A

Enunciats d'examens

Parcial 3/11/2004

1. 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 $\mathrm{O}(\log n)$.

   d) El cost de l'algorisme de l'Omer en el cas pitjor és per força $\Theta(n \log n)$.

2. 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.

3. 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.

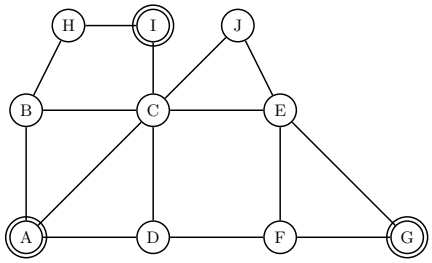
4. Considereu una matriu M de talla $n \times n$ en què cada columna està ordenada de manera estrictament creixent de dalt a baix, i cada fila està ordenada de manera estrictament creixent d'esquerra a dreta. Escriviu en C++ o Java una funció que, donat un element x, determini en temps $\Theta(n)$ quants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

5. 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 $1 \leqslant i < j \leqslant n$.

   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.

6. a) 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 $V' = \{A, G, I\}$.



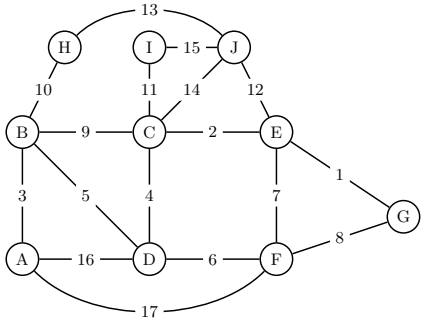
Suposeu que el graf es troba representat per llistes d'adjacència. El vostre algorisme ha de tenir cost $O(|V|(|V| + |E|))$. Formuleu el vostre algorisme amb un grau de detall prou gran perquè tots els seus elements quedin ben clars. Argumenteu la correctesa de la vostra solució.

b) 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 de l'apartat A tot escollint un conjunt V' adequat.

Final 11/1/2005

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



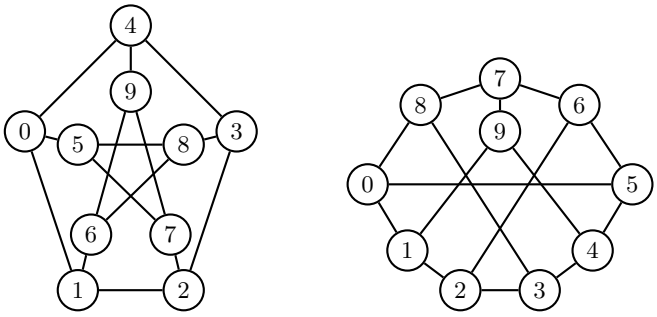
2. S'han de planificar n tasques en un processador que triga $\ell[i]$ unitats de temps a completar la i-èsima tasca. A més, cada tasca ha de començar, com a molt tard, a l'instant $t[i]$ (suposeu que tots els $t[i]$ són diferents). Una seqüència factible és una ordenació de les tasques tal que la seva execució en aquell ordre fa que mai no es realitzi més d'una tasca alhora i que cada tasca i comenci a l'instant $t[i]$ o abans.

   El problema de la planificació consisteix a, donats els $\ell[i]$ i els $t[i]$, trobar una seqüència factible, si aquesta existeix.

   a) Considereu l'algorisme voraç que ordena les tasques en funció del seu $t[i]$ (les tasques amb $t[i]$ més petits abans). Mostreu que aquest algorisme no és correcte, tot donant un exemple d'entrada per al qual falli.

   b) Proposeu un algorisme voraç que resolgui el problema. Demostreu la correctesa del vostre algorisme i analitzeu-ne el cost.

3. Dos grafs no dirigits $G_1 = (V_1, E_1)$ i $G_2 = (V_2, E_2)$ es diuen *isomorfs* si existeix una bijecció $f : V_1 \to V_2$ tal que $\{u, v\} \in E_1$ si i només si $\{f(u), f(v)\} \in E_2$. Per exemple, els dos grafs següents son isomorfs, com ho certifica la bijecció $f(0) = 8$, $f(1) = 3$, $f(2) = 2$, $f(3) = 6$, $f(4) = 7$, $f(5) = 0$, $f(6) = 4$, $f(7) = 1$, $f(8) = 5$ i $f(9) = 9$.



Donats dos subconjunts $S_1 \subseteq V_1$, $S_2 \subseteq V_2$, i una bijecció $f : S_1 \to S_2$, diem que f és un *isomorfisme parcial* per a S_1 i S_2 quan per a tot parell $u, v \in S_1$ es compleix $\{u, v\} \in E_1$ si i només si $\{f(u), f(v)\} \in E_2$. Observeu que si f és un isomorfisme parcial per a $S_1 = V_1$ i $S_2 = V_2$, llavors G_1 i G_2 són isomorfs.

L'equip de programació de la UPC (entrenat per dos professors d'ADA) us ha dissenyat un algorisme de tornada enrera (*backtracking*) per trobar si dos grafs donats són isomorfs o no (i en el cas que ho siguin, retornar una bijecció que ho certifiqui). Malauradament, en enviar-vos el seu programa, part del codi s'ha perdut. Així doncs, completeu la classe següent en C++, tot indicant quina instrucció o condició (només una per espai) posaríeu a cada espai requadrat.

Observeu que els grafs (no dirigits) es troben implementats amb matrius d'adjacència. Per tant, `G[u][v]` és un booleà indicant si `u` i `v` es troben connectats en `G`, i val el mateix que `G[v][u]`.

Observeu també que se suposa que els grafs ja tenen el mateix nombre de vèrtexs i d'arestes (altrament seria immediat concloure que no són isomorfs).

```
typedef matrix<bool> graf;

class Isomorfisme {

    graf G1, G2;            // els grafs d'entrada
    int n;                  // el nombre de vèrtexs
    vector<int> f;          // la bijecció parcial
    vector<bool> usat;      // usat[w] = (algun v ⩽ u compleix f[v] = w)
    bool iso;               // indica si G1 i G2 són isomorfs

    bool backtracking (int u) {
        if ( [                                ] ) return true;
        for (int w=0; w<n; ++w) {
            if (not usat[w]) {
                f[u] = w;
                usat[w] = true;
                if (isomorfisme_parcial(u)) {
```

```
                    if (backtracking(u+1)) {
                        return true;
                    }
                }
                [                                ]
            }
        }
        return false;
    }

    bool isomorfisme_parcial (int u) {
        for (int v=0; v<u; ++v) {
            if ( [                              ] ) {
                return false;
            }
        }
        return true;
    }

public:

    // Precondició: G1 i G2 tenen el mateix nombre de vèrtexs i d'arestes.
    Isomorfisme (graf G1, graf G2) {
        this->G1 = G1;  this->G2 = G2;  n = G1.size();
        f = vector<int>(n);
        usat = vector<bool>(n,false);
        [                                ]
    }

    bool isomorfs ()        { return iso; }

    // Precondició: isomorfs().
    vector<int> bijeccio () { return f;   }
}
```

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

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

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

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

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

JONNY: Ostres, així ja l'hem fotuda!

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

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

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

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

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

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

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

a) Expliqueu quina solució ha trobat l'Steffy per saber si alguns d'ells poden fer ràfting (és a dir, doneu una reducció del problema d'en Jonny al problema del geni).

b) Acabeu de demostrar que el problema del geni és **NP**-complet.

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

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

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

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

Parcial 27/4/2005

1. Considereu el següent cèlebre algorisme camuflat misteriosament:

```
template <typename elem>
void misteri (vector<elem>& v, int i) {
    if (i>1) {
        int p = 0;
        for (int j=1; j<i; ++j) {
            if (v[j] > v[p]) {
                p = j;
        }   }
        swap(v[p],v[i-1]);
        misteri(v,i-1);
}   }
```

a) Analitzeu el cost en temps de la crida `misteri(v,v.size())`, suposant que totes les operacions bàsiques sobre `elems` prenen temps constant.

b) Digueu en una frase què fa i com es diu aquest algorisme.

2. a) Descriviu (en alt nivell, és a dir, sense escriure codi ni pseudocodi però amb precisió) un algorisme de cost $\Theta(\log n)$ que, donada una taula ordenada T amb n elements, un element x i un enter k, indiqui si x apareix k o més vegades a T.

   b) A partir de l'algorisme anterior, descriviu-ne un altre amb el mateix cost que, donada una taula ordenada T amb n elements, indiqui si aquesta conté algun element repetit més de $n/3$ cops.

3. Es té com a entrada dues taules $T_1[1 \ldots n]$ i $T_2[1 \ldots n]$ que contenen a cada posició un registre amb informacions de persones de la forma $\langle \mathit{cognom}_1, \mathit{cognom}_2 \rangle$. Les dues taules contenen la mateixa informació, però T_1 està ordenada per cognom_1 i T_2 està ordenada per cognom_2. Cap cognom conté més de 24 caràcters.

   Es vol construir una nova taula T amb les mateixes dades, però ordenades segons l'ordre lexicogràfic usual. Per exemple, si l'entrada fós les dues taules de l'esquerra, la sortida ha de ser la taula de la dreta:

T_1

ABAD	FERRER
LLAC	SERRAT
LLAC	ROIG
PIN	FERRER

T_2

PIN	FERRER
ABAD	FERRER
LLAC	ROIG
LLAC	SERRAT

$\Rightarrow$

T

ABAD	FERRER
LLAC	ROIG
LLAC	SERRAT
PIN	FERRER

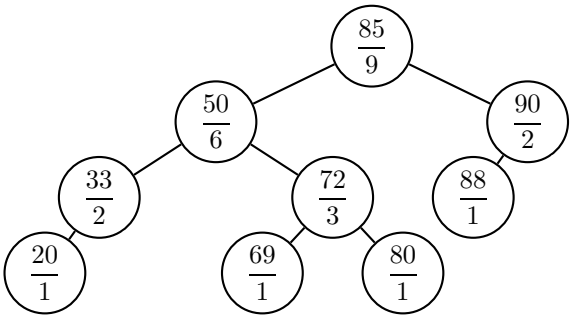
   Descriviu (en alt nivell, és a dir, sense escriure codi ni pseudocodi però amb precisió) un algorisme que resolgui aquest problema amb $\Theta(n)$ passos *en el cas mitjà*. Pista: penseu en una taula de dispersió.

4. Considereu un tipus abstracte de dades genèric per emmagatzemar un conjunt d'elements (sense repetits). Les seves operacions són la creació d'un conjunt buit, la inserció d'un element, la selecció de l'element i-èsim (seguint l'ordre dels elements dins el conjunt), i la consulta del nombre d'elements.

   Per exemple, si al conjunt $\{20, 33, 50, 69, 72, 80, 88, 90\}$ se li inserís l'element 85, llavors s'obtindria el conjunt $\{20, 33, 50, 69, 72, 80, 85, 88, 90\}$, que té 9 elements. Si es seleccionés el vuitè element d'aquest nou conjunt, s'obtindria el 88.

   Per tal d'implementar aquest tipus abstracte de dades, usarem "arbres binaris de cerca augmentats". Un arbre de cerca augmentat és un arbre de cerca en el qual cada node manté la talla del seu subarbre.

   Per exemple, l'arbre binari de cerca augmentat de la figura següent representa el conjunt $\{20, 33, 50, 69, 72, 80, 85, 88, 90\}$.



Observeu que el node corresponent a l'element 50 guarda que la talla del seu subarbre és 6 i que l'arrel guarda el nombre total d'elements al conjunt.

Implementeu les operacions públiques de la classe següent en C++. El cost de l'operació de consulta del nombre d'elements ha de ser constant. El cost de les operacions d'inserció i de selecció ha de ser proporcional a l'alçada de l'arbre. A més, si l'arbre contingués n elements inserits en un ordre aleatori, el cost mitjà d'aquestes dues operacions hauria de ser $\Theta(\log n)$.

Podeu definir funcions auxiliars (a la part privada), però no teniu dret a modificar els camps donats ni afegir-ne de nous. El constructor i el destructor ja es donen implementats, i tampoc els podeu alterar.

```
template <typename elem>
class Conjunt {

private:

    struct node {
        elem x;                 // Element
        node* fe;               // Punter al fill esquerre
        node* fd;               // Punter al fill dret
        int t;                  // Talla del subarbre

        node (elem xx) {        // Constructor de node
            x = xx;  fe = fd = null;  t = 1;
        }

        ~node () {              // Destructor de node
            delete fe;  delete fd;
        }
    };

    node* arrel;                // Arrel de l'ABC augmentat

public:

    // Crea un conjunt buit.
    Conjunt () {
        arrel = null;           // L'ABC es fa buit
    }
```

```
    // Destructor.
    ~Conjunt () {
        delete arrel;            // Allibera recursivament tots els nodes
    }

    // Retorna el nombre d'elements al conjunt.
    int elements () {
         ...
    }

    // Insereix l'element x al conjunt si no hi era.
    void inserir (elem x) {
        ...
    }

    // Retorna l'element i-èsim del conjunt. Precondició: 1 ⩽ i ⩽ elements().
    elem selec (int i) {
        ...
    }

};
```

5. a) Descriviu breument un algorisme que, donat un graf dirigit sense cicles (DAG), retorni la llargada del seu camí més llarg. Pista: Modifiqueu l'algorisme d'ordenació topològica.

   b) Implementeu completament el vostre algorisme en la funció següent:

```
int llargada_camí_més_llarg (graf& G);
```

   Per a fer-ho, utilitzeu les definicions següent per a grafs dirigits (les mateixes que hi ha a les transparències de classe):

```
typedef vector< list<int> > graf;
typedef list<int>::iterator arc;

#define forall_adj(uv,L) for (arc uv=(L).begin(); uv!=(L).end(); ++uv)
#define forall_ver(u,G) for (int u=0; u<(G).size(); ++u)
```

   c) Analitzeu el cost del vostre algorisme.

Final 17/6/2005

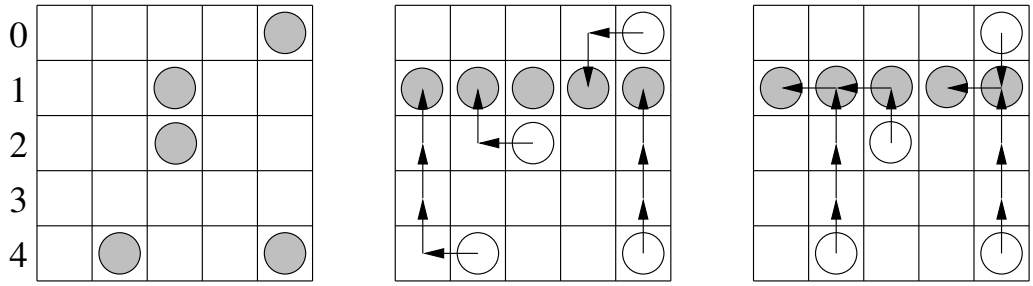
1. Sigui G un graf no dirigit connex amb n vèrtexs, m arestes i un únic cicle que té llargada ℓ. Digueu (breument però de forma justificada) quants arbres d'expansió diferents té aquest graf.

2. Un dels estudiants d'ADA d'aquest curs és germà d'en Xavier Martínez Palau, un teleco-mates sonat que l'abril passat va formar part de l'equip que va representar la

UPC en la final mundial del Concurs de Programació a Shanghai. El problema que en Xavier va haver de resoldre és, molt simplificat, el següent:

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

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



Escriviu la funció `int moviments (int n, int i, vector<int>& X, vector<int>& Y)`, la qual reb el nombre de fitxes, la fila, i les coordenades (`X[i]`,`Y[i]`) de la `i`-èsima peça en els vectors `X` i `Y`. Si us cal, podeu usar qualsevol procediment auxiliar bàsic que s'hagi explicat a l'assignatura.

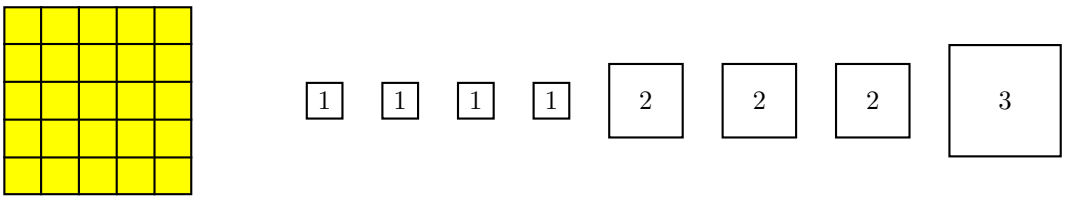
Pista: El nombre de moviments verticals (amunt/avall) necessaris es pot calcular fàcilment.

3. Considereu els dos problemes següents (sobre grafs no dirigits):

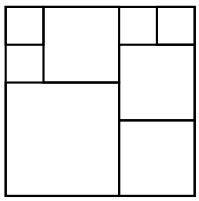
   - CLIQUE: Donats un graf $G = (V, E)$ i un natural s, determinar si G conté una clique amb s vèrtexs.
     Recordeu que una clique és un subconjunt de vèrtexs $S \subseteq V$ tal que per a qualsevol parell de vèrtexs diferents u i v en S hi ha una aresta entre u i v en E.
   - SUBGRAF-ESBORRAT: Donats dos grafs G_1 i G_2, determinar si es pot obtenir un graf isomorf a G_1 tot esborrant arestes de G_2.

   Sabent que CLIQUE és **NP**-complet, demostreu que SUBGRAF-ESBORRAT també ho és.

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



aquesta seria una possible solució:



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

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

```
class Puzzle {

    int n;              // Nombre de files i de columnes del tauler
    int k;              // Nombre de fitxes
    vector<int> A;      // Taula amb la mida de cada fitxa
    bool trob;          // Indica si s'ha trobat una solució
    Tauler T;           // Tauler

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

public:

    // Soluciona el puzzle per a un tauler n × n amb fitxes de mida A.
    Puzzle (int n, vector<int> A) {
        this->n = n;
        this->A = A;
        k = A.size();
        trob = false;
        T = Tauler(n);
        recursiu( ... );
    }

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

```
class Tauler {

public:
```

.....

Construeix un tauler buit de mida `n` × `n`.

.....

```
Tauler (int n);
```

.....

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

.....

```
bool es_pot (int x1, int y1, int x2, int y2);
```

.....

Col·loca una fitxa des de la posició (`x1`, `y1`) a la posició (`x2`, `y2`).
Precondició: `es_pot(x1,y1,x2,y2)`.

.....

```
bool col·loca (int x1, int y1, int x2, int y2);
```

.....

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

.....

```
void retira (int x, int y);
```

.....

Indica si la posició (`x`, `y`) està ocupada per alguna fitxa.

.....

```
bool ocupada (int x, int y);
```

```
};
```

5. Falta una setmana per Sant Joan. La colla d'en Jonny, en Roy i l'Steffy té previst passar la revetlla menjant coca, vevent cava, tirant coets i mirant el vídeo de *Battle Royale*. Els nostres amics són els responsables de comprar els coets. Entre ells té lloc la conversa següent:

   Jonny He aconseguit recaptar P euros per comprar coets. La gent m'ha demanat que aquest any facin molt de soroll però que no en comprem cap de repetit.

   Roy Doncs en aquest catàleg hi ha N tipus de coets disponibles. Per a cada tipus

hi figura el seu preu $p[i]$, el soroll que fa quan explota $s[i]$ i els grams de pólvora que conté $g[i]$.

JONNY Perfecte: mirem quins coets cal comprar per fer el màxim de soroll sense passar-nos del pressupost ni triar-ne dos del mateix tipus, agafem el cotxe de l'Steffy i els comprem.

STEFFY Ep, nois... Cal tenir a més en compte que la legislació vigent prohibeix transportar més de G grams de pólvora en un vehicle privat!

Per tal d'ajudar als nois a resoldre el seu problema, definim la funció $sor(i, p, g)$ com el soroll màxim assolible utilitzant alguns dels i primers coets del catàleg, amb un preu total de p euros o menys i amb g grams de pólvora o menys. L'objectiu és doncs calcular $sor(N, P, G)$.

a) Escriviu una recurrència (o un algorisme recursiu) per a $sor(i, p, g)$.

b) Expresseu en funció de N, P i G quin cost tindria l'algorisme de programació dinàmica resultant (no heu d'escriure l'algorisme, només donar el seu cost).

Comentaris: Suposeu que el soroll que fan diversos coets és la suma del soroll de cadascun d'ells. Suposeu també que tots els valors són nombres naturals.

B

Solucions d'examens

Observacions:

- Les solucions proposades no són úniques.
- Els textos entre claudàtors proporcionen aclariments o informació addicional que no es demanava a l'examen.

Parcial 3/11/2004

1. L'única possibilitat correcta és la b). Es pot veure que totes les altres afirmacions són falses perquè l'algorisme de l'Omer podria ser *molt* lent. En canvi, sigui quin sigui l'algorisme que hagi fet, aquest necessitarà com a mínim mirar un cop cadascun dels n possibles elements.

   [Per cert, per trobar la mediana d'una taula amb n elements no cal ordenar-la, hi ha algorismes lineals en el cas pitjor.]

2. La llibreria estàndard de C++ defineix que el cost d'inserir (`push`) i esborrar (`pop`) d'una cua de prioritats amb m elements és $\mathrm{O}(\log m)$ i que el cost de consultar el més gran (`top`) és $\mathrm{O}(1)$. Com que es fan n insercions i n esborrats en una cua de prioritats que mai té més de n elements, el temps total és $\mathrm{O}(n \log n)$.

   D'altra banda, la `priority_queue` utilitza comparacions per organitzar els seus elements. Com que qualsevol algorisme basat en comparacions necessita $\Omega(n \log n)$ comparacions per ordenar n elements, podem concloure que el temps de l'algorisme proposat és $\Theta(n \log n)$.

   Com que la cua de prioritats ha d'emmagatzemar fins a n elements, l'espai auxiliar requerit és $\Theta(n)$.

   [Com que `priority_queue` és un TAD, no es poden fer assumpcions sobre la seva implementació (en un *heap*, per exemple).]

3. La resposta c) és correcta: Cal ordenar primer pel segon cognom perquè el primer cognom és el primer criteri de l'ordenació i la segona ordenació, pel fet d'haver estat posterior, preval sobre la primera. A més, cal ordenar pel primer cognom amb un algorisme estable (*mergesort* ho és) perquè cal preservar l'ordre relatiu dels elements en cas d'empat.

   El cost en espai és $\Theta(n)$ a causa de l'espai extra que necessita el *mergesort*. [De fet, el *quicksort* pot usar també un espai $\Theta(n)$ per a la pila, segons com estigui implementat.]

   El cost en temps en el cas pitjor és $\Theta(n^2) + \Theta(n \log n) = \Theta(n^2)$, perquè en el cas pitjor *quicksort* triga $\Theta(n^2)$ i *mergesort* triga $\Theta(n \log n)$.

   El cost en temps en el cas mitjà és $\Theta(n \log n)$, perquè en el cas mitjà tant *mergesort* com *quicksort* triguen $\Theta(n \log n)$.

4. L'algorisme següent en C++ fa com a molt $2n$ iteracions:

```
template <class elem>
int menors (matrix<elem>& M, elem x) {
    int n = M.rows(), c = 0, i = 0, j = n-1;
    while (j>=0 and i<n) {
        if (M[i][j] < x) {
            c += j+1;  ++i;
        } else {
            --j;
    }   }
    return c;
}
```

5.a) Si (i, j) és una inversió vol dir que $T[i] > T[j]$. Suposem que existís un k tal que $i < k < j$. Com que la taula té una única inversió, (i, k) no pot ser una inversió. Per tant $T[i] \leqslant T[k]$. Igualment, (k, j) tampoc pot ser una inversió. Per tant $T[k] \leqslant T[j]$. En conseqüència, $T[i] \leqslant T[j]$, fet que contradiu que (i, j) sigui una inversió. D'això se'n dedueix que no existeix tal k i, per tant, $i + 1 = j$.

   Resposta alternativa: Si $\forall i : T[i] \leqslant T[i+1]$, aleshores la taula T no té cap inversió. Per tant, contrarecíprocament, si la taula T té alguna inversió, aleshores $\exists i : T[i] > T[i+1]$. Per tant, si només hi ha una inversió, aquesta ha de ser dos elements consecutius.

5.b) Una possible manera de fer-ho és procedint de la mateixa forma que la cerca dicotòmica, però prenent la precaució de comparar l'element buscat amb els dos elements contigus a l'element de la taula al qual es cau per tal d'evitar "ser enganyats" per la possible inversió. És fàcil veure que quan l'element cercat no és cap dels tres explorats, la cerca dicotòmica funciona bé, encara que hi hagi inversió.

   [Una possible implementació d'aquesta idea en C++ seria:

```
template <typename elem>
bool cerca (vector<elem>& T, elem x, int e, int d) {
    if (e>d) return false;
    int m = (e+d)/2;
    if (T[m]==x) return true;
    if (T[m]<x) {
```

```
            if (m>=1 and T[m-1]==x) return true;
            else return cerca(T,x,m+1,d);
        } else {
            if (m+1<T.size() and T[m+1]==x) return true;
            else return cerca(T,x,e,m-1);
    }   }
```

]

6.a) Per a cada vèrtex de V', es fa una crida a un recorregut en amplada [no en profunditat] començant des d'ell. De cada recorregut en amplada cal mirar quina és la distància mínima de l'arrel a un vèrtex en V'. De totes aquestes distàncies mínimes, cal retornar la més petita. De fet, es pot abortar un recorregut si aquest ja ha vist més nivells que la distància més petita trobada fins al moment.

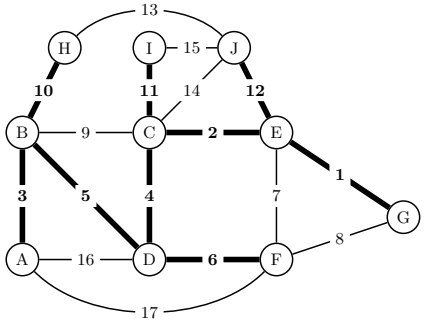
Com que un recorregut en amplada té cost $O(|V| + |E|)$ i en podem fer $|V'| \leqslant |V|$ com a molt, el cost total és el que es demana.

6.b) Tot cicle que passa per u és de la forma $\langle u, v_1, \ldots, v_n, u\rangle$ amb v_1 i v_n veïns de u. Per tant, els veïns de u que estiguin a distancia més curta sense passar per u ens donaran també el cicle més curt passant per u.

Així, esborrem u de G i utilitzem l'algorisme de l'apartat anterior tot escollint els veïns originals de u com a V'. Si a la distància més petita entre els vèrtexs de V' li sumem 2, aquesta és la llargada del cicle més curt que passa per u. Això funciona fins i tot si el cicle no existís, ja que l'algorisme anterior retornaria $+\infty$ al qual se li sumaria 2.

Final 11/1/2005

1. [Com que el graf no té pesos repetits a les arestes, hi ha un únic arbre d'expansió mínim.]



2.a) Un contraexemple possible amb dues tasques es dóna quan $t[1] = 2$ i $\ell[1] = 5$ i quan $t[2] = 3$ i $\ell[2] = 1$. En aquest cas, l'algorisme proposat col·loca la tasca 1 abans de la 2 i, per tant, produeix una seqüència infactible. En canvi, col·locant la tasca 2 abans de la 1 s'obté una seqüència factible.

2.b) L'algorisme voraç ordena les tasques segons el seu temps d'acabament $t[i] + \ell[i]$.

Si la seqüència de tasques no és factible, l'algorisme retorna una seqüència infactible, tal com cal. Altrament, sigui t la solució trobada per l'algorisme proposat i sigui s qualsevol altra solució factible. Llavors s i t tenen, almenys, una inversió. És fàcil veure que

quan s'intercanvien dues tasques consecutives que contradiuen l'ordre proposat inversió, s'obté una altra solució factible. Així, desfent l'una rera l'altra aquestes inversions consecutives de s, s'obté t. Per tant, t és una solució factible, tal com cal.

El cost de l'algorisme és dominat pel cost de l'ordenació i, per tant, és $\Theta(n \log n)$ en el cas pitjor.

2. El darrer espai ha d'iniciar el procés de tornada enrera amb una solució parcial buida, deixant el resultat al camp `iso`. Com que el primer vèrtex és el zero, cal omplir aquest espai amb `iso = backtracking(0);`.

   El primer espai es correspon a la condició de final amb èxit de la tornada enrera. Aquesta es produeix quan la solució parcial és una solució total, per tant cal omplir aquest espai amb `u == n`.

   En el segon espai, es prepara l'iteració següent, la qual generarà el proper fill a provar. Aquí cal restablirel valor de cert a `usat[w]`, perquè `w` ja no és usat. Cal doncs omplir el espai amb `usat[w] = false;`.

   El tercer espai és a la funció que comprova que l'addició del vèrtex `u` a `f` formi un isomorfisme parcial, sabent que ja el formaven tots els demés vèrtexs usats. Per tant, aquí cal comprovar que per a tot `v`, si hi ha una aresta de `v` a `u` en `G1`, aquesta també hi sigui en `G2`, i del revés. Cal doncs omplir el espai amb `G1[u][v]==G2[f[u]][f[v]]`.

4.a) Aquí va la reducció del problema d'en Jonny al problema del geni: Agafem $m := n + 1$ i per a cada $1 \leqslant i \leqslant n$, agafem $b_i := p_i$. A més, agafem $b_{n+1} := -T$.

- $\Rightarrow$ Si alguns dels p_i sumen exactament T, llavors aquells b_i amb $-T$ sumen zero.
- $\Leftarrow$ Si algun conjunt no buit de b_i suma zero, és perquè entre ells hi ha el $-T$ (altrament no sumarien zero perquè tots els b_i amb $i \neq n + 1$ són estrictament positius). Llavors els p_i corresponents sumen T.

Aquesta reducció es pot portar a terme en temps lineal (i, per tant, en temps polinòmic).

[Com que la Steffy ens ha dit que el problema d'en Jonny és **NP**-complet, ara sabem que el problema del geni és **NP**-difícil. Això justifica que el venedor d'Istambul digués que la llàntia resol eficientment qualsevol problema **NP** (el venedor, però, es va callar que calia trobar les reduccions adients per utilitzar-la!).]

3.b) La Steffy ens ha fet saber que el problema d'en Jonny és **NP**-complet, i a l'apartat anterior hem reduït el problema d'en Jonny al del geni. Per tant, per acabar de demostrar que el problema del geni és **NP**-complet, només cal demostrar que aquest pertany a **NP**.

Un testimoni possible per al problema del geni és una taula de m bits: El bit i-èsim indica si cal agafar o no el número b_i. El testimoni té llargada polinòmica respecte de l'entrada (aquesta ha d'ocupar almenys m bits) i es pot comprovar en temps lineal si els números seleccionats sumen o no zero.

3. Trobar el nombre mínim de talls que calen per aconseguir un retall en forma de palíndroms de s correspon a calcular $t[1, n]$. Per fer-ho, utilitzem la recurrència següent:

$$t[i,j] = \begin{cases} 0 & \text{si } s[i,j] \text{ és un palíndrom,} \\ \min\{1 + t[i,k] + t[k+1,j] \ : \ i \leqslant k < j\} & \text{altrament.} \end{cases}$$

En efecte, si $s[i,j]$ no és un palíndrom, cal trobar un punt k amb $i \leqslant k < j$ per fer-li un tall, tallar de forma òptima la part esquerra $s[i,k]$, i tallar de forma òptima la part dreta $s[k+1,j]$. Logicament, cal triar el valor de k que minimitzi aquesta suma de talls.

La recurrència anterior es pot implementar directament de dalt cap a baix de forma recursiva, tot utilitzant memorització per tal de no repetir càlculs.

Com que, gràcies a la memorització, cada $t[i,j]$ es calcula com a molt un cop, el cost d'aquesta algorisme és:

$$T(n) = \sum_{i=1..n} \sum_{j=i..n} \left(\mathrm{O}(j-i+1) + \sum_{k=i..j} \Theta(1) \right) = \Theta(n^3).$$

(Per a cada $t[i,j]$ amb $1 \leqslant i \leqslant j \leqslant n$ calen $\mathrm{O}(j-i+1)$ passos per calcular si $s[i,j]$ és un palíndrom i $\sum_{k=i..j} \Theta(1)$ passos per calcular el mínim de $j-i$ valors.)

També es pot implementar la programació dinàmica de baix cap a dalt omplint la taula iterativament. Com que l'estructura d'aquesta recurrència és idèntica a la del problema dels productes encadenats de matrius o a la del problema del tall de les barres d'acer, un algorisme semblant farà el fet. El seu cost també serà cúbic.

[Amb un algorisme de programació dinàmica més astut es pot aconseguir una solució quadràtica.]

Parcial 27/4/2005

1. a) De l'algorisme extraiem la recurrència

$$T(i) = \begin{cases} \Theta(n) + T(i-1), & \text{si } i > 1 \\ \Theta(1), & \text{altrament} \end{cases}$$

la solució de la qual és $T(i) = \Theta(i^2)$. Per tant, si la talla del vector `v` fós n, la crida plantejada tindria cost $\Theta(n^2)$.

b) Aquest algorisme ordena creixentment els `i` primers elements del vector `v`; es tracta d'una implementació recursiva [final] de l'algorisme d'ordenació per selecció. [A cada crida es busca l'element més gran i es col·loca al final del vector.] [Error freqüent: No és l'algorisme de la bombolla: bombolla fa un nombre quadràtic d'intercanvis, aquest en fa un nombre lineal; bombolla només intercanvia elements contigus, aquest no.]

2. a) Primer es fa una cerca dicotòmica per buscar la primera ocurrència de x en T. Sigui e la posició on s'ha trobat (si x no és a T, retorna fals). Després es fa una segona cerca dicotòmica per buscar la darrera ocurrència de x en T. Sigui d la posició on s'ha trobat. Ja només cal mirar si entre e i d hi ha k o més posicions.

Com que es fan dues cerques dicotòmiques, el cost de l'algorisme és logarítmic, tal com cal.

[Error frequent: Els algorismes que fan una cerca dicotòmica per trobar x i després es desplacen a dreta i/o esquerra incrementalment comptant el nombre d'elements iguals tenen cost $\Theta(\log n + k)$ que és $\Theta(n)$ quan $k = n$.]

b) Si en una taula ordenada amb n elements un element apareix més de $n/3$ cops, llavors aquest s'ha de trobar, com a mínim, a la posició $n/3$ o a la posició $2n/3$. Siguin x_1 i x_2 els elements en aquestes posicions, respectivament. Utilitzem l'algorisme anterior per esbrinar si x_1 apareix almenys $n/3$ cops i per esbrinar si x_2 apareix almenys $n/3$ cops. La resposta és afirmativa si i només si alguna de les crides a l'algorisme anterior retorna cert.

Com que es realitzen dues crides a un algorisme de cost logarítmic, el cost total és també logarítmic, tal com cal.

3. Utilitzem una taula de dispersió encadenada. Les claus són el primer cognom. Els nodes de les llistes contenen el primer i el segon cognom. Quan s'afageix un element, s'afageix pel final a la seva llista.

Es recorren de dalt a baix tots els elements de T_2. Cadascun es col·loca a la taula de dispersió.

Després, es recorre de dalt a baix la taula T_1.

Quan apareix un primer cognom diferent de l'anterior, es cerca a la taula de dispersió, i es recorre la seva llista. Per cada node de la llista pel qual el primer cognom coincideixi amb l'actual de la taula T_1, s'escriu a T el primer i el segon cognom d'aquell node.

Quan apareix un cognom igual a l'anterior, no es fa res.

Com que les llistes de la taula de dispersió mantenen l'ordre dels segons cognoms i el recorregut de T_1 manté els del primer cognom, T queda ordenada tal com cal.

Pel cost, si fem la taula de dispersió lineal amb n (de fet, n'hi hauria prou amb lineal amb el nombre de diferents primers cognoms), cada inserció i cada cerca tindràn cost constant en mitjana. Com que es fan n insercions i, com a molt, n consultes, s'obté un cost lineal en el cas mitjà.

[Hi ha altres respostes correctes força diferents.]

4. Possible solució:

```
public:

    int elements ()         { return talla(arrel);   }
    void inserir (elem x)   { inserir(x,arrel);      }
    elem selec (int i)      { return selec(i,arrel); }

private:

    int talla (node* p) {
        return p ? p->t : 0;
    }

    elem selec (int i, node* p) {
        int m = 1+talla(p->fe);
        if (i==m) return p->x;
        else if (i<m) return selec(i,p->fe);
        else return selec(i-m-1,p->fd);
    }
```

```
bool inserir (elem x, node*& p) {   // El booleà indica si s'ha inserit
    if (!p) {
        p = new node(x);
        return true;
    } else if (p->x==x) {
        return false;
    } else if (p->x > x) {
        bool b = inserir(x,p->fe);
        if (b) ++p->t;
        return b;
    } else {
        bool b = inserir(x,p->fd);
        if (b) ++p->t;
        return b;
}   }
```

[Errors freqüents: No tenir en compte que poden haver-hi punters nuls; suposar que l'element a inserir no hi és; iIncrementar contadors de l'arbre sense assegurar-se que l'element a inserir no hi sigui; aplicar operacions de `Conjunt` a punters a nodes (tipus `arrel->fe.inserir(x);`) fer un recorregut en inordre per seleccionar l'i-èsim (cost lineal respecte el nombre d'elements a l'arbre).]

5. Semblant a cercar una ordenació topològica, però per a cada vèrtex es manté la longitud del camí més llarg trobat fins al moment que hi porta. El cost per a un DAG amb n vèrtexs i m arestes és $\Theta(n+m)$.

```
int llargada_camí_més_llarg (graf& G) {
    int n = G.size();
    vector<int> ge(n,0);      // grau entrada de cada vèrtex
    vector<int> ll(n,0);      // llargada camí més llarg fins a cada vèrtex
    int m = 0;                // màxim de les llargades màximes
    stack<int> P;
    arc uv;

    forall_ver(u,G) forall_adj(uv,G[u]) ++ge[*uv];
    forall_ver(u,G) if (ge[u]==0) P.push(u);

    while (not P.empty()) {
        int u = P.top();  P.pop();
        forall_adj(uv,G[u]) {
            int v = *uv;
            if (--ge[v]==0) P.push(v);
            ll[v] = max(ll[v],ll[u]+1);
            m = max(m,ll[v]);
    }   }

    return m;
}
```

[Errors freqüents: No seguir la pista; fer un algorisme de cost exponencial; confondre DAG amb arbre, fer un recorregut en amplada, trobar només un camí.]

Final 7/6/2005

1. Si G és connex, té n vèrtexs i conté un únic cicle, llavors $m = n$. Per obtenir un arbre d'expansió de G cal, doncs, esborrar-li una aresta. Si s'esborra una aresta que no és al cicle, s'obté un graf no connex (i per tant no s'obté un arbre); si s'esborra una aresta del cicle, s'obté un graf connex amb $m - 1$ arestes (i per tant un arbre). Com que es pot esborrar quasevol de les ℓ arestes del cicle, G té exactament ℓ arbres d'expansió diferents.

2. Cal posar totes les fitxes a la fila i-èsima, una a cada columna. Els dos objectius (1: fila i-èsima, 2: columnes diferents) es poden aconseguir de manera independent:

   (a) Cal pujar o baixar cada fitxa $|i - Y[j]|$ passos.
   (b) S'agafa la fitxa més a l'esquerra encara no usada i es posa a la columna més a l'esquerra encara no ocupada (això pot significar moure la fitxa cap a l'esquerra, o cap a la dreta, o no moure-la); es torna a començar. Ordenant prèviament les fitxes per columna s'aconsegueix fer en temps $\Theta(n \log n)$. Si, en canvi, es busca el mínim a cada pas el cost esdevé $\Theta(n^2)$.

```
int moviments(vector<int> &X, vector<int> &Y, int n, int i) {
    int res = 0;
    for (int j = 0; j < n; ++j) res += abs(X[j] - i);
    sort(Y.begin(), Y.end());
    for (int j = 0; j < n; ++j) res += abs(Y[j] - j);
    return res;
}
```

   [Existeixen solucions més sofisticades i eficients (cost lineal).]

3. Veiem primer que SUBGRAF-ESBORRAT pertany a **NP**: Els testimonis són una bijecció entre els vèrtexs de G_1 i G_2 [també es pot donar una llista d'arestes esborrades, però no és necessària]. Aquesta taula té talla polinòmica respecte la talla de l'entrada i és evident que es pot comprovar un testimoni en temps polinòmic.

   A continuació reduim CLIQUE a SUBGRAF-ESBORRAT: La reducció consisteix en fer $G_2 := G$ i $G_1 :=$ "un graf complet amb s vèrtexs i $n - s$ vèrtexs aïllats" on n és el nombre de vèrtexs de G. És clar que aquesta reducció és polinòmica.

   $\Rightarrow$ Si G conté una clique de talla s, en esborrar totes les arestes de $G_2 = G$ que no pertanyin a aquesta clique quedarà un graf isomorf a G_1.
   $\not\Rightarrow$ Si G no conté cap clique de talla s, no hi ha cap manera d'esborrar arestes de $G_2 = G$ i trobar un graf isomorf a G_1, que sí conté una clique de talla s.

   Queda doncs demostrat que SUBGRAF-ESBORRAT és **NP**-complet.

   [Errors freqüents: fer la reducció del revés, deixar-se els vèrtexs aïllats, donar per fet que determinar si dos grafs són isomorfs és en **P**, repetir les definicions particularitzant-les als problemes.]

4. Aquest solució consisteix en anar provant de col·locar totes les fitxes a totes les posicions possibles on es pugui:

```
void recursiu (int i) {
    if (i==k) {
        trob = true;
    } else {
        int m = A[i];
        for (int x=0; x<n-m; ++x) {
            for (int y=0; y<n-m; ++y) {
                if (T.es_pot(x,y,x+m-1,y+m-1)) {
                    T.col·loca(x,y,x+m-1,y+m-1);
                    recursiu(i+1);
                    if (trob) return;
                    T.retira(x,y);
}   }   }   }   }
```

   La crida inicial és `recursiu(0);`.

   [El cost d'aquest algorisme és $\mathrm{O}(n^{2k})$; hi ha altres solucions més complicades que poden ser una mica més eficients.]

   [Error freqüent: no provar de col·locar les fitxes a totes les posicions possibles.]

5. Recurrència:

$$sor(i,p,g) = \begin{cases} 0 \\ \qquad \text{si } i = 0, \\ sor(i-1,p,g) \\ \qquad \text{si } i \neq 0 \ \wedge \ (p[i] > p \ \vee \ g[i] > g), \\ \max\{sor(i-1,p-p[i],g-g[i]) + s[i] \ , \ sor(i-1,p,g)\} \\ \qquad \text{si } i \neq 0 \ \wedge \ p[i] \leqslant p \ \wedge \ g[i] \leqslant g. \end{cases}$$

   Cost: $\mathrm{O}(N \cdot P \cdot G)$.

   [Errors frequents: Ajuntar els dos casos per a $i > 0$ en un de sol i no afegir el cas base $-\infty$ quan $p < 0 \ \vee \ g < 0$; no seguir l'especificació de la funció *sor*; dir que l'algorisme té cost $\mathrm{O}(n^3)$ o cúbic, o quadràtic, o polinòmic... (exercici fàcil: demostreu que el problema decisional és **NP**-complet).]