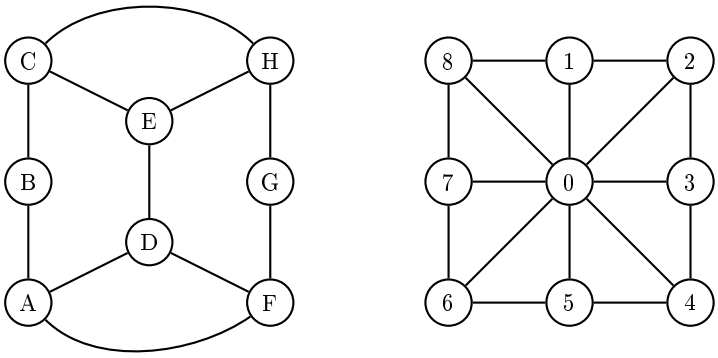
És connex? Si no ho és, quants components connexos té?

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

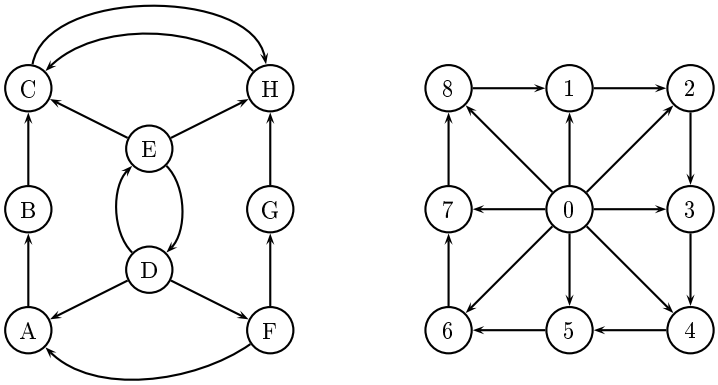
   - $V = \{1, 2, 3, 4, 5\}$
   - $E = \{(1, 2), (1, 4), (3, 2), (4, 5), (5, 1), (5, 2)\}$

   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.



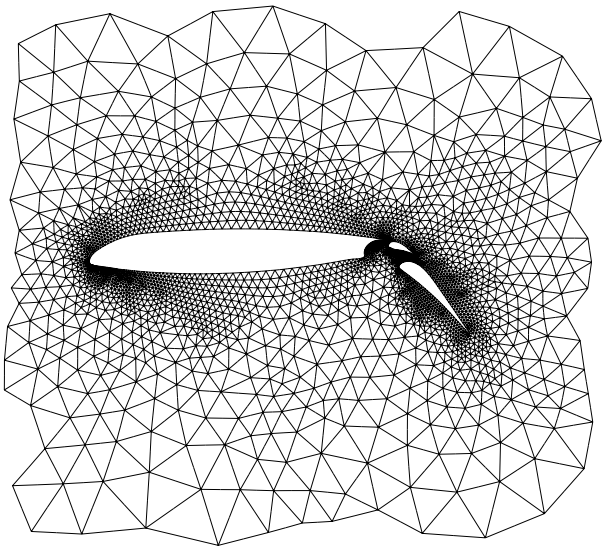
6. Mostreu com es representen els grafs dirigits següents utilitzant una matriu d'adjacència i llistes d'adjacència. Compteu el grau d'entrada i de sortida de cada vèrtex. Digueu si els grafs són fortament o feblement connexos.



✎ 7. Demostreu que en un graf no dirigit $G = (V, E)$ es té $\sum_{u \in V} \text{grau}(u) = 2|E|$.

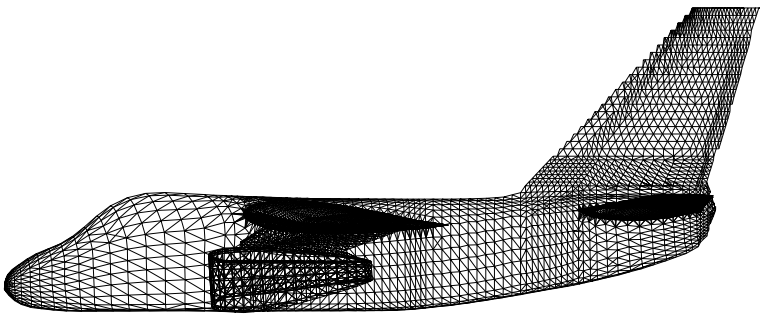
8. Considereu un graf no dirigit amb cinc vèrtexs, tots de grau tres. Si existeix tal graf, feu-ne un dibuix. Si no n'existeix cap, doneu una explicació enraonada.

9. El graf de la figura següent té $n = 4253$ vèrtexs i $m = 12289$ arestes.

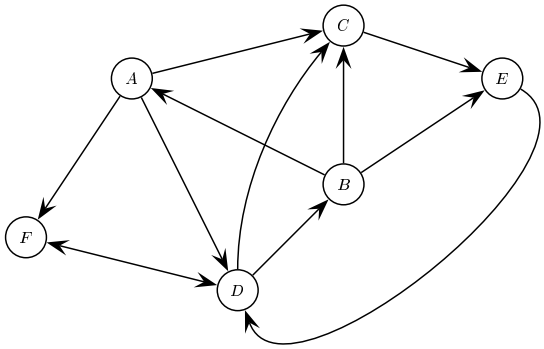


Suposeu que al vostre ordinador cada booleà ocupi un byte, cada enter quatre i cada punter quatre. Digueu quanta memòria cal per emmagatzemar aquest graf utilitzant una representació amb matriu d'adjacència i utilitzant una representació amb llistes d'adjacència.

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 amplada que comenci en el vèrtex B.

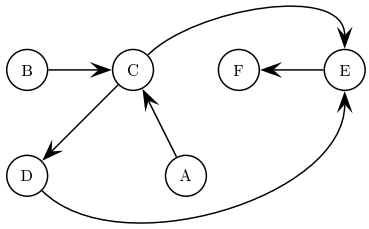
13. (a) Dissenyeu i analitzeu un algorisme per calcular el grau d'un vèrtex donat en un graf no dirigit representat amb una matriu d'adjacència.
    (b) Dissenyeu i analitzeu un algorisme per calcular el grau d'un vèrtex donat en un graf no dirigit representat amb llistes d'adjacència.
    (c) Dissenyeu 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.

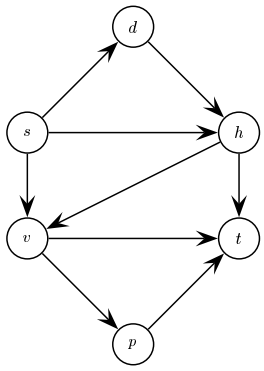
✎ 24. Una *ordenació topològica inversa* d'un graf dirigit acíclic és una seqüència formada per tots els vèrtexs del graf tal que si (v, w) és una aresta del graf, llavors w apareix abans que v a la seqüència. A partir del recorregut en profunditat, dissenyeu i analitzeu un algorisme que obtingui una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

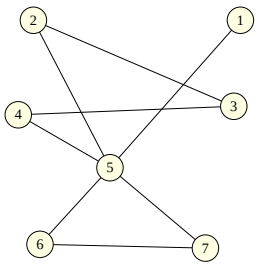
☆☆ 31. Sigui $G = (V, E)$ un graf no dirigit complet. Sigui $G' = (V, E')$ un graf dirigit en el que E' conté les mateixes arestes que E però assignant una orientació arbitraria a cadascuna d'elles. Demostreu que G' conté com a mínim un camí Hamiltonià. Dissenyeu i analitzeu un algorisme que calculi 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.

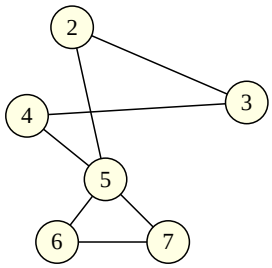
33. Donat un graf $G = (V, E)$ no dirigit amb $n = |V|$, dissenyeu un algorisme que donada una constant natural $k \geq 0$ calculi el *subgraf induït màxim* $H = (U, F)$, si existeix. El subgraf indu ït màxim H ha de complir que tots els seus vèrtexos tinguin grau més gran o igual que k. Feu-ne una anàlisi de l'eficiència.

Ajut: Comenceu l'anàlisi resolent aquests tres casos: $n \leq k$, $n = k + 1$ y $n > k + 1$.

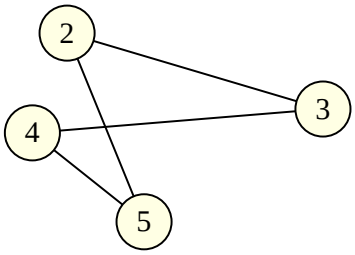
Per exemple, donat el graf següent:



el subgraf induït màxim quan $k = 2$ és



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



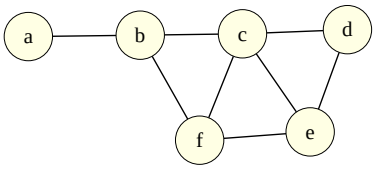
☆☆ 34. Sigui $G = (V, E)$ un graf no dirigit i connex. Dissenyeu i implementeu un algorisme eficient per determinar un *conjunt base dels cicles simples* del graf. Cal proporcionar les arestes que formen cadascun dels cicles simples. Noteu que una aresta pot pertànyer a m és d'un cicle simple. Per facilitar 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 $S = \langle v_1, v_2, \ldots, v_i, \ldots, v_r \rangle$, $|S| \geq 4$ 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 todes les arestes de G que formen part d'algún cicle.

Per exemple, donat el graf de la figura següent,



un possible conjunt base seria

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

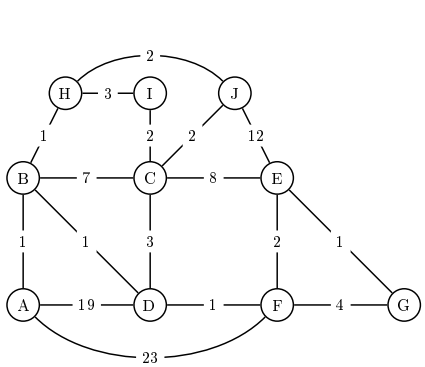
☆☆ 35. Donat un graf dirigit $G = (V, E)$ i dos dels seus vèrtexos p i q, dissenyeu un algorisme que retorni una llista amb tots els camins elementals que van de p a q en G. Una bona versió recorre cada aresta un sol cop. Raoneu la seva correctesa i analitzeu-ne el cost. Recuerdeu que un camí elemental és aquell en què no es repeteixen vèrtexos.

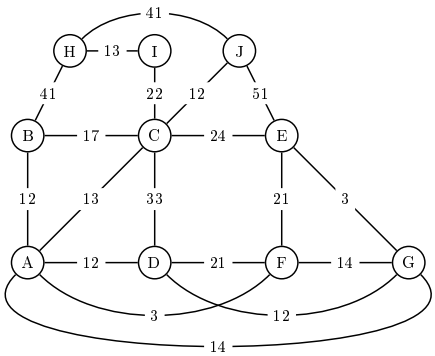
☆☆ 36. Dissenyeu un algorisme eficient per trobar el que es denomina *enllaços crítics* en una xarxa d'ordinadors, és a dir, aquelles connexions entre nodes que en cas d'avaria deixarien parts de la xarxa desconnectades de la resta. No hi ha més d'un enllaç entre dos nodes qualssevol de la xarxa, però si cap enllaç no falla, la xarxa és connexa. Calculeu l'eficiència de l'algorisme en termes del número de nodes n Iy 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, ...

☆☆ 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. Dissenyeu 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 cicle 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.

Capítol 5

Solucions

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 $k + 1$:

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

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 $f(n)$ μs, la talla més gran que es pot processar en t μs 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 $= 10^6 \mu$s, 1 minut $= 6 \cdot 10^7$ μs, 1 hora $= 3.6 \cdot 10^9$ μs, 1 dia $= 8.64 \cdot 10^{10}$ μs, 1 mes $= 2.592 \cdot 10^{12}$ μs, 1 any $= 3.1536 \cdot 10^{13}$ μs i 1 segle $= 3.1536 \cdot 10^{15}$ μs. 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

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

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

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

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 $\alpha = \log_{\frac{10}{9}} 1 = 0 < k$ 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

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

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 $T_A(n) = \Theta(n^\alpha) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\ldots})$.

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 $n \log n = o(n^{\log_2 3})$.

Algorismes de dividir i vèncer

Solució 2.2:

- 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 2.5: 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] ≤ x < v[j], v ordenado creciente
// Post: k = busqueda_ternaria(v, i, j, x) ⟹ v[k] ≤ 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 ≤ 2, v[i] ≤ 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 = 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:

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

de manera que aplicando el *master theorem* para recurrencias de divide y vencerás obtenemos que $T(n) = \Theta(\log n)$. Es decir, asíntoticamente 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 2.18: 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($A, 0, A.\text{size}() - 1, num_inv$)
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 2.19: 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

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

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

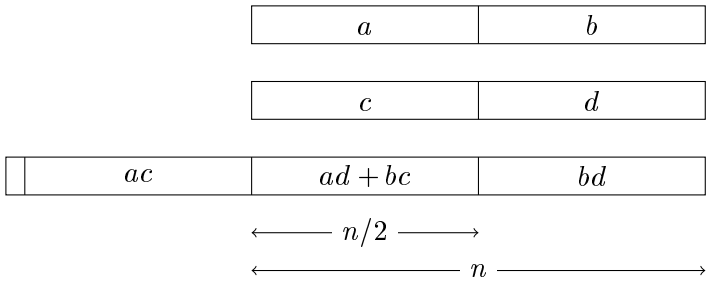
```
int contar(vector<int>& t) {
    sort(t.begin(), t.end());
    vector<int> sumparc(t.size() + 1);
    sumparc[0] = 0;
    for (int i = 1; i < sumparc.size(); ++i)
        sumparc[i] = sumparc[i-1] + t[i-1];

    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 2.27: 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 $a \cdot d + b \cdot c$. Aquest operacions tenén 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 vé 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.

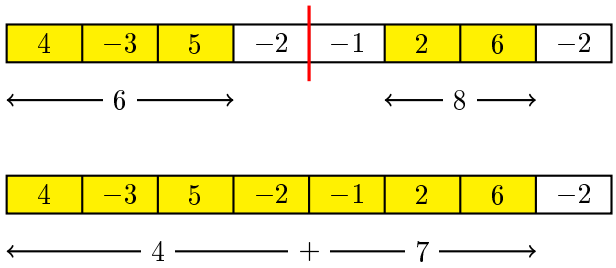
Solució 2.28: a) Aquest algorisme requereix 7 crides recursives al producte de matrius i 17 sumes/restes de matrius. La part no recursiva de l'algorisme té cost $\Theta(n^2)$; llavors el

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




```
function MAXSUM(A, ℓ, r)
    if ℓ > r then
        return 0
    else if ℓ = r then
        return (A[ℓ] > 0)?A[ℓ] : 0
    else
        m := ⌊(ℓ + r)/2⌋
        lmax := 0
        sum := 0
        for i := m downto ℓ do
            sum := sum + A[i]
            if sum > lmax then lmax := sum
            end if
        end for
        rmax := 0
        sum := 0
        for i := m + 1 to r do
            sum := sum + A[i]
            if sum > rmax then rmax := sum
            end if
        end for
        return max(maxsum(ℓ, m), maxsum(m + 1, r),    lmax + rmax)
    end if
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 = σ_{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 = σ_{i,j-1}
          sum += A[j];
          // sum = σ_{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 segmeto 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 cúal 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 ℓ' su extremo inferior (su otro extremo es, por definición, m).

Análogamente, deberemos averiguar cúal 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; ++r) {
          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 $A[\ell..u]$ 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 $A[\ell..u]$ teniendo en cuenta lo siguiente:

- El segmento de suma máxima será el correspondiente a

$$\max(summax1, summax2, sder1 + sizq2)$$

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

$$\max(sder2, sder1 + sum2)$$

- El segmento de suma máxima con extremo inferior en ℓ será el correspondiente a

$$\max(sizq1, sum1 + sizq2)$$

- La suma del segmento es $sum1 + sum2$.

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

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 2.33: 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 2.36: 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 dia d; al menos tiene que haber
// dos jugadores; i <= j-1; el torneo dura n - 1 dias 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 poar 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 $C(n/2) + \Theta(1) = \Theta(n^2)$ ($a = 2, b = 2, k = 2, \alpha = \log_b a = 1 < k = 2$).

Estructures de dades

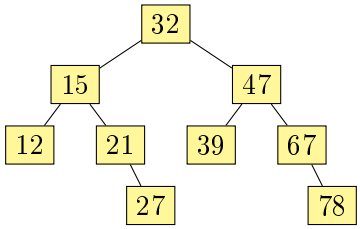
Solució 3.1:

```
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
```

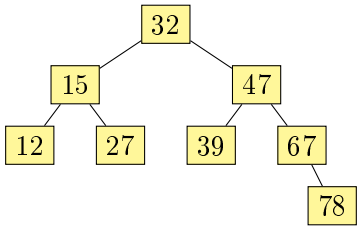
Solució 3.3:

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	128181	7812	1299	999	18267			

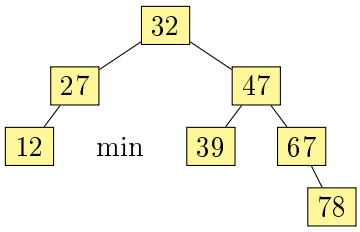
Solució 3.10: 1. Eliminació de la clau 63:



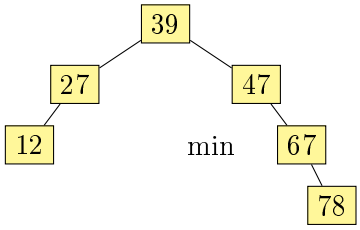
2. Eliminació de la clau 21:



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



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

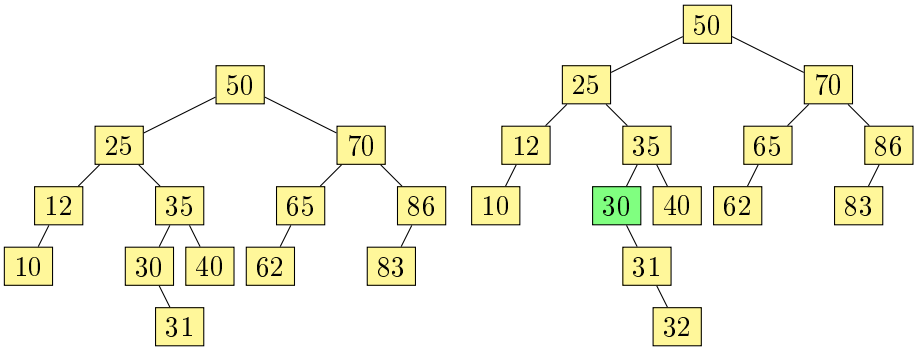


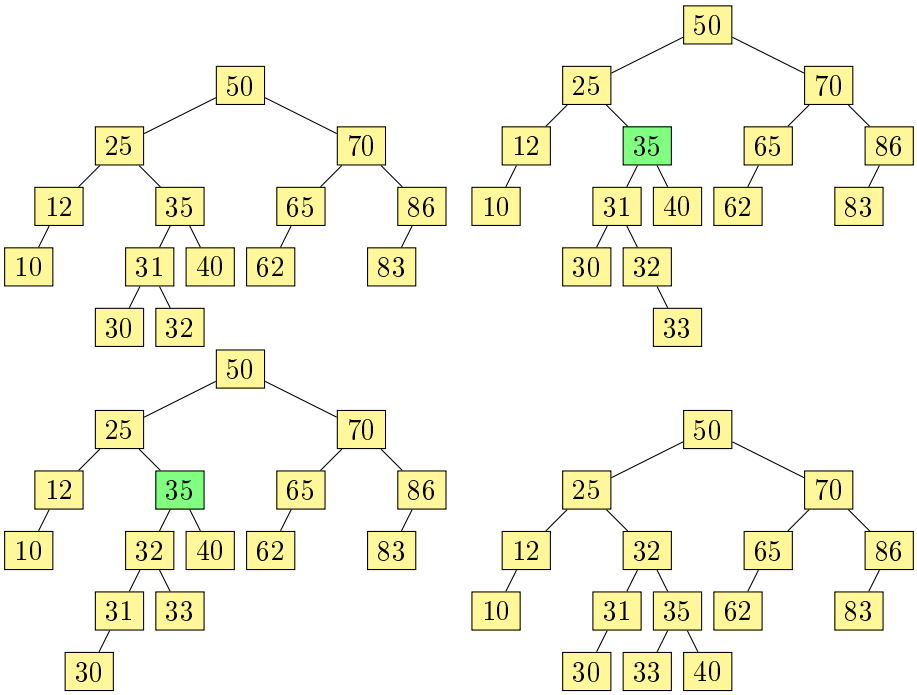
Solució 3.21: Si l'arbre a és buit la resposta és 0. Si l'arrel d'a és menor que x, llavors tots els elements del subarbre esquerre són menors que x, la pròpia arrel ho és, i a més alguns elements del subarbre dret també. Si l'arrel d'a és més gran o igual que x llavors només hem de continuar buscant 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 $\Theta(n)$.

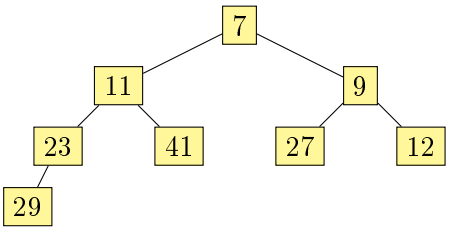
Solució 3.25: Les succesives figures mostren l'evolució de l'AVL. L'inserció de la clau 31 no provoca cap problema. La següent inserció, de la clau 32, provoca una violació de l'invariant dels AVLs al node 30 (marcat en verd) i la següent figura mostra el resultat un cop que fem la corresponent rotació simple. La inserció de la clau 33 requereix una rotació doble per a corregir el desequilibri que hi ha al node 35.





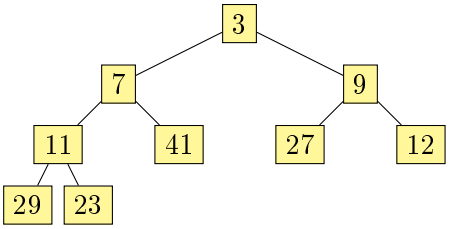
Solució 3.35: L'estat inicial és aquest:

7	11	9	23	41	27	12	29	



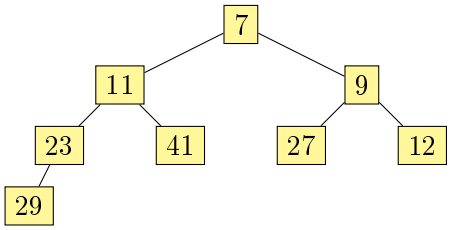
Per a inserir el nou element (el 3) es posa a l'última posició del vector i s'aplica un operació de "surar" que en aquest cas el fa pujar fins l'arrel donat que és mínim. El heap queda així:

3	7	9	11	41	27	12	29	23



Finalment, per eliminar el mínim (que és el 3), es subsituteix 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 3.40:

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* pude 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 $n-2$, $n-3$, ..., $n-k = n - \log_2(n+1)$. 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 $n-2, n-3, \ldots, n-k+1$.

   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 3.45:

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 3.53: 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, treien 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 3.56: 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 contiguns 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 4.7: 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 4.17: 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**
    $G^T[w]$.Append(v)
  end for
end for

Solució 4.19: 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 ← |V|
    for v ← 1, n do
        for w ← 1, n do
            for x ← 1, n do
                for y ← 1, n do
                    if {v, w}, {w, x}, {x, y}, {y, v} ∈ 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 ∈ V do
        for all vèrtex w adjacent amb el vèrtex v do
            if v ≠ w then
                for all vèrtex x adjacent amb el vèrtex w do
                    if v ≠ x and w ≠ x then
                        for all vèrtex y adjacent amb el vèrtex x do
                            if v ≠ y and w ≠ y and x ≠ y then
                                if {v, y} ∈ E then
                                    return true
                                end if
                            end if
                        end for
                    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 ≠ w do
        X := {x ∈ V | {v, x} ∈ E}
        Y := {y ∈ V | {w, y} ∈ E}
        if |X ∩ Y \ {v, w}| ≥ 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 4.20: 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 podriem 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 $n - 1$ pasos 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 ← 1
    for all j ← 2, n do
        if (i, j) ∈ E then
            i ← j
        end if
    end for
    for all j from 1 to n do
        if (i, j) ∈ E or ((j, i) ∉ E and i ≠ 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 4.22: Si el graf és connex, basta calcular-ne la talla m i comparar-la amb el seu ordre n.

```
function ACÍCLIC(V, E)
```

```
    n ← m ← 0
    for all vèrtex v ∈ V do
        n ← n + 1
    end for
    for all vèrtex v ∈ V do
        for all vèrtex w adjacent amb el vèrtex v do
            m ← 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 4.24: 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 ∈ G do
        vis[v] ← false
    end for
    L ← [ ]
    for all v ∈ G do
        if ¬vis[v] then
            OrdenacióInversa-Rec(G, v, vis, L)
        end if
    end for
end procedure
procedure OrdenacióInversa-Rec(G, v, vis, L)
    vis[v] ← true
        ▷ G[v] = llista de succesors del vèrtex v
    for all w ∈ G[v] do
        if ¬vis[w] then
            OrdenacióInversa-Rec(G, v, vis, L)
        end if
    end for
    L ← L.Append(v)
end procedure
```

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

Capítol 6

Solucions (professors)

Anàlisi d'algorismes

Algorismes de dividir i vèncer

Solució 2.1: 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

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

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 2.3: La solución de este problema es casi idéntica a la del problema 2. 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 2.4: 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 2.6: 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 2.8: 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 2.9: 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 2.10: 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 2.15: 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 qued así: $v = [4', 4, 6, 7]$. Las siguiente llamadas recursivas no modifican nada.

Solució 2.16: 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 2.23: Siguin N i M les longituts 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 ≤ i ∨ j' ≤ i' then
        Resolem el problema trivialment
    else
        ▷ El k-èssim global es troba a A[i..j] + B[i'..j']
        m ← (i + j)/2; m' ← (i' + j')/2
        if x > y then
            if k ≥ 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

Estructures de dades

Solució 3.5: 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 3.18:
```
// 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 3.20: Si el BST es vacío, la lista a devolver es también vacía. Supongamos que el BST no es vacío 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 3.39: El algoritmo que aplicamos hunde sucesivamente los elementos $A[n/2]$, $A[n/2-1]$,..., $A[1]$. La representación en vector del *min-heap* resultante es:

11	21	27	45	53	97	34	78

Solució 3.54: 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 componets 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 ténen un cost asimptòticament equivalent.

El segón algorisme és interessant si els elements van arrivant 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 $\Theta(n)$.

Grafs

Solució 4.13: El resultado del algoritmo se devolverá a través de dos vectors `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 $v \in G$ **do**
    $visitado[v] \leftarrow false$
    $ge[v] \leftarrow 0; gs[v] \leftarrow 0$
  end for
  for all $v \in G$ **do**
    if $\neg visitado[v]$ **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.