

Problema 1 - Misteri (Anàlisi d'Algorismes)

[1 punt]

Considereu l'algorisme següent, camuflat misteriosament:

```
#include <algorithm>
#include <vector>
using namespace std;

void misteri(vector<int>& v, int i)
{
    if (i==0) i=1;
    if (i<v.size())
    {
        if (v[i-1]<= v[i]) misteri(v,i+1);
        else {
            swap(v[i-1],v[i]);
            misteri(v,i-1);
        }
    }
}

void misteri(vector<int>& v)
{
    misteri(v,0);
}
```

Es demana:

- a) Quin és el cas pitjor per a aquest algorisme?
- b) Analitzeu el cost temporal del cas pitjor de la crida `misteri(v)`.
- c) Digueu en una frase què fa aquest algorisme.
- d) Us recorda a algun altre algorisme?

Problema 2 - Bricolatge (Dividir i Vèncer)

[2 punts]

A la secció de bricolatge d'un gran magatzem ha arribat la darrera remesa de cargols i femelles que ha d'abastir el centre per a tot l'estiu. El material ha arribat en dues grans caixes: en una hi ha n cargols, tots d'una amplada diferent, i a l'altre hi ha les n femelles corresponents.

Per oferir el material cal posar-lo als prestatges d'una forma que sigui fàcil de trobar pels clients. Per això, s'ha d'aparellar cada cargol amb la seva femella abans de col·locar-lo al corresponent prestatge.

Malauradament, quan l'encarregat de la secció es disposava a aparellar cada cargol amb la seva femella corresponent, hi ha hagut un tall de corrent que ha deixat tot el centre a les fosques. A causa de la foscor no es poden llegir les mides exactes dels cargols ni tampoc de les femelles. L'única comparació possible és entre cargol i femella, intentant cargolar un cargol a una femella per comprovar si és massa gran, si és massa petit o si encaixa perfectament.

Es demana:

- a) Descriviu un algorisme de dividir i vèncer per aparellar cada cargol amb la seva femella en $\Theta(n \log n)$ passos en el cas mitjà.

b) Quin és el cas pitjor del vostre algorisme?

Pista 1: Penseu en un algorisme d'ordenació famós.

Pista 2: Un cargol es pot fer servir més d'una vegada per comparar-lo amb diverses femelles.

Problema 3 - La grip aviària (Cerca Exhaustiva) [2 punts]

La grip aviària és una malaltia contagiosa en animals causada per virus que normalment afecten aus i, menys habitualment, porcs. Entre aus de granja es contagia ràpidament i té una mortalitat que pot arribar al 100%, sovint en només 48 h.

En un laboratori de referència s'està estudiant el cep més perillós del virus entre una mostra d'aus salvatges recollides a les nostres costes. El laboratori disposa de n^2 gàbies col·locades formant un quadrat $n \times n$.

Es vol *col·locar el màxim nombre d'aus a les gàbies del laboratori*, però s'han de tenir en compte les restriccions següents:

- Algunes gàbies estan espatllades i no es poden utilitzar.
- Cada gàbia només pot contenir una au, com a molt.
- Quan es col·loca una au en una gàbia, les gàbies del voltant han de quedar buides (per tal d'evitar contagis).

Per solucionar aquest problema podríem fer cerca exhaustiva, tot modelitzant les gàbies com un taulell $n \times n$ (vegeu el dibuix).

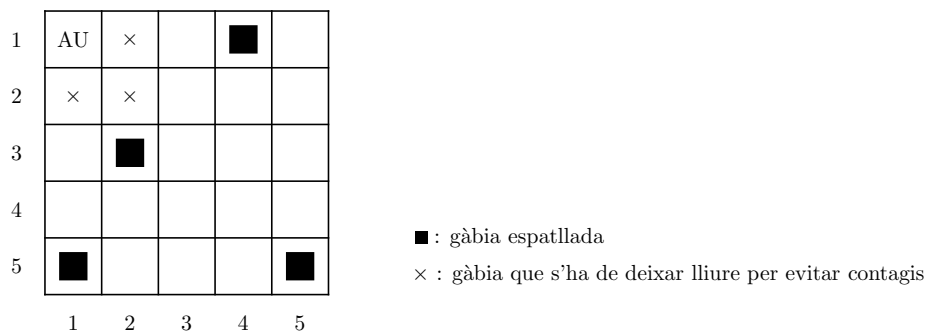


Figura 1: Les gàbies del laboratori vistes com un taulell

Es demana:

- a) Completeu la proposta d'algorisme de backtracking que trobareu a la plana següent per tal que calculi quin és el nombre màxim d'aus que es poden estudiar al laboratori.

```


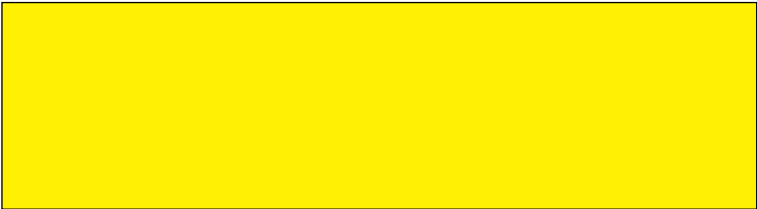
#include "ada.hh"

class Laboratori {
    int n; // nombre de files i columnes
    matrix<int> G; // les gàbies del laboratori
    int maxim; // màxim nombre d'aus que es poden engabiar

    static const int ESPATLLADA = -1;
    static const int LLIURE = 0;
    static const int OCUPADA = 1;

    bool dintre_laboratori (int x, int y) {
        return (x>=0 and x<n and y>=0 and y<n);
    }

    bool perill_contagi (int x, int y) {
        // Determina si la cassella (x,y) està en perill de contagi
        return ((dintre_laboratori(x-1,y-1) and G[x-1][y-1]==OCUPADA) or
                (dintre_laboratori(x-1,y) and G[x-1][y]==OCUPADA) or
                (dintre_laboratori(x-1,y+1) and G[x-1][y+1]==OCUPADA) or
                (dintre_laboratori(x,y-1) and G[x][y-1]==OCUPADA));
    }

    void recursiu (int cas, int nombre_aus) {
        if (cas==n*n) {
            
        } else {
            
        }
    }
}

public:

    GripAviaria (int n) {
        this->n = n;
        G = matrix<int>(n,n,LLIURE);
        maxim = 0;
    }

    void inhabilita (int x, int y) {
        if (dintre_laboratori(x,y)) G[x][y]=ESPATLLADA;
    }

    int configura () {
        recursiu(0,0);
        return maxim;
    }
};

// PROGRAMA PRINCIPAL

int main () {
    int n, x, y;
    cin >> n;
    Laboratori lab(n);
    while(cin >> x) { // Llegeix la informació de gàbies espatllades
        cin >> y;
        lab.inhabilita(x,y);
    }
    cout << lab.configura() << endl;
}

```

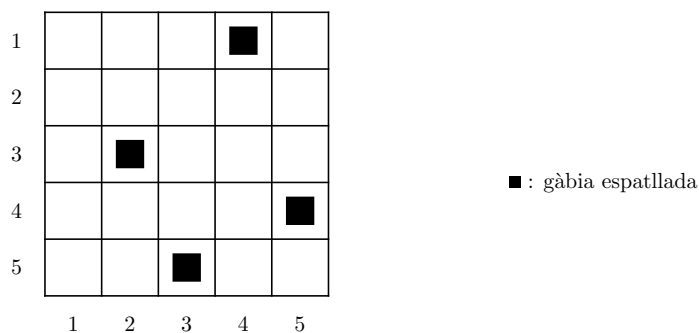
- b) Doneu la versió decisional del problema i demostreu que és a NP.

Problema 4 - La grip aviària (Programació Dinàmica) [3 punts]

Recordeu que al laboratori de referència del problema anterior es troben engabiades aus salvatges per estudiar-ne la grip aviària. El laboratori disposa de n^2 gàbies col·locades formant un quadrat $n \times n$, a on algunes gàbies es troben espatllades (vegeu el dibuix).

Amb les aus a estudiar ja col·locades per dur a terme els estudis pertinents, el becari del laboratori ha d'anar a una gàbia (x, y) determinada i, si està ocupada, prendre un seguit de mostres de l'au que hi viu.

Degut a la distribució del laboratori, el recorregut del becari ha de començar a la gàbia $(1, 1)$. Per avançar pel laboratori des d'una determinada gàbia (i, j) , el becari només pot accedir a la següent gàbia de la dreta o a la següent gàbia de sota.



Tenint en compte que passar per una gàbia no espatllada té un cost constant c_1 i que passar per una gàbia espatllada té un cost constant c_2 (amb $1 \leq c_1 < c_2$), es planteja el problema següent: *calcular el cost del camí més barat que pot conduir al becari des de la gàbia $(1, 1)$ a la gàbia (x, y) donada*. Es demana:

- Expliqueu per què té sentit aplicar l'esquema de Programació Dinàmica per resoldre aquest problema.
- Doneu una recurrència que el resolgui, explicant clarament què descriu el terme de la recurrència. Quin terme d'aquesta recurrència calcula l'objectiu del problema?
- Descriviu l'estructura de dades auxiliar que guardarà els resultats dels subproblemes ja resolts i expliqueu amb detall com s'omple l'estructura.
- Justifiqueu quin és el cost, espacial i temporal, de la vostra solució.

Problema 5 - Recorreguts sobre grafs [2 punts]

Sigui $G = (V, E)$ un graf dirigit. Fent servir les definicions utilitzades al llarg del curs:

```
#include "ada.hh"

typedef vector< list<int> > graph;
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<int((G).size()); ++u)
```

es demana:

- a) Modifiquen l'algorisme de recorregut en profunditat perquè calculi el grau d'entrada i el grau de sortida de tots els vèrtexs del graf.

```
void compta_graus (graph& G)
{
    vector<int> grau_ent(G.size(),0);
    vector<int> grau_sor(G.size(),0);
    vector<boolean> vis(G.size(),false);

    forall_ver(u,G) {
        if (not vis[u]) compta_graus(G,u,vis,grau_ent,grau_sor);
    }
    cout << "Graus d'entrada: " << grau_ent << endl;
    cout << "Graus de sortida: " << grau_sor << endl;
}

void compta_graus(graph& G, int u, vector<boolean>& vis, vector<int>& gent,
                  vector<int>& gsor)
{
    
}
```

- b) Feu el mateix però modificant l'algorisme de recorregut en amplada.

```

void compta_graus (graph& G)
{
    vector<int> grau_ent(G.size(),0);
    vector<int> grau_sor(G.size(),0);
    vector<boolean> vis(G.size(),false);

    forall_ver(u,G) {
        if (not vis[u]) compta_graus(G,u,vis,grau_ent,grau_sor);
    }
    cout << "Graus d'entrada: " << grau_ent << endl;
    cout << "Graus de sortida: " << grau_sor << endl;
}

void compta_graus(graph& G, int u, vector<boolean>& vis, vector<int>& gent,
                  vector<int>& gsor)
{

}

```

SOLUCIONS

Problema 1

- a) Quan el vector d'entrada està ordenat de forma decreixent
- b) $\Theta(n^2)$
- c) d) És un algorisme d'ordenació (conegut com *Gnome sort*). És similar a l'ordenació per inserció excepte que el fet de moure un element al seu lloc corresponent es fa mitjançant intercanvis, com en l'algorisme de la bombolla.

Problema 2

Provar un cargol amb totes les femelles possibles, trobar la femella corresponent i separar la resta de femelles que queden en dos grups: femelles més petites i femelles més grans. Pels següents cargols provar-los amb la femella del primer per decidir amb quin conjunt s'ha de comparar i repetir el procediment recursivament per cada subgrup de cargols i femelles. *Un altre possibilitat: Un cop trobada la primera parella (cargol i femella) i tenint el conjunt de les femelles dividit en dos subgrups (FMP i FMG), agafar la femella aparellada i comparar-la amb tots els cargols (excepte amb el que encaixa, és clar!). Després d'aquesta comparació obtenim dos nous subgrups de cargols, els més grans CMG i els més petits CMP i podem*

repetir el procés però ara amb els conjunts *FMP* i *CMP* (que tindran la mateixa mida perquè són els cargols i les femelles més petits comparats amb la parella trobada) i amb els conjunts *FMG* i *CMG* (que corresponen a les femelles i cargols més grans) .

El cost pitjor d'aquest algorisme és el mateix que el cost pitjor del Quicksort i és $\Theta(n^2)$. Aquest cost es dona quan sempre s'escull el pivot (cargol o femella) més gran (o més petit) de tot el conjunt a emparellar perquè produeix dues particions absolutament descompensades: una té 0 elements i l'altre $n - 1$.

Problema 3

a)

```
void recursiu (int cas, int nb_au) {
    if (cas==n*n) {
        if (nb_au>maxim) maxim = nb_au;
    } else {
        int i = cas/n, j = cas%n;
        if (G[i][j]==ESPATLLADA or perill_contagi(i,j)) {
            recursiu(cas+1,nb_au);
        } else {
            G[i][j] = OCUPADA;
            recursiu(cas+1,nb_au+1);
            G[i][j] = LLIURE;
            recursiu(cas+1,nb_au);
        }
    }
}
```

b) Sigui T un taulell $n \times n$ que representa les gàbies i té marcades les que no estan disponibles.

Versió funcional: Donat T , trobar el màxim nombre d'aus que s'hi poden col·locar tenint en compte les restriccions que (1) les gàbies marcades a T no es poden utilitzar perquè estan espatllades, (2) cada gàbia només pot contenir una au, i (3) quan es col·loca una au en una gàbia, les gàbies del voltant han de quedar buides.

Versió decisional: Donat T i un enter k , decidir si es poden col·locar al menys k aus a T , tot respectant les restriccions que (1) les gàbies marcades a T no es poden utilitzar perquè estan espatllades, (2) cada gàbia només pot contenir una au, i (3) quan es col·loca una au en una gàbia, les gàbies del voltant han de quedar buides.

Observeu que, fent servir la versió decisional del problema i variant el valor de k (per exemple com indica una cerca dicotòmica), podem resoldre la versió funcional del problema.

Aquest problema és a **NP** perquè hi ha un testimoni de mida polinòmica (un taulell amb aus col·locades a certes caselles) i un algorisme verificador que rep el taulell T (amb les gàbies espatllades), la k i el taulell TE (testimoni) i en temps polinòmic el verificador torna cert si TE conté k o més aus col·locades tot respectant les restriccions de T i de la manera de disposar les aus (torna fals quan falla alguna de les condicions). Quan aquest verificador torna cert podem assegurar que la versió decisional del problema té solució i que el testimoni TE és una de elles.

Problema 4

a) Perquè el problema es pot dividir en subproblemes i compleix els dos criteris necessaris per aplicar programació dinàmica: la propietat de *subestructura òptima* (solucions òptimes a subproblemes es poden utilitzar per trobar solucions òptimes al problema complet), i *subproblemes repetits o solapats* (el mateix subproblema s'ha de resoldre diverses vegades en intentar resoldre el problema complet).

- b) Sigui $C(i, j)$ el cost del camí més barat per arribar des de la gàbia $(1, 1)$, fins la gàbia (i, j) (només amb moviments cap a la dreta i cap a baix). Segon això, l'objectiu del problema és calcular $C(x, y)$.

Aleshores, $C(i, j)$ es calcula segons la següent recurrència:

$$C(i, j) = \begin{cases} \chi(i, j) & \text{si } i = j = 1 \\ \chi(i, j) + C(i, j - 1) & \text{si } i = 1, \text{ i } 1 < j \leq n \\ \chi(i, j) + C(i - 1, j) & \text{si } j = 1, \text{ i } 1 < i \leq n \\ \chi(i, j) + \min \{C(i - 1, j), C(i, j - 1)\} & \text{altrament} \end{cases}$$

on $\chi(i, j)$ és una variable indicadora del fet que la gàbia (i, j) està espatllada o no,¹ és a dir:

$$\chi(i, j) = \begin{cases} c_1 & \text{si la gàbia } (i, j) \text{ no està espatllada} \\ c_2 & \text{si la gàbia } (i, j) \text{ està espatllada} \end{cases}$$

En la recurrència, el cas base correspon al cost de passar per la gàbia $(1, 1)$. El primer cas recursiu correspon a emplenar la primera “fila” del laboratori. El segon cas recursiu correspon a emplenar la primera “columna” del laboratori.

- c) Es va considerar correcta com a resposta una matriu \mathbb{C} de talla $x \times y$ (o de talla $n \times n$), on $\mathbb{C}[i][j] = C(i, j)$ tot i que amb un vector de y posicions hi ha prou ja que només interessa la darrera fila calculada de la matriu anterior. La següent fila es calcula sobreescrivint la fila i de dreta a esquerra. Per omplir la matriu, s’han de respectar les dependències entre valors que descriu la recurrència. Una forma d’emplenar la matriu respectant aquestes restriccions seria emplenant-la per files (però n’hi ha d’altres).
- d) El cost temporal és $\Theta(xy)$ o $O(n^2)$ (quan el becari ha d’anar a la gàbia (n, n)). I com que amb una fila de mida y n’hi ha prou, el cost espacial és $\Theta(y)$ o $O(n)$.

Problema 5

- a)
- ```
void compta_graus (graph& G)
{
 vector<int> grau_ent(G.size(),0);
 vector<int> grau_sor(G.size(),0);
 vector<boolean> vis(G.size(),false);

 forall_ver(u,G) {
 if (not vis[u]) compta_graus(G,u,vis,grau_ent,grau_sor);
 }
 cout << "Graus d'entrada: " << grau_ent << endl;
 cout << "Graus de sortida: " << grau_sor << endl;
}

void compta_graus(graph& G, int u, vector<boolean>& vis, vector<int>& gent,
 vector<int>& gsor)
{
 vis[u]=true;
 forall_adj(uv,G[u]) {
 gent[*uv]++;
 if (not vis[*uv]) compta_graus(G,*uv,vis,gent,gsor);
 gsor[u]++;
 }
}
```
- b)

<sup>1</sup>També es podria prescindir d'aquesta variable i expressar tots els casos de la recurrència (els casos quan  $\chi(i, j) = c_1$  i els casos quan  $\chi(i, j) = c_2$ ) per separat.



```

void compta_graus (graph& G)
{
 vector<int> grau_ent(G.size(),0);
 vector<int> grau_sor(G.size(),0);
 vector<boolean> vis(G.size(),false);

 forall_ver(u,G) {
 if (not vis[u]) compta_graus(G,u,vis,grau_ent,grau_sor);
 }
 cout << "Graus d'entrada: " << grau_ent << endl;
 cout << "Graus de sortida: " << grau_sor << endl;
}

void compta_graus(graph& G, int u, vector<boolean>& vis, vector<int>& gent,
 vector<int>& gsor)
{
 queue<int> Q;
 Q.push(u);
 vis[u]=true;
 while (not Q.empty()) {
 int v = Q.front(); Q.pop();
 forall_adj(vw,G[v]) {
 int w=*vw;
 gent[w]++;
 gsor[v]++;
 if (not vis[w]) {
 Q.push(w);
 vis[w]=true;
 }
 }
 }
}

```