Anàlisi i Disseny d'Algorismes

Transparències

16 de setembre de 2005

Jordi Petit
Salvador Roura
Albert Atserias

Anàlisi i Disseny d'Algorismes

Introducció al C++

Entrada/Sortida bàsica

Programa en C++ que escriu un missatge per la sortida estàndard.

```
#include <iostream>
using namespace std;

int main () {    // Funció principal
    cout << "Hola a tothom!" << endl;
}
```

Llegeix dos números per l'entrada estàndard i n'escriu la suma per la sortida estàndard.

```
#include <iostream>
using namespace std;

int main () {
    int a,b;
    cin >> a >> b;
    int s = a + b;
    cout << a << " + " << b << " = " << s << endl;
}
```

Calcula la mitjana d'una seqüència de reals llegida per l'entrada estàndard.

```
#include <iostream>
using namespace std;

int main () {
    double s = 0;
    int n = 0;
    double x;
    while (cin >> x) {
        s += x;
        ++n;
    }
    cout << s/n << endl;
}
```

Funcions

Funció que calcula el màxim de dos enters.

```
#include <iostream>
using namespace std;

int max (int a, int b) {
    return a>b ? a : b;
}

int main () {
    int x,y;
    cin >> x >> y;
    cout << max(x,y) << endl;
}
```

Sobrecàrrega de noms.

```
#include <iostream>
#include <string>
using namespace std;

int max (int a, int b) {
    return a>b ? a : b;
}

double max (double a, double b) {
    return a>b ? a : b;
}

string max (string a, string b) {
    return a>b ? a : b;
}

int main () {
    cout << max(3,4) << endl;
    cout << max(3.5,12.3) << endl;
    cout << max(3,15.0) << endl;
    cout << max("Hola","Adeu") << endl;
}
```

Funció màx genèrica.

```
#include <iostream>
#include <string>
using namespace std;

template <typename tipus>
tipus max (tipus a, tipus b) {
    return a>b ? a : b;
}

int main () {
    cout << max(3,4) << endl;
    cout << max(1.12,3.14) << endl;
    cout << max("bon dia","bona tarda") << endl;
    cout << max('a','d') << endl;
}
```

Intercanvi incorrecte: Paràmetres per valor.

```
#include <iostream>
using namespace std;

void swap (int x, int y) {
    int z = x;
    x = y;
    y = z;
}

int main () {
    int a,b;
    cin >> a >> b;
    swap(a,b);
    cout << a << b;
}
```

Intercanvi correcte: Paràmetres per referència.

```
#include <iostream>
using namespace std;

void swap (int& x, int& y) {
    int z = x;
    x = y;
    y = z;
}

int main () {
    int a,b;
    cin >> a >> b;
    swap(a,b);
    cout << a << b;
}
```

Funció d'intercanvi genèrica.

```
#include <iostream>
using namespace std;

template <typename T>
void swap (T& x, T& y) {
    T z = x;
    x = y;
    y = z;
}

int main () {
    int a,b;
    cin >> a >> b;
    swap(a,b);
    cout << a << b;
}
```

Vectors

Declaracions de vectors.

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

int main () {
    // Vector per a 10 enters
    vector<int> v1(10);

    // Vector per a 20 reals, tots inicialitzats a 1
    vector<double> v2(20,1);

    // Vector de caràcters, sense capacitat
    vector<char> v3;

    // Vector de booleans, eficient en espai però no en temps
    vector<bool> vb(1000);

    // Matriu de 10 × 20 reals
    vector< vector<double> > M(10,20);
}
```

Ús de vectors:

- `[]` accedeix a les posicions (compte si fora de rang!),
- `size()` retorna la seva talla,
- `empty()` diu si la talla és zero,
- `front()` retorna el primer element,
- `back()` retorna el darrer element,
- `push_back()` afageix un element pel final del vector,
- els vectors es poden assignar entre ells.

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

int main () {
    vector<int> v(10);
    for (int i=0; i<v.size(); ++i) {
        v[i] = i;
    }

    vector<int> v2 = v;
    v2.push_back(49);
    cout << v2.front() << " " << v2.back() << endl;
    v2.pop_back();
    cout << v2.empty() << endl;

    vector<int> v3 = v2;
}
```

Llistes i iteradors

Llistes.

- el constructor crea una llista buida,
- `push_back()` afegeix un element pel final,
- `push_front()` afegeix un element pel davant,
- `pop_back()` esborrar un element pel final,
- `pop_front()` esborrar un element pel davant,
- `front()` retorna el primer element,
- `back()` retorna el darrer element,
- `size()` retorna la seva talla,
- `empty()` diu si la talla és zero,
- altres mètodes: `sort()`, `reverse()`, `merge()`, `unique()`, `splice()`, ...

```
#include <iostream>
#include <list>
using namespace std;

int main () {
    list<int> L;
    L.push_back(3); L.push_back(4); L.push_back(5);
    L.push_front(2); L.push_front(1); L.push_front(0);
    cout << L.size()  << " " << L.empty() << endl;
    cout << L.front() << " " << L.back()  << end;

    list<int> L2 = L;
    L2.reverse();
    cout << L2.front() << " " << L2.back()  << end;
    L2.sort();
    cout << L2.front() << " " << L2.back()  << end;
}
```

Iteradors.

Els iteradors permeten recorrer els contenidors com ara les llistes. Els mètodes `begin()` i `end()` retornen iteradors. L'iterador a `end()` senyala el final de la llista (no s'hi pot accedir).

```
#include <iostream>
#include <list>
using namespace std;

typedef list<int>::iterator iter;

void escriure_llista (list<int>& L) {
    for (iter it=L.begin(); it!=L.end(); ++it) {
        cout << *it << endl;
}   }

bool hi_es (list<int>& L, int x) {
    iter it = L.begin();
    while (it!=L.end() and *it!=x) ++it;
    return it!=L.end();
}

bool hi_es2 (list<int>& L, int x) {
    for (iter it=L.begin(); it!=L.end(); ++it) {
        if (*it==x) return true;
    }
    return false;
}
```

Iteradors.

Els iteradors també permeten fer insercions en punts qualsevols. En aquest exemple es mostra una ordenació per inserció.

```
#include <iostream>
#include <list>
using namespace std;

typedef list<int> llista;
typedef llista::iterator iter;

iter cerca_pos (llista& L, int x) {
    iter it=L.begin();
    while (it!=L.end() and *it<x) ++it;
    return it;
}

llista ordenar (llista& L) {
    llista L2;
    for (iter it=L.begin(); it!=L.end(); ++it) {
        iter it2 = cerca_pos(L2,*it);
        L2.insert(it2,*it);
    }
    return L2;
}
```

Classes

Les estructures permeten agrupar diverses dades en una de sola.

```
struct data {
    int dia,mes,any;
};

struct persona {
    string nom,cognom1,cognom2;
    data naix;
};


...

data d;
d.dia = 30; d.mes = 11; d.any = 1971;

persona p;
p.nom = "Jordi";
p.naix = d;
cout << p.naix.any;
```

Les estructures poden definir constructors, destructors i mètodes.

```
struct data {

    int dia,mes,any;

    data (int d, int m, int a) {
        dia = d;  mes = m;  any = a;
    }

    bool legal () {
        ...
    }

    void incrementa () {
        ...
    }

};

...
data d(30,11,1971);
if (d.legal()) {
    data d2 = d;
    d2.incrementa();
}
```

Classes.

Les classes són el mateix que les estructures, però en elles se sol definir una part privada que implementa una interfície pública.

```
class data {

private:

    int dia,mes,any;

public:

    data (int d, int m, int a) {
        dia = d;  mes = m;  any = a;
    }

    bool legal () { ... }

    void incrementa () { ... }

};

...
data d(30,11,1971);
if (d.legal()) {
    data d2 = d;
    d2.incrementa();
}
```

Operadors.

Es poden definir operadors aritmètics i relacionals.

```
class data {

private:

  int dia,mes,any;

public:

  data (int d, int m, int a) {
      dia = d;  mes = m;  any = a;
  }

  bool legal () { ... }

  data& operator++ () {...}
  data& operator+= (int n) {...}

  friend static bool operator<  (data d1, data d2) {...}
  friend static bool operator== (data d1, data d2) {...}
  ...
};
```

Punters

Punters.

```
// Declaració d'un punter p a un enter:

int* p;

// Assignació d'un punter nul a p:

p = 0;
p = null;          // null és una constant definida a <ada.hh>

// Agafar memòria dinàmica per a un enter
// i guardar-ne el seu punter en p:

p = new int;

// Desreferenciar el punter:
// (evidentment, no es pot seguir mai un punter nul)

*p = 21;
if (*p==13) ...

// Alliberar la memòria dinàmica

delete p;
```

```
// Els punters que apunten a un mateix tipus es poden assignar
// (amb =) i comparar (amb == o !=).

int* p1 = ...;
int* p2 = ...;
int* p3 = ...;
if (p1==p2) {
    p3 = p1;
}

// Els punters es converteixen automàticament en booleans si cal:
// nul es converteix a fals, no nul es converteix a cert.

int* p = ...;
if (p and *p!=24) ...
```

Punters i estructures (o classes).

En aquest curs sempre usarem els punters per apuntar a estructures. En aquest cas `*punter.camp` no funciona, cal escriure `(*punter).camp` o, més fàcil, `punter->camp`.

En aquest exemple es mostra un exemple per a definir llistes enllaçades.

```
struct node {
    node* seg;
    int   inf;
};

typedef node* llista;

bool hi_es (llista p, int x) {
    node* q = p;
    while (q and q->inf!=x) q = q->seg;
    return q;
}
```

Classes i punters.

El fet d'utilitzar memòria dinàmica per a implementar classes fa que calgui anar en compte amb no tenir problemes d'aliasing i fuites de memòria. Per això, en aquest cas, sovint caldrà definir operadors i constructors de còpia i destructors.

Vegeu l'exemple de la pila.

Exemple complet: Pila genèrica

Exemple complet: Pila genèrica amb memòria dinàmica.

```
template <typename T>
class Pila {

    struct node {
        T x;                // informació
        node* s;            // punter al node següent
    };

    int n;                  // nombre d'elements
    node* c;                // punter al cim

public:

    // Constructor: crea una pila buida
    Pila () {
        c = null;  n = 0;
    }

    // Constructor de còpia
    Pila (const Pila& P) {
        c = copiar(P.c);  n = P.n;
    }

    // Destructor
    ~Pila () {
        alliberar(c);
    }
```

```
// Operador de còpia
Pila& operator= (const Pila& P) {
    if (&P!=this) {
        alliberar(c);  c = copiar(P.c);  n = P.n;
    }
    return *this;
}

// Apila x sobre la pila
void apilar (T x) {
    node* p = new node;  p->x = x;  p->s = c;
    c = p;  ++n;
}

// Treu l'element del cim de la pila (error si és buida)
void desapilar () {
    if (buida()) throw ErrorPrec("Pila buida");
    node* p = c;  c = c->s;  delete p;  --n;
}

// Retorna l'element del cim de la pila (error si és buida)
T cim () {
    if (buida()) throw ErrorPrec("Pila buida");
    return c->x;
}

// Indica si la pila és buida
bool buida () {
    return n==0;
}
```

```
    // Retorna el nombre d'elements a la pila
    int talla () {
        return n;
    }

private:

    void alliberar (node* p) {
        if (p) {
            alliberar(p->s);
            delete p;
    }   }

    node* copiar (node* p) {
        if (p) {
            node* q = new node;
            q->x = p->x;
            q->s = copiar(p->s);
            return q;
        } else {
            return null;
    }   }

};
```

```
// Exemple d'ús:

int main () {
    Pila<int> p;
    p.apilar(3); p.apilar(5);
    while (!p.buida()) {
        cout << p.cim() << endl;
        p.desapilar();
}   }
```

Anàlisi i Disseny d'Algorismes

Algorismes d'ordenació

Ordenació per selecció.

```
template <typename elem>
void sel_sort (vector<elem>& T) {
    int n = T.size();
    for (int i=0; i<n-1; ++i) {
        int p = pos_min(T,i,n-1);
        swap(T[i],T[p]);
}   }

template <typename elem>
int pos_min (vector<elem>& T, int e, int d) {
    int p = e;
    for (int j=e+1; j<=d; ++j) {
        if (T[j] < T[p]) {
            p = j;
    }   }
    return p;
}
```

Ordenació per inserció (primera versió).

```
template <typename elem>
void ins_sort_1 (vector<elem>& T) {
    int n = T.size();
    for (int i=1; i<n; ++i) {
        for (int j=i; j>0 and T[j-1]>T[j]; --j) {
            swap(T[j-1],T[j]);
}   }   }
```

Ordenació per inserció (segona versió).

Aquest cop es deixen de fer cadenes d'intercanvis: en lloc de fer intercanvis, es desplacen els elements cap a la dreta i es col·loca l'element a inserir al final (reduïm de 3 assignacions per volta a 1).

```
template <typename elem>
void ins_sort_2 (vector<elem>& T) {
    int n = T.size();
    for (int i=1; i<n; ++i) {
        elem x = T[i];
        int j;
        for (j=i; j>0 and T[j-1]>x; --j) {
            T[j] = T[j-1];
        }
        T[j] = x;
}   }
```

Ordenació per inserció (tercera versió).

Aquest cop, per evitar el test de final a cada iteració, es col·loca l'element més petit a la primera posició de la taula.

```
template <typename elem>
void ins_sort_3 (vector<elem>& T) {
    int n = T.size();
    swap(T[0],T[pos_min(T,0,n-1)]);
    for (int i=2; i<n; ++i) {
        elem x = T[i];
        int j;
        for (j=i; T[j-1] > x; --j) {
            T[j] = T[j-1];
        }
        T[j] = x;
}   }
```

Ordenació per bombolla.

```
template <typename elem>
void bubble_sort (vector<elem>& T) {
    int n = T.size();
    for (int i=0; i<n-1; ++i) {
        for (int j=n-1; j>i; --j) {
            if (T[j-1] > T[j]) {
                swap(T[j-1],T[j]);
}   }   }   }
```

Ordenació per fusió (1a versió).

```
template <typename elem>
void merge_sort_1 (vector<elem>& T) {
    merge_sort_1(T,0,T.size()-1);
}

template <typename elem>
void merge_sort_1 (vector<elem>& T, int e, int d) {
    if (e<d) {
        int m = (e+d)/2;
        merge_sort_1(T,e,m);
        merge_sort_1(T,m+1,d);
        merge(T,e,m,d);
}   }
```

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d-e+1);
    int i = e, j = m+1, k = 0;
    while (i<=m and j<=d) {
        if (T[i]<=T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];
    }
    while (i<=m) B[k++] = T[i++];
    while (j<=d) B[k++] = T[j++];
    for (k=0; k<=d-e; ++k) T[e+k] = B[k];
}
```

Ordenació per fusió (2a versió).

Tallem la recursió quan el subvector a ordenar és “prou petit” i llavors usem ordenació per inserció.

```
template <typename elem>
void merge_sort_2 (vector<elem>& T) {
    merge_sort_2(T,0,T.size()-1);
}

template <typename elem>
void merge_sort_2 (vector<elem>& T, int e, int d) {
    const int talla_critica = 50;
    if (d-e<talla_critica) {
        ins_sort(T,e,d);
    } else {
        int m = (e+d)/2;
        merge_sort_2(T,e,m);
        merge_sort_2(T,m+1,d);
        merge(T,e,m,d);
}   }
```

Ordenació per fusió d’avall cap amunt.

```
template <typename elem>
void merge_sort_bu (vector<elem>& T) {
    int n = T.size();
    for (int m=1; m<n; m*=2) {
        for (int i=0; i<n-m; i+=2*m) {
            merge(T, i, i+m-1, min(i+2*m-1,n-1) );
}   }   }
```

Ordenació ràpida (partició de Hoare).

```
template <typename elem>
void quick_sort_1 (vector<elem>& T) {
    quick_sort_1(T,0,T.size()-1);
}

template <typename elem>
void quick_sort_1 (vector<elem>& T, int e, int d) {
    if (e<d) {
        int q = partition(T,e,d);
        quick_sort_1(T,e,q);
        quick_sort_1(T,q+1,d);
}   }

template <typename elem>
int partition (vector<elem>& T, int e, int d) {
    elem x = T[e];   int i = e-1;  int j = d+1;
    for (;;) {
        while (x < T[--j]);
        while (T[++i] < x);
        if (i>=j) return j;
        swap(T[i],T[j]);
}   }
```

Ordenació ràpida (2a versió).

Tria de pivot aleatòria.

```
template <typename elem>
void quick_sort_2 (vector<elem>& T) {
    quick_sort_2(T,0,T.size()-1);
}

template <typename elem>
void quick_sort_2 (vector<elem>& T, int e, int d) {
    if (e<d) {
        int p = randint(e,d);
        swap(T[e],T[p]);
        int q = partition(T,e,d);
        quick_sort_2(T,e,q);
        quick_sort_2(T,q+1,d);
}   }
```

Ordenació ràpida (3a versió).

Aquest cop es deixa d'ordenar quan el subvector és "prou petit". Al final, es fa una darrera passada amb ordenació per inserció.

```
template <typename elem>
void quick_sort_3 (vector<elem>& T) {
    quick_psort_3(T,0,T.size()-1);
    ins_sort(T,0,T.size()-1);
}

template <typename elem>
void quick_psort_3 (vector<elem>& T, int e, int d) {
    const int talla_critica = 100;
    if (d-e>=talla_critica) {
        int q = partition(T,e,d);
        quick_psort_3(T,e,q);
        quick_psort_3(T,q+1,d);
}   }
```

Heap sort amb TAD

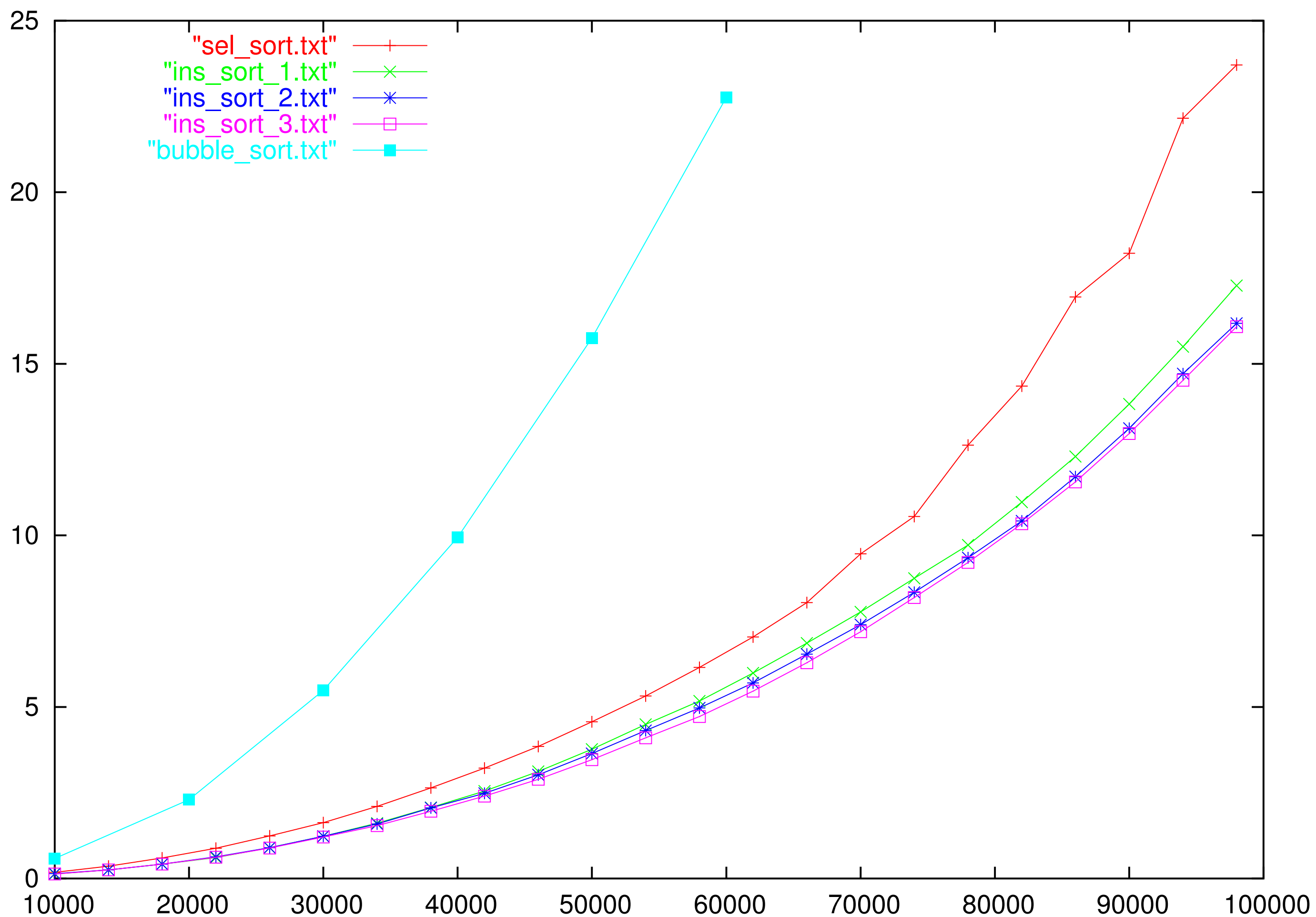
```
template <typename elem>
void heap_sort_0 (vector<elem>& T) {
    int n = T.size();
    priority_queue<elem> pq;
    for (int i=0; i<n; ++i) {
        pq.push(T[i]);
    }
    for (int i=n-1; i>=0; --i) {
        T[i] = pq.top();
        pq.pop();
}   }
```

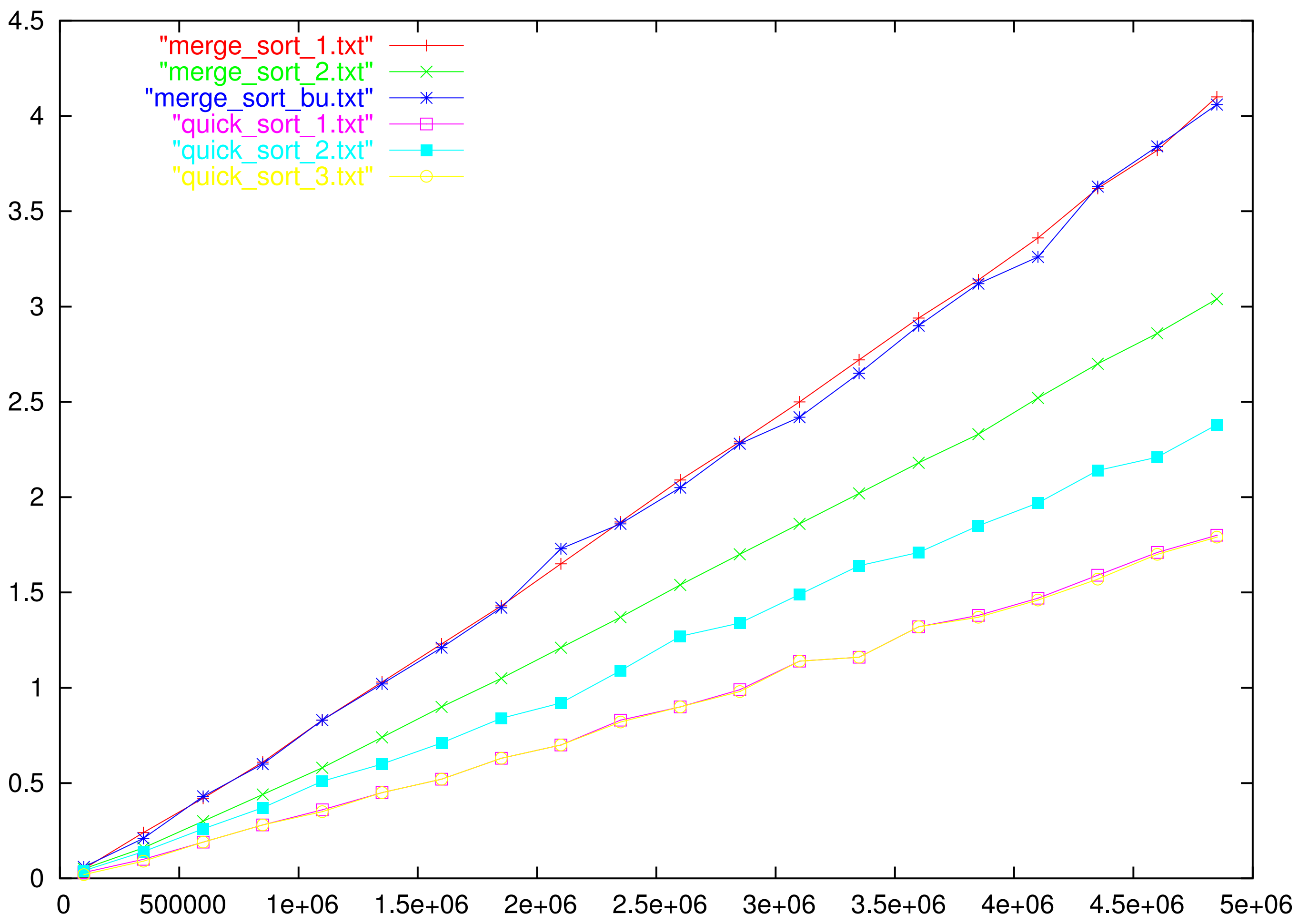
Heap sort

```
template <typename elem>
void heap_sort (vector<elem>& T) {
    int n = T.size();
    make_heap(T);
    for (int i=n-1; i>=1; --i) {
        swap(T[0],T[i]);
        sink(T,i,0);
}   }
```

```
template <typename elem>
void make_heap (vector<elem>& T) {
    int n = T.size();
    for (int i=n/2-1; i>=0; i--) {
        sink(T,n,i);
}   }

template <typename elem>
void sink (vector<elem>& T, int n, int i) {
    elem x = T[i];
    int c = 2*i+1;
    while (c<n) {
        if (c+1<n and T[c]<T[c+1]) c++;
        if (x>=T[c]) break;
        T[i] = T[c];
        i = c;
        c = 2*i+1;
    }
    T[i] = x;
}
```


"sel_sort.txt"
"ins_sort_1.txt"
"ins_sort_2.txt"
"ins_sort_3.txt"
"bubble_sort.txt"
25
20
15
10
5
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000


"merge_sort_1.txt"
"merge_sort_2.txt"
"merge_sort_bu.txt"
"quick_sort_1.txt"
"quick_sort_2.txt"
"quick_sort_3.txt"
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
5e+06

Anàlisi i Disseny d'Algorismes

Diccionaris

Taules de dispersió encadenades

Implementació d'un diccionari amb una taula de dispersió amb encadenament separat.

Se suposa que es té una funció `hash` que, donada una clau, retorna un natural en temps $\Theta(1)$.

Aquesta implementació organitza els elements del diccionari en `M` llistes encadenades. La i-èssima llista encadenada conté tots els parells clau/informació $\langle c, k \rangle$ tals que $\texttt{hash}(k) = i$. Cada llista es guarda en una taula `t` amb `M` posicions. Hi ha un comptador `n` que manté el nombre d'elements emmagatzemats al diccionari.

Els costos en el cas mitjà pressuposen que la funció de dispersió és bona i que $n/M = \Theta(1)$.

El cost de les còpies i del destructor és $\Theta(n + M)$. No cal definir aquestes operacions perquè les implícites funcionen bé.

```
#include <utility>
#include <list>
#include "ada.hh"

template <typename Clau, typename Info>
class Diccionari {
```

```
private:

    typedef pair<Clau,Info> Parella;
    typedef list<Parella> Llista;
    typedef typename Llista::iterator iter;

    vector<Llista> t;    // Taula de dispersió
    int n;               // Nombre d'elements
    int M;               // Nombre de posicions

public:
```

....

Constructor. Crea un diccionari buit.

Cost: $\Theta(M)$.

....

```
    Diccionari (int M = 1009)
    :   t(M), n(0), M(M) { }
```

. . . .

Insereix la clau `clau` dins del diccionari amb informació `info`.

Si la clau ja hi és, canvia la informació associada.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà: $\Theta(n/M)$.

. . . .

```
void assignar (Clau& clau, Info& info) {
    int h = hash(clau)%M;
    iter p = cerca(clau,t[h]);
    if (p!=t[h].end()) {
        p->second = info;
    } else {
        t[h].push_back(Parella(clau,info));
        ++n;
}   }
```

. . . .

Esborrar la clau `clau` del diccionari amb la seva informació associada. Si la clau no hi és, el diccionari no canvia.
Cost en el cas pitjor: $\Theta(n)$.
Cost en el cas mitjà: $\Theta(n/M)$.

. . . .

```
void esborrar (Clau& clau) {
    int h = hash(clau)%M;
    iter p = cerca(clau,t[h]);
    if (p!=t[h].end()) {
        t[h].erase(p);
        --n;
}   }
```

. . . .

Retorna una referència a la informació associada a `clau` dins del diccionari.

Llança una excepció de precondició si la clau no és present.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà: $\Theta(n/M)$.

. . . .

```
Info& consulta (Clau& clau) {
    int h = hash(clau)%M;
    iter p = cerca(clau,t[h]);
    if (p!=t[h].end()) {
        return p->second;
    } else {
        throw ErrorPrec("La clau no era present");
}   }
```

. . . .

Indica si la clau és o no present dins del diccionari.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà: $\Theta(n/M)$.

. . . .

```
bool present (Clau& clau) {
    int h = hash(clau)%M;
    iter p = cerca(clau,t[h]);
    return p!=t[h].end();
}
```

. . . .

Retorna la talla del diccionari.

Cost: $\Theta(1)$.

. . . .

```
int talla () {
    return n;
}
```

```
private:
```

Cerca `c` en `L` i en retorna un iterador (`end()` si no hi és).

```
    static iter cerca (Clau& c, list<Parella>& L) {
        iter p = L.begin();
        while (p!=L.end() and p->first!=c) {
            ++p;
        }
        return p;
    }

};
```

Taules de dispersió no encadenades

Implementació d'un diccionari amb una taula de dispersió no encadenada amb memòria estàtica i exploració lineal.

Se suposa que es té una funció `hash` que, donada una clau, retorna un natural en temps $\Theta(1)$.

La taula consta de `M` posicions, on una posició és una tripla clau, informació i estat. Els possibles valor de l'estat són lliure, ocupat o esborrat.

El cost de les còpies i del destructor és $\Theta(M)$. No cal definir aquestes operacions perquè les implícites funcionen bé.

```
#include "ada.hh"

template <typename Clau, typename Info>
class Diccionari {
```

```
private:

    enum Estat {LLI,OCU,ESB};
    struct Pos {
        Clau clau;
        Info info;
        Estat estat;
    };

    int n;                  // Nombre d'elements
    int M;                  // Nombre de posicions
    vector<Pos> t;          // Taula de dispersió
```

public:

. . . .

Constructor. Crea un diccionari buit en una taula de dispersió amb `M` posicions.
Cost: $\Theta(M)$.

. . . .

```
Diccionari (int M = 1009)
:   n(0), M(M), t(M) {
    for (int i=0; i<M; i++) {
        t[i].estat = LLI;
}   }
```

. . . .

Insereix la clau `clau` dins del diccionari amb informació `info`.
Si la clau ja hi és, canvia la informació associada.
Llança una excepció d'implementació si la taula és plena.
Cost en el cas pitjor: $\Theta(n)$.
Cost en el cas mitjà sense esborrats quan la taula no és "gaire" plena: $\Theta(1)$.

. . . .

```
void assignar (Clau& clau, Info& info) {
    int pos;
    if (hi_es(clau,pos)) {
        t[pos].info = info;
    } else {
        if (n == M) throw ErrorImpl("Taula plena");
        t[pos].clau = clau;
        t[pos].info = info;
        t[pos].estat = OCU;
        ++n;
}   }
```

. . . .

Treu la clau `clau` del diccionari amb la seva informació associada. Si la clau no hi és, el diccionari no canvia.
Cost en el cas pitjor: $\Theta(n)$.
Cost en el cas mitjà sense esborrats quan la taula no és "gaire" plena: $\Theta(1)$.

. . . .

```
void treure (Clau& clau) {
    int pos;
    if (hi_es(clau,pos)) {
        t[pos].estat = ESB;
        --n;
}   }
```

. . . .

Retorna una referència a la informació associada a `clau` dins del diccionari.

Llença una excepció de precondició si la clau no era present.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà sense esborrats: $\Theta(1)$.

. . . .

```
Info& consulta (Clau& clau) {
    int pos;
    if (not hi_es(clau,pos))
        throw ErrorPrec("no hi és");
    return t[pos].info;
}
```

. . . .

Indica si la clau és o no present dins del diccionari.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà sense esborrats: $\Theta(1)$.

. . . .

```
bool present (Clau& clau) {
    int pos;
    return hi_es(clau,pos);
}
```

. . . .

Retorna la talla del diccionari.

Cost: $\Theta(1)$.

. . . .

```
    int talla () {
        return n;
    }

private:
```

. . . .

Cerca si una `clau` es troba en el diccionari i indica on és o on caldria posar-la.

Cost en el cas pitjor: $\Theta(n)$.

Cost en el cas mitjà sense esborrats: $\Theta(1)$.

. . . .

```
    bool hi_es (Clau& clau, int& pos) {
        pos = -1;
        int p = hash(clau)%M;
        for (int c=0; c<M; ++c) {
            if (t[p].estat==OCU) {
                if (t[p].clau==clau) {
                    pos = p;
                    return true;
                }
            } else {
                if (pos==-1) pos = p;
                if (t[p].estat==LLI) return false;
            }
            if (++p > M) p = 0;
        }
        return false;
    }
};
```

Arbres binaris de cerca

Diccionari amb arbre binari de cerca (ABC).

Se suposa que les claus són comparables.

Un ABC és un arbre binari on els nodes contenen parells clau/informació i es compleix la propietat que per a tot node u, la clau enmagatzemada en u és més gran que qualsevol clau enmagatzemada en el subarbre esquerre de u i més petita que qualsevol clau enmagatzemada en el subarbre dret de u.

Un ABC es representa mitjançant un punter a un `Node` que conté: una clau, la informació associada, un punter al subarbre esquerre i un punter al subarbre dret. Els arbres buits es codifiquen amb un punter nul. Cada diccionari té un camp `arrel` que és un punter a l'arrel de l'ABC que té les dades. També es guarda un comptador `n` pel nb d'elements enmagatzemats.

En aquesta classe, n denota la talla d'un diccionari.

El cost de la majoria de les operacions és $\Theta(n)$ en cas pitjor, ja que una entrada ordenada pot convertir l'arbre en una llista. Per a un diccionari semi-estàtic (sense esborrat) i insercions aleatòries, el cost mig de les operacions de cerca i inserció és $\Theta(\log n)$. El cost espacial és sempre $\Theta(n)$.

```
#include "ada.hh"

template <typename Clau, typename Info>
class Diccionari {
```

private:

. . . .

Un `Node` enmagatzema una clau, la seva informació associada i dos punters a dos altres `Node`. La següent propietat es manté: Totes les claus a `fesq` són menors que `clau`; totes les claus a `fdre` són majors que `clau`.

. . . .

```
struct Node {
    Clau clau;
    Info info;
    Node* fesq;      // Punter al fill esquerre
    Node* fdre;      // Punter al fill dret

    Node (Clau& c, Info& i, Node* fe, Node* fd)
    :   clau(c), info(i), fesq(fe), fdre(fd) { }
};

int n;              // Nombre d'elements en l'ABC
Node* arrel;        // Punter a l'arrel de l'ABC
```

public:

. . . .

Constructor. Crea un diccionari buit.

Cost: $\Theta(1)$.

. . . .

```
Diccionari () {
    n = 0;
    arrel = null;
}
```

. . . .

Constructor de còpia.

Cost en el cas pitjor: $\Theta(n)$.

. . . .

```
Diccionari (Diccionari& d) {
    n = d.n;
    arrel = copia(d.arrel);
}
```

. . . .

Destructor.

Cost en el cas pitjor: $\Theta(n)$.

. . . .

```
~Diccionari () {
    alliberar(arrel);
}
```

. . . .

Operador d'assignació.

Cost en el cas pitjor: $\Theta(n + \texttt{d}.n)$.

. . . .

```
Diccionari& operator= (Diccionari& d) {
    if (&d!=this) {
        alliberar(arrel);
        n = d.n;
        arrel = copia(d.arrel);
    }
    return *this;
}
```

....

Assigna a `clau` el valor `info`.

Cost en el cas pitjor: $\Theta(n)$.

....

```
void assignar (Clau& clau, Info& info) {
    assignar(arrel,clau,info);
}
```

....

Treu la clau clau del diccionari amb la seva informació associada.

Si la clau no hi és, el diccionari no canvia.

Cost en el cas pitjor: $\Theta(n)$.

....

```
void treure (Clau& clau) {
    esborrar_3(arrel,clau);
}
```

. . . .

Retorna una referència a la informació associada a `clau` dins del diccionari.

Llança una excepció de precondició si la clau no era present.

Cost en el cas pitjor: $\Theta(n)$.

. . . .

```
Info& consulta (Clau& clau) {
    if (Node* p=cerca(arrel,clau)) {
        return p->info;
    } else {
        throw ErrorPrec("La clau no era present");
}   }
```

. . . .

Indica si la clau és o no present dins del diccionari.

Cost en el cas pitjor: $\Theta(n)$.

. . . .

```
bool present (Clau& clau) {
    return cerca(arrel,clau)!=null;
}
```

. . . .

Retorna la talla del diccionari.

Cost: $\Theta(1)$.

. . . .

```
int talla () {
    return n;
}
```

private:

. . . .

Elimina l'arbre apuntat per `p`.

Cost en el cas pitjor: $\Theta(s)$ on s és el nb de nodes a l'arbre arrelat en `p`.

. . . .

```
static void alliberar (Node* p) {
    if (p) {
        alliberar(p->fesq);
        alliberar(p->fdre);
        delete p;
}   }
```

. . . .

Retorna un punter a una còpia de l'arbre apuntat per `p`.

Cost en el cas pitjor: $\Theta(s)$ on s és el nb de nodes a l'arbre arrelat en `p`.

. . . .

```
static Node* copia (Node* p) {
    return p ?
        new
Node(p->clau,p->info,copia(p->fesq),copia(p->fdre))
        : null;
}
```

. . . .

Retorna un punter al node de l'arbre apuntat per `p` que contingui la clau donada, o `null` si la clau no hi és.
Cost en el cas pitjor: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
static Node* cerca (Node* p, Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            return cerca(p->fesq,clau);
        } else if (clau > p->clau) {
            return cerca(p->fdre,clau);
    }   }
    return p;
}
```

. . . .

Assigna la informació `info` a la clau `clau` si aquesta es troba en el subarbre arrelat en `p`. Altrament afegeix un nou node a `p` amb `clau` i `info`.

Cost en el cas pitjor: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
void assignar (Node*& p, Clau& clau, Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq,clau,info);
        } else if (clau > p->clau) {
            assignar(p->fdre,clau,info);
        } else {
            p->info = info;
        }
    } else {
        p = new Node(clau,info,0,0);
        ++n;
}   }
```

. . . .

Retorna un punter al node que conté el valor mínim en `p`. Se suposa que `p` no és nul.
Cost en el cas pitjor: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
static Node* minim (Node* p) {
    return p->fesq ? minim(p->fesq) : p;
}
```

. . . .

Retorna un punter al node que conté el valor màxim en `p`. Se suposa que `p` no és nul.
Cost en el cas pitjor: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
static Node* maxim (Node* p) {
    while (p->fdre) p = p->fdre;
    return p;
}
```

. . . .

Esborrat versió 1: Esborrar un node amb algun fill buit és fàcil. Quan els dos subarbres no són buits, es penja el fill esquerre com a nou fill esquerre del mínim del fill dret.

Inconvenient: els arbres es degraden molt aviat.

Cost en el cas pitjor: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en p.

. . . .

```
void esborrar_1 (Node*& p, Clau& clau) {
    if (p) {
        if (clau<p->clau) {
            esborrar_1(p->fesq,clau);
        } else if (clau>p->clau) {
            esborrar_1(p->fdre,clau);
        } else {
            Node* q = p;
            if (!p->fesq)       p = p->fdre;
            else if (!p->fdre)  p = p->fesq;
            else {
                Node* m = minim(p->fdre);
                m->fesq = p->fesq;
                p = p->fdre;
            }
            delete q;  --n;
}   }   }
```

. . . .

Esborrat versió 2: Esborrar un node amb algun fill buit és fàcil. Quan els dos subarbres no són buits, es copia el mínim del seu fill dret al node que s'esborra i es demana l'esborrat del node mínim, que segur que no té fill esquerre. Inconvenient: es copien claus i informacions, no punters.
Cost: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
void esborrar_2 (Node*& p, Clau& clau) {
    if (p) {
        if (clau<p->clau) {
            esborrar_2(p->fesq,clau);
        } else if (clau>p->clau) {
            esborrar_2(p->fdre,clau);
        } else if (!p->fesq) {
            Node* q = p;  p = p->fdre;
            delete q;  --n;
        } else if (!p->fdre) {
            Node* q = p;  p = p->fesq;
            delete q;  --n;
        } else {
            Node* m = minim(p->fdre);
            p->clau = m->clau; p->info = m->info;
            esborrar_2(p->fdre,m->clau);
}   }   }
```

. . . .

Esborrat versió 3: Esborrar un node amb algun fill buit és fàcil. Quan els dos subarbres no són buits, s'utilitza `esborrar_minim`, que esborra el mínim del fill dret, retornant-ne un punter. Aquest node es converteix en la nova arrel.
Cost: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
void esborrar_3 (Node*& p, Clau& clau) {
    if (p) {
        if (clau<p->clau) {
            esborrar_3(p->fesq,clau);
        } else if (clau>p->clau) {
            esborrar_3(p->fdre,clau);
        } else {
            Node* q = p;
            if (!p->fesq)      p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {
                Node* m = esborrar_minim(p->fdre);
                m->fesq = p->fesq; m->fdre = p->fdre;
                p = m;
            }
            delete q;  --n;
}   }   }
```

. . . .

Esborra de l'arbre el node que conté l'element mínim de `p`. El node no és eliminat, sino retornat.

Cost: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
    Node* esborrar_minim (Node*& p) {
        if (p->fesq) {
            return esborrar_minim(p->fesq);
        } else {
            Node* q = p;
            p = p->fdre;
            return q;
    }   }

};
```

Arbres AVL

Diccionari amb arbre de Adel′son-Vel′skiĭ i Landis (AVL).

Aquesta implementació conté molt poca documentació. Consulteu qualsevol llibre de text o els apunts de classe.

L'alçada de `null` és -1 i l'alçada d'una fulla és 0.

```
#include "ada.hh"

template <typename Clau, typename Info>
class Diccionari {

private:

    struct Node {
        Clau clau;
        Info info;
        Node* fesq;       // Punter al fill esquerre
        Node* fdre;       // Punter al fill dret
        int alc;          // Alçada de l'arbre

        Node (Clau& c, Info& i, Node* fe, Node* fd, int a)
        :   clau(c), info(i), fesq(fe), fdre(fd), alc(a)
        { }
    };
```

```
    int n;              // Nombre d'elements en l'AVL
    Node* arrel;        // Punter a l'arrel de l'AVL

public:
```

....

Constructor. Crea un diccionari buit.

Cost: $\Theta(1)$.

....

```
    Diccionari () {
        n = 0;
        arrel = null;
    }
```

....

Constructor de còpia.

Cost: $\Theta(n)$.

....

```
    Diccionari (Diccionari& d) {
        n = d.n;
        arrel = copia(d.arrel);
    }
```

. . . .

Destructor.

Cost: $\Theta(n)$.

. . . .

```
~Diccionari () {
    alliberar(arrel);
}
```

. . . .

Operador d'assignació.

Cost: $\Theta(n + \texttt{d}.n)$.

. . . .

```
Diccionari& operator= (Diccionari& d) {
    if (&d!=this) {
        alliberar(arrel);
        n = d.n;
        arrel = copia(d.arrel);
    }
    return *this;
}
```

. . . .

Assigna a `clau` el valor `info`.

Cost: $\Theta(\log n)$.

. . . .

```
void assignar (Clau& clau, Info& info) {
    assignar(arrel,clau,info);
}
```

. . . .

Treu la clau clau del diccionari amb la seva informació associada. Si la clau no hi és, el diccionari no canvia.

Cost: $\Theta(\log n)$.

. . . .

```
void treure (Clau& clau) {
    esborrar(arrel,clau);
}
```

. . . .

Retorna una referència a la informació associada a `clau` dins del diccionari.

Llança una excepció de precondició si la clau no és present.

Cost: $\Theta(\log n)$.

. . . .

```
Info& consulta (Clau& clau) {
    if (Node* p=cerca(arrel,clau)) {
        return p->info;
    } else {
        throw ErrorPrec("La clau no és present");
}   }
```

. . . .

Indica si la clau és o no present dins del diccionari.

Cost: $\Theta(\log n)$.

. . . .

```
bool present (Clau& clau) {
    return cerca(arrel,clau)!=null;
}
```

. . . .

Retorna la talla del diccionari.

Cost: $\Theta(1)$.

. . . .

```
int talla () {
    return n;
}
```

```
private:
```

. . . .

Elimina l'arbre apuntat per `p`.

Cost: $\Theta(n)$ on n és el nombre de nodes a l'arbre arrelat en `p`.

. . . .

```
static void alliberar (Node* p) {
    if (p) {
        alliberar(p->fesq);
        alliberar(p->fdre);
        delete p;
}   }
```

. . . .

Retorna un punter a una còpia de l'arbre apuntat per `p`.

Cost: $\Theta(s)$ on s és el nombre de nodes a l'arbre arrelat en `p`.

. . . .

```
static Node* copia (Node* p) {
    return
        p ? new
Node(p->clau,p->info,copia(p->fesq),copia(p->fdre))
          : null;
}
```

. . . .

Retorna un punter al node de l'arbre apuntat per `p` que contingui la clau donada, o `null` si la clau no hi és.
Cost: $\Theta(h)$ on h és l'alçada de l'arbre arrelat en `p`.

. . . .

```
static Node* cerca (Node* p, Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            return cerca(p->fesq,clau);
        } else if (clau > p->clau) {
            return cerca(p->fdre,clau);
    }   }
    return p;
}
```

. . . .

Retorna l'alçada d'un node `p` (que pot ser buit).

Cost: $\Theta(1)$.

. . . .

```
static int alcada (Node* p) {
    return p ? p->alc : -1;
}
```

. . . .

Actualitza l'alçada d'un node no buit assumint que l'alçada dels deus fills és correcta.

Cost: $\Theta(1)$.

. . . .

```
static void actualitzar_alcada (Node* p) {
    p->alc = 1+max(alcada(p->fesq),alcada(p->fdre));
}
```

. . . .

Realitza una rotació simple esquerra-esquerra (LL) sobre un node `p` no buit amb fill esquerre no buit. Cost: $\Theta(1)$.

. . . .

```
static void LL (Node*& p) {
    Node* q = p;
    p = p->fesq;
    q->fesq = p->fdre;
    p->fdre = q;
    actualitzar_alcada(q);
    actualitzar_alcada(p);
}
```

. . . .

Realitza una rotació simple dreta-dreta (RR) sobre un node `p` no buit amb fill dret no buit. Cost: $\Theta(1)$.

. . . .

```
static void RR (Node*& p) {
    Node* q = p;
    p = p->fdre;
    q->fdre = p->fesq;
    p->fesq = q;
    actualitzar_alcada(q);
    actualitzar_alcada(p);
}
```

. . . .

Realitza una rotació doble esquerra-dreta (LR) sobre un node `p` no buit amb fill esquerre no buit que també té un fill dret no buit.
Cost: $\Theta(1)$.

. . . .

```
static void LR (Node*& p) {
    RR(p->fesq);
    LL(p);
}
```

. . . .

Realitza una rotació doble dreta-esquerra (RL) sobre un node `p` no buit amb fill dret no buit que també té un fill esquerre no buit.
Cost: $\Theta(1)$.

. . . .

```
static void RL (Node*& p) {
    LL(p->fdre);
    RR(p);
}
```

. . . .

Mètode privat per assignar en un AVL.

. . . .

```
void assignar (Node*& p, Clau& clau, Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq,clau,info);
            if (alcada(p->fesq)-alcada(p->fdre)==2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p);
            }
            actualitzar_alcada(p);
        } else if (clau > p->clau) {
            assignar(p->fdre,clau,info);
            if (alcada(p->fdre)-alcada(p->fesq)==2) {
                if (clau > p->fdre->clau) RR(p);
                else RL(p);
            }
            actualitzar_alcada(p);
        } else {
            p->info = info;
        }
    } else {
        p = new Node(clau,info,0,0,0);  ++n;
}   }
```

. . . .

Funció d'esborrat en un AVL.

. . . .

```
void esborrar (Node*& p, Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar(p->fesq,clau);
            balanceja_esq(p);
        } else if (clau > p->clau) {
            esborrar(p->fdre,clau);
            balanceja_dre(p);
        } else {
            Node* vell = p;
            if (p->alc==0) {
                p = null;
            } else if (!p->fesq) {
                p = p->fdre;
            } else if (!p->fdre) {
                p = p->fesq;
            } else {
                Node* q = esborra_minim(p->fdre);
                q->fesq = p->fesq;  q->fdre = p->fdre;
                p = q;
                balanceja_dre(p);
            }
            delete vell;  --n;
}   }   }
```

```
void balanceja_esq (Node*& p) {
    if (alcada(p->fdre)-alcada(p->fesq)==2) {
        if (alcada(p->fdre->fesq) -
            alcada(p->fdre->fdre) == 1)
        {
            RL(p);
        } else {
            RR(p);
        }
    } else {
        actualitzar_alcada(p);
}   }

void balanceja_dre (Node*& p) {
    if (alcada(p->fesq)-alcada(p->fdre)==2) {
        if (alcada(p->fesq->fdre) -
            alcada(p->fesq->fesq) == 1)
        {
            LR(p);
        } else {
            LL(p);
        }
    } else {
        actualitzar_alcada(p);
}   }
```

```
    Node* esborra_minim (Node*& p) {
        if (p->fesq) {
            Node* q = esborra_minim(p->fesq);
            balanceja_esq(p);
            return q;
        } else {
            Node* q = p;
            p = p->fdre;
            return q;
    }   }

};
```

Anàlisi i Disseny d'Algorismes

Cues de prioritat

Cues de prioritat — versió recursiva

Cua de prioritats. Implementació recursiva.

La cua de prioritats es manté amb un heap en una taula dinàmica Un heap és un arbre binari complert amb la propietat de que la clau enmagatzemada a cada node és menor o igual a les claus enmagatzemades als nodes dels seus subarbres. Un heap per n elements es pot implementar facilment en les posicions 1 a n. En aquest cas, per a un element a la posició i amb $1 \le i \le n$, el seu fill esquerre es troba a la posició $2i$ si $2i \le n$, el seu fill dret es troba a la posició $2i + 1$ si $2i + 1 \le n$, i el seu pare a la posició $\lceil i/2 \rceil$ si $i > 1$.

Per facilitat, la taula inclou sempre una posició 0 que no es fa servir mai.

En aquest classe, n denota la talla de la cua de prioritat. Se suposa que afegir al final d'un vector té cost constant (no és del tot correcte: és constant amortitzat).

```
#include "ada.hh"

template <typename Elem>
class CuaPrio {

private:

    vector<Elem> t;      // Taula on es forma el heap
                         // (la posició 0 no s'utilitza)
```

public:

Constructor. Crea una cua de prioritats buida.

Cost: $\Theta(1)$.

```
CuaPrio () {
    t.push_back(Elem());
}
```

Afegeix un nou element x.

Cost en el cas pitjor: $\Theta(\log n)$.

```
void afegir (Elem& x) {
    t.push_back(x);
    surar(talla());
}
```

. . . .

Treu i retorna l'element mínim.

Cost en el cas pitjor: $\Theta(\log n)$.

. . . .

```
Elem treure_min () {
    if (buida()) throw ErrorPrec("CuaPrio buida");
    Elem x = t[1];
    t[1] = t.back();
    t.pop_back();
    enfonsar(1);
    return x;
}
```

. . . .

Retorna un element amb prioritat mínima.

Cost: $\Theta(1)$.

. . . .

```
Elem minim () {
    if (buida()) throw ErrorPrec("CuaPrio buida");
    return t[1];
}
```

. . . .

Retorna la talla de la cua de prioritat.

Cost: $\Theta(1)$.

. . . .

```
int talla () {
    return t.size()-1;
}
```

. . . .

Indica si la cua de prioritats és buida.

Cost: $\Theta(1)$.

. . . .

```
bool buida () {
    return t.talla()==0;
}
```

```
private:

    ..................................................................
    Operació de surar.
    ..................................................................
    void surar (int i) {
        if (i!=1 and t[i/2]>t[i]) {
            canviar(t[i],t[i/2]);
            surar(i/2);
    }   }


    ..................................................................
    Operació d'enfonsar.
    ..................................................................
    void enfonsar (int i) {
        int n = talla();
        int c = 2*i;
        if (c<=n) {
            if (c+1<=n and t[c+1]<t[c]) c++;
            if (t[i]>t[c]) {
                canviar(t[i],t[c]);
                enfonsar(c);
    }   }   }

};
```

Cues de prioritat — versió iterativa

Cua de prioritats. Implementació iterativa.

Igual que la versió anterior però amb implementació iterativa i optimitzada per les operacions de surar i enfonsar.

```
#include "ada.hh"

template <typename Elem>
class CuaDePrio {

private:

    vector<Elem> t;         // Taula on es forma el heap
                            // (la posició 0 no s'utilitza)

public:
```

Constructor. Crea una cua de prioritats buida.

Cost: $\Theta(1)$.

```
    CuaDePrio () {
        t.push_back(Elem());
    }
```

Afegeix un nou element x.

Cost en el cas pitjor: $\Theta(\log n)$.

```
void afegir (Elem& x) {
    t.push_back(x);    // Fico x per ficar-hi alguna cosa
    int i = talla();
    while (i!=1 and t[i/2]>x) {
        t[i] = t[i/2];
        i = i/2;
    }
    t[i] = x;
}
```

. . . .

Treu i retorna l'element mínim.

Llança una excepció de precondició si la cua de prioritats és buida.

Cost en el cas pitjor: $\Theta(\log n)$.

. . . .

```
Elem treure_min () {
    if (buida()) throw ErrorPrec("CuaDePrio buida");
    int n = talla();
    Elem e = t[1],  x = t[n];
    t.pop_back();  --n;
    int i = 1,  c = 2*i;
    while (c<=n) {
        if (c+1<=n and t[c+1]<t[c]) ++c;
        if (x<=t[c]) break;
        t[i] = t[c];
        i = c;
        c = 2*i;
    }
    t[i] = x;
    return e;
}
```

. . . .

Retorna un element amb prioritat mínima.

Llança una excepció de precondició si la cua és buida.

Cost: $\Theta(1)$.

. . . .

```
Elem minim () {
    if (buida()) throw ErrorPrec("CuaDePrio buida");
    return t[1];
}
```

. . . .

Retorna la talla de la cua de prioritat.

Cost: $\Theta(1)$.

. . . .

```
int talla () {
    return t.size()-1;
}
```

. . . .

Indica si la cua de prioritats és buida.

Cost: $\Theta(1)$.

. . . .

```
bool buida () {
    return talla()==0;
}

};
```

Anàlisi i Disseny d'Algorismes

Grafs i particions

Definició de tipus per a grafs

Definició de tipus per a grafs.

Representem els grafs amb una taula de llistes d'adjacència, fent que els vèrtexs siguin enters entre 0 i $|V|-1$. Els grafs són dirigits, però es poden usar com a grafs no dirigits fent que tots els arcs siguin dobles.

Definim dues macros: `forall_ver(u,G)` fa que l'enter `u` recorri tots els vèrtexs del graph `G` i `forall_adj(uv,G[u])` fa que l'iterador `uv` recorri tota la llista d'adjacència `G[u]`. En aquest darrer cas, hom es pot imaginar que `u` és el vèrtex de sortida, que `uv` és l'arc i que `*uv` és el vèrtex de destí.

```
#ifndef graf_hh
#define graf_hh

#include <vector>
#include <list>

using namespace std;

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)

#endif
```

Recorregut en profunditat

Recorregut en profunditat.

La funció retorna una llista dels vèrtexs segons el seu ordre de visita en un recorregut en profunditat.

```
#include <stack>
#include "ada.hh"
#include "graph.hh"
```

Versió recursiva

```
void dfs_rec (graph& G, int u,
        vector<boolean>& vis, list<int>& L) {
    if (not vis[u]) {
        vis[u] = true;  L.push_back(u);
        forall_adj(uv,G[u]) {
            dfs_rec(G,*uv,vis,L);
}   }   }

list<int> dfs_rec (graph& G) {
    list<int> L;
    vector<boolean> vis(G.size(),false);
    forall_ver(u,G) {
        dfs_rec(G,u,vis,L);
    }
    return L;
}
```

Versió iterativa

```
list<int> dfs_ite (graph& G) {
    list<int> L;
    stack<int> S;
    vector<bool> vis(G.size(),false);

    forall_ver(u,G) {
        S.push(u);
        while (not S.empty()) {
            int v = S.top();  S.pop();
            if (not vis[v]) {
                vis[v] = true;  L.push_back(v);
                forall_adj(vw,G[v]) {
                    S.push(*vw);
    }   }   }   }
    return L;
}
```

Recorregut en amplada

Recorregut en amplada.

La funció retorna una llista dels vèrtexs segons el seu ordre de visita en un recorregut en amplada.

```
#include <queue>
#include "ada.hh"
#include "graph.hh"

list<int> bfs (graph& G) {
    list<int> L;
    queue<int> Q;
    vector<bool> enc(G.size(),false);

    forall_ver(u,G) if (not enc[u]) {
        Q.push(u);  enc[u] = true;
        while (not Q.empty()) {
            int v = Q.front(); Q.pop();
            L.push_back(v);
            forall_adj(vw,G[v]) {
                int w = *vw;
                if (not enc[w]) {
                    Q.push(w); enc[w] = true;
    }   }   }   }
    return L;
}
```

Ordenació topològica

Ordenació topològica.

Donat un graf dirigit acíclic, retorna una ordenació topològica dels seus vèrtexs, és a dir, una ordenació on v no apareix mai abans de u si hi ha un camí de u a v.

```
#include <stack>
#include "ada.hh"
#include "graph.hh"

list<int> ordenacio_topologica (graph& G) {
    vector<int> ge(G.size(),0);
    forall_ver(u,G) forall_adj(uv,G[u]) {
        ++ge[*uv];
    }

    stack<int> S;
    forall_ver(u,G) if (ge[u]==0) S.push(u);

    list<int> L;
    while (not S.empty()) {
        int u = S.top();  S.pop();
        L.push_back(u);
        forall_adj(uv,G[u]) {
            int v = *uv;
            if (--ge[v] == 0) {
                S.push(v);
    }   }   }
    return L;
}
```

Particions — versió 1

Implementació del TAD Partició **amb MF-sets.**

Cada classe d'equivalència es representa per un arbre. Així, la partició en classes d'equivalència es representa per un bosc. Cada arbre conté un node per cadascun dels elements dins de la seva classe. El representant canònic de qualsevol classe es defineix com l'element enmagatzemat a l'arrel del seu arbre.

La implementació considera particions del conjunt dels enters entre 0 i $\texttt{n}-1$ i manté una taula `t` d'enters. Els arbres s'organitzen a través de punters als pares. Per a cada element $0 \leq i < \texttt{n}$, es guarda en $\texttt{t}[i]$ la posició del seu pare, excepte per les arrels, per les quals es guarda un valor especial -1.

En aquest classe, n denota la talla de la partició. Se suposa que l'usuari mai passa paràmetres incorrectes.

```
#include "ada.hh"

class Particio {

private:

    vector<int> t;    // Taula per als enllaços als pares
```

```
public:
```

. . . .

Constructor. Cost: $\Theta(n)$.

. . . .

```
    explicit Particio (int n) : t(n,-1) { }
```

. . . .

Uneix les classes que contenen `x` i `y` en una sola classe.

Cost en el cas pitjor: $\Theta(n)$.

. . . .

```
    void unir (int x, int y) {
        int rx = representant(x);
        int ry = representant(y);
        if (rx!=ry) t[ry] = rx;
    }
```

. . . .

Retorna un representant canònic de la classe de `x`.

Cost en cas pitjor: $\Theta(n)$.

. . . .

```
    int representant (int x) {
        return t[x]==-1 ? x : representant(t[x]);
    }

};
```

Particions — versió 2

Implementació del TAD Partició **amb MF-sets i unió per talla.**

En aquesta implementació es manté, per a cada arbre, la seva talla. Aquest valor es guarda a l'arrel, canviat de signe (per poder-lo distingir dels enllaços als pares). D'aquesta forma es pot garantir que els arbres tenen alçada $\log n$ per a particions de n elements.

```
#include "ada.hh"

class Particio {

private:

    vector<int> t;

public:
```

Constructor.

Cost: $\Theta(n)$.

```
    explicit Particio (int n) : t(n,-1) { }
```

. . . .

Retorna el representant canònic de la classe de `x`.

Cost en cas pitjor: $\Theta(\log n)$.

. . . .

```
int representant (int x) {
    return t[x]<0 ? x : representant(t[x]);
}
```

. . . .

Uneix les classes que conten `x` i `y` en una sola classe.

Cost en cas pitjor: $\Theta(\log n)$.

. . . .

```
void unir (int x, int y) {
    int rx = representant(x);
    int ry = representant(y);
    if (rx!=ry) {
        if (t[ry]>=t[rx]) {
            t[rx] += t[ry];
            t[ry] = rx;
        } else {
            t[ry] += t[rx];
            t[rx] = ry;
}   }   }

};
```

Particions — versió 3

Implementació del TAD Partició **amb MF-sets, unió per talla i compressió de camins.**

Per accelerar la implementació precedent, un cop s'ha trobat l'arrel d'un node qualsevol, aquest es pot arrelar directament sota aquesta arrel, produint d'aquesta forma un estalvi de temps en la propera vegada que es demani l'arrel d'aquest node.

Amb aquest implementació, el cost de realitzar m operacions de `unio` i n operacions de `representant` és $O(m \log^* n)$.

```
#include "ada.hh"

class Particio {

    vector<int> t;

public:
```

....

Constructor.

Cost: $\Theta(n)$.

....

```
    explicit Particio (int n) : t(n,-1) { }
```

. . . .

Retorna el representant canònic de la classe de `x`.

Cost en el cas pitjor: $\Theta(\log n)$.

. . . .

```
int representant (int x) {
    return t[x]<0 ? x : t[x] = representant(t[x]);
}
```

. . . .

Uneix les classes que conten `x` i `y` en una sola classe.

Cost en el cas pitjor: $\Theta(\log n)$.

. . . .

```
void unir (int x, int y) {
    int rx = representant(x);
    int ry = representant(y);
    if (rx!=ry) {
        if (t[ry]>=t[rx]) {
            t[rx] += t[ry];
            t[ry] = rx;
        } else {
            t[ry] += t[rx];
            t[rx] = ry;
}   }   }
```

. . . .

Embelliment de `representant`

. . . .

```
int operator[] (int x) {
    return representant(x);
}

};
```

Anàlisi i Disseny d'Algorismes

Algorismes voraços

Definició de tipus per a grafs amb pesos

Definició de tipus per a grafs amb pesos reals.

```
#ifndef wgraf_hh
#define wgraf_hh

#include <vector>
#include <list>

using namespace std;

struct aresta {
    int u;          // vèrtex d'orígen
    int v;          // vèrtex de destí
    double p;       // pès de l'arc

    aresta (int u, int v, double p) : u(u), v(v), p(p) { }

    bool operator< (const aresta& b) const {
        return p > b.p;     // al revés perquè les
priority_queue són pel màxim
                            // i ens els algorismes ens
interessa el mínim.
    }

};

typedef vector< list<aresta> > wgraph;
typedef list<aresta>::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)


#endif
```

Arbres d'expansió mínims: Algorisme de Prim

Calcula un arbre d'expansió mínim d'una graf no dirigit connex amb pesos utilitzant l'algorisme de Prim.

```
class Prim {

private:

    list<aresta> L;

public:

    Prim (wgraph& G) {
        int n = G.size();
        priority_queue<aresta> CP;
        vector<boolean> S(n,false);
        forall_adj(uv,G[0]) CP.push(*uv);
        S[0] = true;
        int c = 1;
        while (c!=n) {
            aresta uv = CP.top();  CP.pop();
            if (not S[uv.v]) {
                L.push_back(uv);
                forall_adj(vw,G[uv.v]) {
                    if (not S[vw->v]) {
                        CP.push(*vw);
                }   }
                S[uv.v] = true;
                ++c;
    }   }   }

    list<aresta> arbre () {
        return L;
    }
};
```

Arbres d'expansió mínims: Algorisme de Kruskal

Calcula un arbre d'expansió mínim d'una graf no dirigit connex amb pesos utilitzant l'algorisme de Kruskal.

```
class Kruskal {

private:

    list<aresta> L;

public:

    Kruskal (wgraph& G) {
        int n = G.size();
        Particio P(n);
        priority_queue<aresta> CP;
        forall_ver(u,G) {
            forall_adj(uv,G[u]) {
                CP.push(*uv);
        }   }
        while (not CP.empty()) {
            aresta a = CP.top();  CP.pop();
            if (P[a.u]!=P[a.v]) {
                L.push_back(a);
                P.unir(a.u,a.v);
    }   }   }

    list<aresta> arbre () {
        return L;
    }
};
```

Camins mínims: Algorisme de Dijkstra

Calcula els camins mínims en un graf dirigit `G` des del vèrtex `u`. utilitzant l'algorisme de Dijkstra.

```
class Dijkstra {

private:

    vector<int> p;
    vector<double> d;

public:

    Dijkstra (wgraph& G, int u) {

        // inicialització
        int n = G.size();
        vector<boolean> S(n,false);
        p = vector<int>(n,-1);
        d = vector<double>(n,infinit);
        d[u] = 0;
        priority_queue<aresta> CP;
        CP.push(aresta(-1,u,0));
```

```
        // bucle principal
        while (not CP.empty()) {
            aresta a = CP.top();  CP.pop();
            int v = a.v;
            if (not S[v]) {
                S[v] = true;
                forall_adj(vw,G[v]) {
                    int w = vw->v;
                    if (not S[w]) {
                        if (d[w] > d[v]+vw->p) {
                            d[w] = d[v]+vw->p;
                            p[w] = v;
                            CP.push(aresta(v,w,d[w]));
    }   }   }   }   }   }

    double distancia (int v) {
        return d[v];
    }

    vector<double> distancies () {
        return d;
    }

    int pare (int v) {
        return p[v];
    }

    vector<int> pares () {
        return p;
    }
};
```

Codis de Huffman

Creació d'un arbre de Huffman.

A partir d'una taula de freqüències que assigna a cada caràcter la seva freqüència es construeix un arbre de Huffman. S'ofereixen mètodes per codificar i decodificar utilitzant aquest arbre.

```
#include "ada.hh"

class Huffman {

private:

    struct node {
        node *fe, *fd, *pare;
        char c;
        double f;

        node (node* fe, node* fd, char c, double f) :
            fe(fe), fd(fd), pare(null), c(c), f(f) {
                if (fe) fe->pare = this;
                if (fd) fd->pare = this;
            }

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

    node* arrel;
    map<char,node*> fulles;
```

```
    struct comparador {
        bool operator() (node* p, node* q) {
            return p->f > q->f;
        }
    };

public:

    ~Huffman () {
        delete arrel;
    }

    Huffman (map<char,double>& F) {
        priority_queue<node*,vector<node*>,comparador> CP;
        foreach(it,F) {
            node* p = new node(null,null,
                                it->first,it->second);
            CP.push(p);
            fulles[it->first] = p;
        }
        while (CP.size()!=1) {
            node* p = CP.top();  CP.pop();
            node* q = CP.top();  CP.pop();
            CP.push(new node(p,q,' ',p->f+q->f));
        }
        arrel = CP.top();
    }
```

```
    string decodifica (string s) {
        string r;
        node* p = arrel;
        unsigned i = 0;
        while (i<=s.size()) {
            if (p->c) {
                r += p->c;
                p = arrel;
            } else {
                p = s[i++]=='0' ? p->fe : p->fd;
        }   }
        return r;
    }

    string codifica (string s) {
        string r;
        for (unsigned i=0; i<s.size(); ++i)
            r += codifica(fulles[s[i]]);
        return r;
    }

private:

    string codifica (node* p) {
        if (p->pare==null) {
            return "";
        } else if (p->pare->fe == p) {
            return codifica(p->pare) + '0';
        } else {
            return codifica(p->pare) + '1';
    }   }
};
```

```
// Exemple d'ús

int main () {
    map<char,double> F;

    F['A'] = 0.35;
    F['B'] = 0.1;
    F['C'] = 0.2;
    F['D'] = 0.2;
    F['E'] = 0.15;

    Huffman h(F);
    string s = h.codifica("ABAC");
    cout << s << endl;
    cout << h.decodifica(s) << endl;
}
```

Anàlisi i Disseny d'Algorismes

Algorismes de tornada enrera

Les n reines — versió 1

Problema de les n-reines.

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

Aquesta és una primera solució utilitzant tornada enrera sense marcatges.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class NReines {

    int n;                        // nombre de reines
    vector<int> T;                // configuració actual

    void recursiu (int i) {
        if (i==n) {
            escriure();
        } else {
            for (int j=0; j<n; ++j) {
                T[i] = j;
                if (legal(i)) {
                    recursiu(i+1);
    }   }   }   }
```

```
    bool legal (int i) {
    // Indica si la configuració amb les dames 0..i és legal
    // sabent que la configuració amb les dames0..i − 1 ho és.
        for (int k=0; k<i; ++k) {
            if (T[k]==T[i] or T[i]-i==T[k]-k
                or T[i]+i==T[k]+k) {
                return false;
        }   }
        return true;
    }

    void escriure () {
        for (int i=0; i<n; ++i) {
            for (int j=0; j<n; ++j) {
                cout << (T[i]==j ? "O" : "* ") ;
            }
            cout << endl;
        }
        cout << endl;
    }

public:

    NReines (int n) {
        this->n = n;
        T = vector<int>(n);
        recursiu(0);
    }
};
```

. . . .

Programa principal

. . . .

```
int main () {
    int n = readint();
    NReines r(n);
}
```

Les n reines — versió 2

```
}
```

Problema de les n-reines.

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

Aquesta és una segona solució utilitzant tornada enrera amb marcatges.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class NReines {

    int n;                          // nombre de reines
    vector<int> T;                  // congiguració actual

    vector<boolean> mc;             // marca de les columnes
    vector<boolean> md1;            // marca de les diagonals 1
    vector<boolean> md2;            // marca de les diagonals 2

    inline int diag1 (int i, int j) {
        return n-j-1 + i;
    }

    inline int diag2 (int i, int j) {
        return i+j;
    }
```

```
void recursiu (int i) {
    if (i==n) {
        escriure();
    } else {
        for (int j=0; j<n; ++j) {
            if (not mc[j] and not md1[diag1(i,j)]
                and not md2[diag2(i,j)]) {
                T[i] = j;
                mc[j] = true;
                md1[diag1(i,j)] = true;
                md2[diag2(i,j)] = true;
                recursiu(i+1);
                mc[j] = false;
                md1[diag1(i,j)] = false;
                md2[diag2(i,j)] = false;
}   }   }   }

void escriure () {
    for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
            cout << (T[i]==j ? "O" : "* ") ;
        }
        cout << endl;
    }
    cout << endl;
}
```

```
public:

    NReines (int n) {
        this->n = n;
        T   = vector<int>(n);
        mc  = vector<int>(n,false);
        md1 = vector<int>(2*n-1,false);
        md2 = vector<int>(2*n-1,false);
        recursiu(0);
    }
};
```

Programa principal

```
int main () {
    int n = readint();
    NReines r(n);
}
```

Les n reines — versió 3

Problema de les n-reines.

Dissenyeu un algorisme que escrigui una possible manera de col·locar n reines en un tauler amb $n \times n$ escacs de forma que cap amenaci cap altra.

Aquesta és una solució utilitzant tornada enrera amb marcatges i un booleà per finalitzar la cerca.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class NReines {

    int n;                      // nombre de reines
    vector<int> T;              // configuració actual
    bool trobat;                // indica si ja s'ha trobat una
solució
    vector<boolean> mc;         // marca de les columnes
    vector<boolean> md1;        // marca de les diagonals 1
    vector<boolean> md2;        // marca de les diagonals 2

    inline int diag1 (int i, int j) {
        return n-j-1 + i;
    }

    inline int diag2 (int i, int j) {
        return i+j;
    }
```

```
void recursiu (int i) {
    if (i==n) {
        trobat = true;
        escriure();
    } else {
        for (int j=0; j<n and not trobat; ++j) {
            if (not mc[j] and not md1[diag1(i,j)]
                and not md2[diag2(i,j)]) {
                T[i] = j;
                mc[j] = true;
                md1[diag1(i,j)] = true;
                md2[diag2(i,j)] = true;
                recursiu(i+1);
                mc[j] = false;
                md1[diag1(i,j)] = false;
                md2[diag2(i,j)] = false;
}   }   }   }

void escriure () {
    for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
            cout << (T[i]==j ? "O" : "* ") ;
        }
        cout << endl;
    }
    cout << endl;
}
```

```
public:

    NReines (int n) {
        this->n = n;
        T   = vector<int>(n);
        mc  = vector<int>(n,false);
        md1 = vector<int>(2*n-1,false);
        md2 = vector<int>(2*n-1,false);
        trobat  = false;
        recursiu(0);
    }

};
```

....

Programa principal

....

```
int main () {
    int n = readint();
    NReines r(n);
}
```

El quadrat llatí

Problema del quadrat llatí.

Un quadrat llatí d'ordre n és una taula $n \times n$ on cada casella està pintada amb un color escollit d'entre n possibles sense que cap fila ni cap columna contingui colors repetits. Dissenyeu un algorisme que escrigui tots els quadrats llatins d'ordre n.

Solució per tornada enrera amb marcatges.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class QuadratLlati {

    int n;                  // nombre de files i columnes
    matrix<int> Q;          // el quadrat llatí
    matrix<boolean> F;      // F[i][c] = c és lliure per fila i
    matrix<boolean> C;      // F[j][c] = c és lliure per columna c
```

```
    void recursiu (int cas) {
        if (cas==n*n) {
            cout << Q << endl;
        } else {
            int i = cas/n, j = cas%n;
            for (int c=0; c<n; ++c) {
                if (F[i][c] and C[j][c]) {
                    Q[i][j] = c;
                    F[i][c] = C[j][c] = false;
                    recursiu(cas+1);
                    F[i][c] = C[j][c] = true;
    }   }   }   }

public:

    QuadratLlati (int n) {
        this->n = n;
        Q       = matrix<int>(n,n);
        F       = matrix<boolean>(n,n,true);
        C       = matrix<boolean>(n,n,true);
        recursiu(0);
    }

};
```

Programa principal

```
int main () {
    int n = readint();
    QuadratLlati qll(n);
}
```

Salts de cavall

Problema dels salts de cavall.

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

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class SaltsDeCavall {

    typedef matrix<int> matriu;

    int n;                      // nombre de files i columnes
    int ox,oy;                  // punt d'orígen
    bool trobat;                // ja s'ha trobat una solució
    matriu M;                   // configuració actual
    matriu S;                   // solució (si trobat)

    inline void provar (int pas, int x, int y) {
        if (not trobat and x>=0 and x<n
            and y>=0 and y<n and M[x][y]==-1) {
            M[x][y] = pas+1;
            recursiu(pas+1,x,y);
            M[x][y] = -1;
    }   }
```

```
    void recursiu (int pas, int x, int y) {
        if (pas==n*n-1) {
            trobat = true;
            S = M;
        } else {
            provar(pas,x+2,y-1); provar(pas,x+2,y+1);
            provar(pas,x+1,y+2); provar(pas,x-1,y+2);
            provar(pas,x-2,y+1); provar(pas,x-2,y-1);
            provar(pas,x-1,y-2); provar(pas,x+1,y-2);
    }   }

public:

    SaltsDeCavall (int n, int ox, int oy) {
        this->n   = n;
        this->ox  = ox;
        this->oy  = oy;
        trobat    = false;
        M         = matriu(n,n,-1);
        M[ox][oy] = 0;
        recursiu(0,ox,oy);
    }

    bool te_solucio () {
        return trobat;
    }

    matriu solucio () {
        return S;
    }
};
```

. . . .

Programa principal. Una solució en 6x6 es troba començant a 0,1 (triga una estoneta).

. . . .

```
int main () {
    int n,ox,oy;
    cin >> n >> ox >> oy;

    SaltsDeCavall sc(n,ox,oy);
    if (sc.te_solucio()) cout << sc.solucio() << endl;
}
```

Planificació de tasques

Problema de l'assignació de tasques.

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

Jordi Petit, novembre 2004.

```
#include "ada.hh"

typedef matrix<double> matriu;

class Tasques {

  matriu T;               // la matriu amb els temps
  int n;                  // nombre de tasques i de treballadors
  vector<int> assig;      // assignació: per cada treballador,
                          // quina tasca li toca
  vector<boolean> feta;// per cada tasca, diu si ja esta feta
  vector<int> sol;        // millor solució fins al moment
  double millor;          // cost de la millor sol fins al moment
```

```
void recursiu (int treb, double t) {
    // treb=índex del treballador, t=temps acumulat
    if (treb==n) {
        if (t<millor) {
            millor = t;
            sol = assig;
        }
    } else {
        for (int tasc=0; tasc<n; ++tasc) {
            if (not feta[tasc]) {
                assig[treb] = tasc;
                feta[tasc] = true;
                if (t+T[treb][tasc]+fita(treb,tasc) <
millor) {
                    recursiu(treb+1,t+T[treb][tasc]);
                }
                feta[tasc] = false;
                assig[treb] = -1;
}   }   }   }

double fita (int treb, int tasc) {
    double f = 0;
    for (int i=treb+1; i<n; ++i) {
        double m = infinit;
        for (int j=0; j<n; ++j) if (not feta[j]) {
            m = min(m,T[i][j]);
        }
        f += m;
    }
    return f;
}
```

```
public:

    Tasques (matriu T) {
        this->T = T;
        n = T.rows();
        assig = vector<int>(n,-1);
        feta = vector<int>(n,false);
        millor = infinit;
        recursiu(0,0);
    }

    vector<int> solucio () {
        return sol;
    }

    double cost () {
        return millor;
    }

};
```

....

El programa principal llegeix n , s'inventa la matriu de temps, crea el solucionador, l'executa i n'escriu la solució.

....

```
int main () {
    int n = readint();
    matriu M = randmatrix(n);
    cout << M;
    Tasques tasques(M);
    cout << tasques.cost() << endl;
    cout << tasques.solucio() << endl;
}
```

Graf Hamiltonià

Problema del graf hamiltonià.

Dissenyeu un algorisme per decidir si un graf G és hamiltonià.

Solució per tornada enrera. Se suposa que el graf és connex, altrament estem fent el burro. Se suposa que les llistes d'adjacència estan ordenades.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

typedef vector< list<int> > graf;
typedef list<int>::iterator iter;

class GrafHamiltonia {

    graf G;                 // el graf
    int n;                  // nombre de vèrtexs
    bool trobat;            // indica si ja s'ha trobat un cicle
    vector<int> s;          // següent de cada vèrtex
                            // (−1 si encara no utilitzat)
    vector<int> S;          // solució (si trobat)
```

```
void recursiu (int v, int t) {
    // v = darrer vèrtex del camí, t = talla del camí
    if (t==n) {
        // cal assegurar-nos que el cicle es pugui tancar
        if (*G[v].begin()==0) {
            s[v] = 0;
            trobat = true;
            S = s;
            s[v] = -1;
        }
    } else {
        iter it=G[v].begin();
        while (not trobat and it!=G[v].end()) {
            int u = *it;
            if (s[u]==-1) {
                s[v] = u;
                recursiu(u,t+1);
                s[v] = -1;
            }
            ++it;
}   }   }
```

```
public:

    GrafHamiltonia (graf G) {
        this->G = G;
        n = G.size();
        s = vector<int>(n,-1);
        trobat = false;
        recursiu(0,1);
    }

    bool te_solucio () {
        return trobat;
    }

    vector<int> solucio () {
        return S;
    }
};
```

```
graf llegir_graf () {
```

. . . .

Llegeix un graf: primer es dóna le nombre de vèrtexs. Després, per a cada vèrtex es dóna el seu grau i la seva llista d'adjacència.

. . . .

```
    graf G;

    int n = readint();
    G = graf(n);
    for (int u=0; u<n; u++) {
        int d = readint();
        for (int i=0; i<d; i++) {
            G[u].push_back(readint());
        }
        G[u].sort();
    }
    return G;
}
```

. . . .

El programa principal llegeix el graf, crea el solucionador i l'executa.

. . . .

```
int main () {
    GrafHamiltonia ham(llegir_graf());
    if (ham.te_solucio()) {
        vector<int> s = ham.solucio();
        cout << 0 << " ";
        for (int u=s[0]; u!=0; u=s[u]) {
            cout << u << " ";
        }
        cout << endl;
}   }
```

El problema del viatjant

Problema del viatjant.

Un viatjant ha de visitar tots els seus clients que viuen en n ciutats diferents. La distància per anar de la ciutat i a la ciutat j ve donada per $D[i][j]$. El viatjant vol, sortint de la seva ciutat, passar un i un sol cop per a cada ciutat on viuen els seus clients i retornar al punt de partida. A més, vol que la distància total recorreguda sigui mínima.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

typedef matrix<double> matriu;

class Viatjant {

  matriu M;              // la matriu de distàncies
  int n;                 // nombre de ciutats
  vector<int> s;         // següent de cada vèrtex
                         // (−1 si encara no utilitzat)
  vector<int> sol;       // millor solució fins al moment
  double millor;         // cost de la millor solució fins al moment
```

```
    void recursiu (int v, int t, double c) {
        // v = darrer vèrtex del camí
        // t = talla del camí
        // c = cost fins ara
        if (t==n) {
            c += M[v][0];
            if (c<millor) {
                millor = c;
                sol = s;
                sol[v] = 0;
            }
        } else {
            for(int u=0; u<n; ++u) if(u!=v and s[u]==-1) {
                if (c+M[v][u]<millor) {
                    s[v] = u;
                    recursiu(u,t+1,c+M[v][u]);
                    s[v] = -1;
    }   }   }   }

public:

    Viatjant (matriu M) {
        this->M = M;
        n = M.rows();
        s = vector<int>(n,-1);
        sol = vector<int>(n);
        millor = infinit;
        recursiu(0,1,0);
    }
```

```
    vector<int> solucio () {
        return sol;
    }

    int seguent (int x) {
        return sol[x];
    }

    double cost () {
        return millor;
    }
};
```

. . . .

El programa principal llegeix n, s'inventa la matriu de distàncies en base a cuitats col·locades aleatòriament, executa el viatjant de comerç i n'escriu el cost de la millor solució.

. . . .

```
int main () {
    int n = readint();
    vector<double> x = randvector(n);
    vector<double> y = randvector(n);
    matriu M = matriu(n,n);
    for (int u=0; u<n; ++u) {
        for (int v=0; v<n; ++v) {
            M[u][v] = sqrt((x[u]-x[v])*(x[u]-x[v]) +
(y[u]-y[v])*(y[u]-y[v]));
    }   }

    double t = now();
    Viatjant vc(M);
    t = now() - t;
    cout << "temps: " << t << endl;
    cout << vc.cost() << endl;
    cout << vc.solucio() << endl;
}
```

La motxilla – versió 1

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

Solució per tornada enrera sense fita superior.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class Motxila {

    int n;                      // nombre d'objectes
    vector<double> p;           // pes de cada objecte
    vector<double> v;           // valor de cada objecte
    double C;                   // capacitat de la motxila
    vector<boolean> s;          // solució activa
    vector<boolean> sol;        // millor solució fins al moment
    double millor;              // cost millor sol fins al moment
```

```
void recursiu (int i, double val, double pes) {
    // i=objecte que toca tractar
    // val=valor acumulat
    // pes=pes acumulat
    if (i==n) {
        if (val>millor) {
            millor = val;
            sol = s;
        }
    } else {
        // 1a possibilitat: intentar agafar l'objecte i
        if (pes+p[i] <= C) {
            s[i] = true;
            recursiu(i+1,val+v[i],pes+p[i]);
        }
        // 2a possibilitat: no agafar l'objecte i
        s[i] = false;
        recursiu(i+1,val,pes);
}   }
```

```
public:

    Motxila (int n, vector<double> p, vector<double> v,
double C) {
        this->n = n;
        this->p = p;
        this->v = v;
        this->C = C;
        s = sol = vector<boolean>(n);
        millor  = 0;
        recursiu(0,0,0);
    }

    vector<boolean> solucio () {
        return sol;
    }

    double cost () {
        return millor;
    }
};
```

. . . .

El programa principal llegeix el nombre d'objectes, s'inventa els pesos, els valors i la capacitat, crea el solucionador, l'executa i n'escriu la solució.

. . . .

```
int main () {
    int n = readint();
    vector<double> p = randvector(n);
    vector<double> v = randvector(n);
    double C = 0.4*n;
    cout << v << endl << p << endl << C << endl;

    Motxila motx(n,p,v,C);
    cout << motx.cost() << endl;
    cout << motx.solucio() << endl;
}
```

La motxilla – versió 2

Problema de la motxila.

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

Solució per tornada enrera amb fita superior: Aquest cop es té en compte la contribució màxima que podrien arribar a tenir tots els objectes a partir del i+1 (encara que no hi capiguen a la motxila). Si fins i tot agafant-los tots no es pogués superar el millor cost trobat fins ara, no cal seguir per aquell camí.

Possiblement, una reordenació astuta dels objectes milloria l'algorisme.

Jordi Petit, novembre 2004.

```
#include "ada.hh"

class Motxila {

    int n;                        // nombre d'objectes
    vector<double> p;             // pes de cada objecte
    vector<double> v;             // valor de cada objecte
    double C;                     // capacitat de la motxila
    vector<boolean> s;            // solució activa
    vector<boolean> sol;          // millor sol fins al moment
    double millor;                // cost millor sol fins al moment
    vector<double> sv;            // suma valors per fita inferior
```

```
void recursiu (int i, double val, double pes) {
    // i=objecte que toca tractar
    // val=valor acumulat
    // pes=pes acumulat
    if (i==n) {
        if (val>millor) {
            millor = val;
            sol = s;
        }
    } else {
        // 1a possibilitat: intentar agafar l'objecte i
        if (pes+p[i] <= C and val+sv[i] > millor) {
            s[i] = true;
            recursiu(i+1,val+v[i],pes+p[i]);
        }
        // 2a possibilitat: no agafar l'objecte i
        if (val+sv[i+1] > millor) {
            s[i] = false;
            recursiu(i+1,val,pes);
}   }   }
```

```
public:

    Motxila (int n, vector<double> p, vector<double> v,
double C) {
        this->n = n;
        this->p = p;
        this->v = v;
        this->C = C;
        s = sol = vector<boolean>(n);
        millor  = 0;
        sv      = vector<double>(n+1);
        sv[n] = 0;
        for (int i=n-1; i>=0; --i) {
            sv[i] = sv[i+1]+v[i];
        }
        recursiu(0,0,0);
    }

    vector<boolean> solucio () {
        return sol;
    }

    double cost () {
        return millor;
    }

};
```

. . . .

El programa principal llegeix el nombre d'objectes, s'inventa els pesos, els valors i la capacitat, crea el solucionador, l'executa i n'escriu la solució.

. . . .

```
int main () {
    int n = readint();
    vector<double> p = randvector(n);
    vector<double> v = randvector(n);
    double C = 0.4*n;
    cout << v << endl << p << endl << C << endl;

    Motxila motx(n,p,v,C);
    cout << motx.cost() << endl;
    cout << motx.solucio() << endl;
}
```

Anàlisi i Disseny d'Algorismes

Algorismes de ramificació i poda

Assignació de tasques

Problema de l'assignació de tasques.

Solució per Branch and Bound.

Jordi Petit i Albert Atserias, novembre 2004.

```
#include "ada.hh"

typedef matrix<int> matriu;

class AssignacioDeTasques {

  int n;              // Nombre de tasques i de treballadors
  matriu C;           // C[i][j] = cost que treb i faci tasca j
  vector<int> sol;    // sol[i] = j ⇔ treballador i fa tasca j
  int millor;         // Cost de la millor solució

  struct conf {
    vector<int> s;        // solució parcial
    vector<boolean> a;    // a[j] = tasca j ja assignada
    int i;                // treballadors 0..i − 1 ja assignats
    int cp;               // cost parcial
    int fi;               // fita inferior al cost restant

    friend bool operator< (conf c1, conf c2) {
        return c1.cp+c1.fi > c2.cp+c2.fi;
        // del revés per la CP!
  } };
```

. . . .

Genera la solució inicial assignant la tasca i al treballador i.

. . . .

```
void generar_solucio_inicial () {
    sol = vector<int>(n);
    millor = 0;
    for (int i=0; i<n; i++) {
        sol[i] = i;
        millor += C[i][i];
}   }
```

. . . .

Genera una configuració parcial corresponent a una assignació parcial buida.

. . . .

```
conf conf_inicial () {
    conf c;
    c.s = vector<int>(n,-1);
    c.a = vector<boolean>(n,false);
    c.i = c.cp = c.fi = 0;
    for (int j=0; j<n; ++j) {
        int m = C[0][j];
        for (int i=1; i<n; ++i) {
            m = min(m,C[i][j]);
        }
        c.fi += m;
    }
    return c;
}
```

. . . .

Retorna el j-èsim fill de la configuració c.

. . . .

```
conf fill (conf c, int j) {
    c.s[c.i] = j;
    c.a[j] = true;
    c.cp += C[c.i][j];
    ++c.i;
    c.fi = 0;
    for (int j=0; j<n; ++j) if (not c.a[j]) {
        int m = 99999999;       // xapuça per l'infinit
        for (int i=0; i<n; ++i) if (c.s[i]==-1) {
            m = min(m,C[i][j]);
        }
        c.fi += m;
    }
    return c;
}
```

```
void BandB () {
    priority_queue<conf> CP;

    generar_solucio_inicial();
    CP.push(conf_inicial());

    while (not CP.empty()) {
        conf c = CP.top();  CP.pop();
        if (c.cp+c.fi < millor) {
            for (int j=0; j<n; ++j) if (not c.a[j]) {
                conf c2 = fill(c,j);
                if (c2.cp+c2.fi < millor) {
                    if (c2.i==n) {
                        millor = c2.cp;
                        sol = c2.s;
                    } else {
                        CP.push(c2);
}   }   }   }   }   }
```

```
public:

    AssignacioDeTasques (matriu C) {
        this->C = C;
        n = C.rows();
        BandB();
    };

    int solucio (int t) {
        return sol[t];
    }

    vector<int> solucio () {
        return sol;
    }

    int cost () {
        return millor;
    }
};

int main () {
    int n = 4;
    matriu M = matriu(n,n);
    M[0][0]=11; M[0][1]=12; M[0][2]=18; M[0][3]=40;
    M[1][0]=14; M[1][1]=15; M[1][2]=13; M[1][3]=22;
    M[2][0]=11; M[2][1]=17; M[2][2]=19; M[2][3]=23;
    M[3][0]=17; M[3][1]=14; M[3][2]=20; M[3][3]=28;

    AssignacioDeTasques A(M);
    cout << A.solucio() << endl << A.cost() << endl;
}
```

El problema del viatjant

Problema del viatjant. Solució per BandB.

Jordi Petit i Albert Atserias. Novembre 2004

```
#include "ada.hh"

typedef matrix<double> matriu;

class Viatjant {

    matriu M;                       // la matriu de distàncies
    int n;                          // nombre de ciutats
    vector<int> sol;                // millor solució fins al moment
    double millor;                  // cost millor solució
    vector<double> mp;              // més proper de cada ciutat

    struct conf {
        vector<int> s;              // solució parcial
        int l;                      // nombre de vèrtexs en el camí
        int d;                      // darrer vèrtex en el camí
        double cp;                  // cost parcial fins ara
        double fi;                  // fita inferior al cost restant

        friend bool operator< (conf a, conf b) {
            return a.cp + a.fi > b.cp + b.fi;
    }   };
```

Calcula una solució i el seu cost de forma ràpida. En aquest cas, tria cada cop el vèrtex no visitat més proper.

```
void solucio_inicial () {
    sol = vector<int>(n,-1);
    millor = 0;
    int u = 0;
    for (int nb = 1; nb<n; nb++) {
        int w = -1;
        double m = infinit;
        for (int v=0; v<n; ++v) {
            if (u!=v and sol[v]==-1 and M[u][v] < m) {
                m = M[u][v];
                w = v;
        }   }
        sol[u] = w;
        millor += M[u][w];
        u = w;
    }
    sol[u] = 0;
    millor += M[u][0];
}
```

. . . .

Calcula una configuració inicial. Triem sortir del vèrtex 0.

. . . .

```
conf conf_inicial () {
    conf c;
    c.s = vector<int>(n,-1);
    c.l = 1; c.d = 0; c.cp = 0; c.fi = 0;
    for (int u=0; u<n; ++u) {
        c.fi += mp[u];
    }
    return c;
}
```

. . . .

Calcula la taula mp per les ciutats més properes.

. . . .

```
void calcular_mp () {
    mp = vector<double>(n);
    for (int u=0; u<n; ++u) {
        mp[u] = infinit;
        for (int v=0; v<n; v++) if (v!=u) {
            mp[u] = min(mp[u],M[u][v]);
}   }   }
```

Completa una configuració c anant de u a v

```
conf completar (conf c, int u, int v) {
    c.s[u] = v;
    c.d = v;
    ++c.l;
    c.cp += M[u][v];
    c.fi -= mp[u];
    if (c.l==n) {
        c.s[v] = 0;
        c.cp += M[v][0];
        c.fi -= mp[v];
    }
    return c;
}
```

El branch and bound

```
void BandB () {
    priority_queue<conf> CP;

    calcular_mp();
    solucio_inicial();
    CP.push(conf_inicial());

    while (not CP.empty()) {
        conf c = CP.top();  CP.pop();
        if (c.cp+c.fi < millor) {
            int u = c.d;
            for (int v=0; v<n; ++v) {
                if (v!=u and c.s[v]==-1) {
                    conf c2 = completar(c,u,v);
                    if (c2.cp+c2.fi < millor) {
                        if (c2.l==n) {
                            millor = c2.cp;
                            sol = c2.s;
                        } else {
                            CP.push(c2);
}   }   }   }   }   }   }
```

```
public:

    Viatjant (matriu M) {
        this->M = M;
        n = M.rows();
        BandB();
    }

    vector<int> solucio () {
        return sol;
    }

    int seguent (int x) {
        return sol[x];
    }

    double cost () {
        return millor;
    }

};
```

Programa principal:

Llegeix el nombre de ciutats, tira les seves coordenades aleatoriament en el pla i construeix la matriu de distàncies amb la seva distància Euclídea. Després, resol el problema i escriu la seva solució.

```
int main () {
    int n = readint();
    vector<double> x = randvector(n);
    vector<double> y = randvector(n);
    matriu M = matriu(n,n);
    for (int u=0; u<n; ++u) {
        for (int v=0; v<n; ++v) {
            M[u][v] = sqrt((x[u]-x[v])*(x[u]-x[v]) +
(y[u]-y[v])*(y[u]-y[v]));
    }   }

    Viatjant vc(M);
    cout << vc.cost() << endl;
    cout << vc.solucio() << endl;
}
```

La motxilla

Problema de la motxila.

Solució per BandB (acaba sent més lenta que backtracking!).

Albert Atserias i Jordi Petit, novembre 2004.

```
#include <stack>
#include "ada.hh"

class Motxila {

  int n;                        // nombre d'objectes
  vector<double> p;             // pes de cada objecte
  vector<double> v;             // valor de cada objecte
  double C;                     // capacitat de la motxila
  vector<boolean> sol;          // millor solució fins al moment
  double millor;                // cost millor solució
  vector<double> sv;            // suma de valors (fita superior)

    struct conf {
      vector<boolean> s;        // solució parcial
      int i;                    // número d'objectes considerats
      double vp;                // valor parcial
      double pp;                // pes parcial
      double fi;                // fita superior

      friend bool operator< (conf a, conf b) {
          return a.vp + a.fi < b.vp + b.fi;
      }
    };
```

```
typedef priority_queue<conf> cuaprio;

conf conf_inicial () {
    conf c;
    c.s = vector<boolean>(n);
    c.i = 0;  c.vp = c.pp = 0;
    c.fi = sv[0];
    return c;
}
```

....
Solució voraç: ordena decreixentment per valor/pes i va omplint la motxila fins que no n'hi capiguen més.
....

```
void generar_solucio_inicial () {
    vector< pair<double,int> > d(n);
    for (int i=0; i<n; i++) {
        d[i].first = -v[i]/p[i];
        d[i].second = i;
    }
    sort(d.begin(),d.end());
    sol = vector<boolean>(n,false);
    millor = 0;
    double pp = 0;
    for (int i=0; i<n; i++) {
        int j = d[i].second;
        pp += p[j];
        if (pp > C) break;
        sol[j] = true;
        millor += v[j];
}   }
```

. . . .

Calcula el vector `sv` amb la suma de valors per a la fita superior.

. . . .

```
void calcular_sv () {
    sv = vector<double>(n+1);
    sv[n] = 0;
    for (int i=n-1; i>=0; --i) {
        sv[i] = sv[i+1]+v[i];
}   }
```

. . . .

El branch and bound

. . . .

```
void BandB () {
    cuaprio CP;

    calcular_sv();
    generar_solucio_inicial();
    CP.push(conf_inicial());

    while (not CP.empty()) {
        conf c = CP.top(); CP.pop();
        if (c.vp+c.fi > millor) {
            tractar_seguent(c,false,CP);
            tractar_seguent(c,true,CP);
}   }   }
```

. . . .

Genera un fill d'una configuració `c` escollint agafar o no l'objecte següent. Tracta la nova configuració actualitzant la millor, encuant-la o rebutjant-la.

. . . .

```
void tractar_seguent (conf c, bool af, cuaprio& CP) {
    int i = c.i;
    c.s[i] = af;
    if (af) {
        c.vp += v[i];
        c.pp += p[i];
    }
    c.fi = sv[i+1];
    c.i++;
    if (c.pp <= C and c.vp+c.fi > millor) {
        if (c.i == n) {
            sol = c.s;
            millor = c.vp;
        } else {
            CP.push(c);
}   }   }
```

```
public:

    Motxila (vector<double> p, vector<double> v, double C)
    {
        this->n = p.size();
        this->p = p;
        this->v = v;
        this->C = C;
        BandB();
    }

    vector<boolean> solucio () {
        return sol;
    }

    double cost () { return millor;  }
};
```

....

El programa principal llegeix el nombre d'objectes, s'inventa els pesos, els valors i la capacitat, crea el solucionador, l'executa i n'escriu la solució.

....

```
int main () {
    int n = readint();
    vector<double> p = randvector(n);
    vector<double> v = randvector(n);
    double C = 0.4*n;
    cout << v << endl << p << endl << C << endl;
    Motxila motx(p,v,C);
    cout << motx.cost() << endl << motx.solucio() << endl;
}
```

Anàlisi i Disseny d'Algorismes

Algorismes de programació dinàmica

Números binomials — versió 1

Càlcul dels nombres binomials.

Dissenyeu un algorisme recursiu per calcular $\binom{n}{k}$. Useu la igualtat

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

Salvador Roura, desembre 2004.

```
#include "ada.hh"

int Bin(int n, int k) {
  if (k < 0 or k > n) return 0;
  if (n == 0) return 1;
  return Bin(n-1, k) + Bin(n-1, k-1);
}

int main() {
  int n, k;
  while (cin >> n >> k) cout << Bin(n, k) << endl;
}
```

Números binomials — versió 2

Càlcul dels nombres binomials.

Dissenyeu un algorisme per calcular $\binom{n}{k}$, usant programació dinàmica.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class Binomial {

  vector<vector<int> > M;

  int consulta (int i, int j) {
    return j < 0 or j > i ? 0 : M[i][j];
  }

public:

  Binomial() {
    M = vector<vector<int> >(1, vector<int>(1, 1));
  }

  int operator() (int n, int k) {
    int m = M.size();
    for (int i = m; i <= n; ++i) {
      M.push_back(vector<int>(i+1));
      for (int j = 0; j <= i; ++j)
        M[i][j] = consulta(i-1, j) + consulta(i-1, j-1);
    }
    return consulta(n, k);
  }
};
```

```
int main() {
  Binomial Bin;
  int n, k;
  while (cin >> n >> k) cout << Bin(n, k) << endl;
}
```

Números binomials — versió 3

Càlcul dels nombres binomials.

Dissenyeu un algorisme recursiu per calcular $\binom{n}{k}$. Useu la igualtat

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

Salvador Roura, desembre 2004.

```
#include "ada.hh"

int Bin(int n, int k) {
  if (k < 0 or k > n) return 0;
  if (k == 0) return 1;
  return Bin(n, k-1)*(n-k+1)/k;
}

int main() {
  int n, k;
  while (cin >> n >> k) cout << Bin(n, k) << endl;
}
```

Números de Fibonacci — versió 1

Càlcul dels nombres de Fibonacci.

Recursiu.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

int Fib(int n) {
  if (n <= 1) return n;
  return Fib(n-2) + Fib(n-1);
}

int main() {
  int n;
  while (cin >> n) cout << Fib(n) << endl;
}
```

Números de Fibonacci — versió 2

Càlcul dels nombres de Fibonacci.

Amb programació dinàmica.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class Fibonacci {
  vector<int> V;

public:

  Fibonacci() {
    V.push_back(0);
    V.push_back(1);
  }

  int operator()(int n) {
    int m = V.size();
    for (int i = m; i <= n; ++i)
      V.push_back(V[i-2] + V[i-1]);
    return V[n];
  }
};

int main() {
  Fibonacci Fib;
  int n;
  while (cin >> n) cout << Fib(n) << endl;
}
```

Números de Fibonacci — versió 3

Càlcul dels nombres de Fibonacci.

Recursiu amb immersió.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

// Post: a = F(n), b = F(n+1)
void Fib(int n, int &a, int &b) {
  if (n == 0) {
    a = 0;
    b = 1;
  } else if (n == 1) a = b = 1;
    else {
    Fib(n-2, a, b);
    a = a + b;
    b = a + b;
  }
}

int main() {
  int n;
  while (cin >> n) {
    int a, b;
    Fib(n, a, b);
    cout << a << endl;
  }
}
```

Laberint

Trobar el nombre de camins des d'una cantonada a una altra.

Dissenyeu un algorisme que, donada una taula amb $n \geq 1$ files i $m \geq 1$ columnes, i una seqüència de posicions de la taula per les quals no es pot passar, calculi de quantes maneres diferents es pot anar de la posició $(1,1)$ fins a la posició (n,m) fent només passos cap a la dreta i cap amunt.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class Laberint {
  int n, m;
  matrix<boolean> M;

public:

  Laberint(int n, int m)  {
    this->n = n;  this->m = m;
    M = matrix<boolean>(n, m, true);
  }

  void Marca(int x, int y) {
    if (x >= 0 and x < n and y >= 0 and y < m)
        M[x][y] = false;
  }
```

```
  int Camins() {
    // A[i][j] = nombre maneres d'anar de (i, j) fins a (n-1, m-1)
    matrix<int> A(n, m, -10);
    A[n-1][m-1] = M[n-1][m-1];
    for (int j = m-2; j >= 0; --j)
        A[n-1][j] = M[n-1][j]*A[n-1][j+1];
    for (int i = n-2; i >= 0; --i) {
      A[i][m-1] = M[i][m-1]*A[i+1][m-1];
      for (int j = m-2; j >= 0; --j)
        A[i][j] = M[i][j]*(A[i][j+1] + A[i+1][j]);
    }
    return A[0][0];
  }
};

int main() {
  Laberint L(readint(), readint());
  int x, y;
  while (cin >> x >> y) L.Marca(x, y);
  cout << L.Camins() << endl;
}
```

Maximització del resultat final s'una seqüència de sumes i productes

Maximització del resultat final s'una seqüència de sumes i productes.

Considereu les seqüències generades per la gramàtica següent:

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

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

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class Sequencia {
  int n;                // nombre d'operands
  vector<double> N;     // n operands
  vector<char> C;       // n-1 operadors (+ o be *)
  // M[i][m] = maxim valor d'una sequencia de mida m+1 que comenca a i.
  vector<vector<double> > M;

  static double opera(double a, char c, double b) {
    if (c == '+') return a+b;
    return a*b;
  }
```

```
public:

  Sequencia(double x) {
    n = 1;
    N.push_back(x);
    M.push_back(vector<double>(1, x));
  }

  void Concat(char c, double x) {
    ++n;  C.push_back(c);  N.push_back(x);

    // Cal calcular els resultats maxims per a les noves
    // subsequencies: totes les que acaben amb x.
    M.push_back(vector<double>(1, x));
    for (int i = n-2; i >= 0; --i) {
      int m = n-i; // mida des de la posicio i fins al final

      // comencem calculant el maxim resultat
      // d'operar 1 amb els altres m-1
      double maxim = opera(M[i][0], C[i], M[i+1][m-2]);

      // despres calculem 2 amb m-2, 3 amb m-3, etc,
      // i ens quedem amb el maxim
      for (int j = 2; j < m; ++j)
        maxim = max(maxim,
            opera(M[i][j-1], C[i+j-1], M[i+j][m-j-1]));
      M[i].push_back(maxim);
    }
  }
```

```
  double Maxim() {
    return M[0][n-1];
  }
};

int main() {
  Sequencia Seq(readdouble());
  cout << Seq.Maxim() << endl;
  char c;
  while (cin >> c) {
    Seq.Concat(c, readdouble());
    cout << Seq.Maxim() << endl;
  }
}
```

Múltiples de 3

Múltiples de 3.

Dissenyeu i analitzeu un algorisme que calculi quants nombres de n bits, m dels quals són u (per tant, amb $n - m$ zeros), són múltiples de 3.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class Multiples3 {
  typedef vector<int> VI;
  typedef vector<VI> VVI;
  typedef vector<VVI> VVVI;

  // M[n][m][k] = nombre nombres amb n bits i m uns que
  // son congruents amb k mod 3
  VVVI M;

  int consulta(int i, int j, int k) {
    if (j < 0 or j > i) return 0;
    return M[i][j][k];
  }
```

```
public:

  Multiples3() {
    M = VVVI(1, VVI(1, VI(3)));
    M[0][0][0] = 1;
    M[0][0][1] = M[0][0][2] = 0;
  }

  int operator()(int n, int m) {
    int s = M.size();
    for (int i = s; i <= n; ++i) {
      M.push_back(VVI(i+1, VI(3)));
      for (int j = 0; j <= i; ++j) {
        M[i][j][0] = consulta(i-1, j, 0) +
                     consulta(i-1, j-1, 1);
        M[i][j][1] = consulta(i-1, j, 2) +
                     consulta(i-1, j-1, 0);
        M[i][j][2] = consulta(i-1, j, 1) +
                     consulta(i-1, j-1, 2);
      }
    }
    return consulta(n, m, 0);
  }
};

int main() {
  Multiples3 Mul3;
  int n, m;
  while (cin >> n >> m) cout << Mul3(n, m) << endl;
}
```

Transformació d'una taula a un palíndrom

Tranformació d'una paraula a palíndrom.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

int minim(int a, int b, int c) {
  return min(a, min(b, c));
}

int Palin(string s) {
  int n = s.size();
  // M[i][mida] = nb minim de canvis per a s[i..i+mida-1]
  matrix<int> M(n, n+1);
  for (int i = 0; i < n; ++i) M[i][0] = M[i][1] = 0;
  for (int mida = 2; mida <= n; ++mida)
    for (int i = 0; i <= n - mida; ++i)
      M[i][mida] = ( s[i] == s[i+mida-1] ?
                     M[i+1][mida-2] :
                     1 + minim(M[i+1][mida-2],
                               M[i][mida-1],
                               M[i+1][mida-1]) );
  return M[0][n];
}

int main() {
  string s;
  while (cin >> s) cout << Palin(s) << endl;
}
```

El joc del peons

Peons encadenats.

Salvador Roura, desembre 2004.

```
#include "ada.hh"

class PeonsEncadenats {
  int m;
  // M[n][j] = nombre maneres de posar n peons en un taulell
  // n x m amb l'ultim peo a la columna j, 0 ¡= j ¡ m.
  vector<vector<int> > M;
  vector<int> res;

  int consulta(int i, int j) {
    if (j < 0 or j >= m) return 0;
    return M[i][j];
  }

public:

  PeonsEncadenats(int m) : m(m) {
    M = vector<vector<int> >(2, vector<int>(m, 1));
    res = vector<int>(2, m);
    // M[0] i res[0] no s'usen
  }
```

```
  int operator()(int n) {
    int s = M.size();
    for (int i = s; i <= n; ++i) {
      M.push_back(vector<int>(m));
      int acum = 0;
      for (int j = 0; j < m; ++j)
        acum += M[i][j] =
            consulta(i-1, j-1) + consulta(i-1, j+1);
      res.push_back(acum);
    }
    return res[n];
  }
};

int main() {
  // m es fixa durant tot el programa
  PeonsEncadenats Peons(readint());
  int n;
  while (cin >> n) cout << Peons(n) << endl;
}
```