1.2 Stacks: stack

Stacks.
Reads a sequence of numbers and writes it backwards.

```
#include <stack>
#include <iostream>

using namespace std;

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

1.3 Queues: queue

Queues.
Reads a sequence of numbers and writes it straight.

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

int main() {
    queue<int> q;
    int x;
    while (cin >> x) q.push(x);
    while (not q.empty()) {
        cout << q.front() << endl;
        q.pop();
}   }
```

1.4 Priority queues: priority_queue

Priority queues.
Reads a sequence of numbers and writes it in decreasing order.

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

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

Priority queues with inverted order.
Reads a sequence of numbers and writes it in increasing order. The third parameter of the type is the important one, but providing the second is required.

```
priority_queue<int, vector<int>, greater<int>> pq;
```

```
SET
===

Un set<T> és un conjunt de T.

Els elements del conjunt es guarden ordenats de menor a major.

(internament estan implementats amb arbres binaris de cerca balancejats)

Cal fer: #include <set>

Assumim que iterator és sinònim de set<T>::iterator.

Un iterator it es mou cap endavant amb ++it i cap endarrera amb --it

Per accedir a l'element apuntat per un iterator it: *it

.............................................................

pair<iterator,bool> insert ( const T& x );

Afegeix x al conjunt.

Si no hi era, retorna un iterador que apunta on s'ha ficat x, i true.

Sinó retorna un iterador que apunta on ja hi havia x, i false.

Cost: Theta(log n)
```

```
UNORDERED_MAP
=============

Com el map, però no es garanteix que recórrer el map des de begin()
fins a end() respecti l'ordre de les claus.

insert, find, erase funcionen en temps lineal en el cas pitjor, però
en temps constant en mitjana.

(internament estan implementats amb taules de hash)

#include <unordered_map>
#include <iostream>

using namespace std;

int main() {
  unordered_map<string, int> m;
  string x;
  while (cin >> x) ++m[x];
  for(auto p : m)
    cout << p.first << " " << p.second << endl;
}
```

```
iterator begin(): retorna l'iterador a l'element més petit
iterator end  (): retorna l'iterador al següent de l'element més gran

Cost: Theta(1)

.................................................................

iterator find ( const T& x ) const

Busca l'element x al conjunt.
Si el troba, retorna un iterador que hi apunta.
Sinó retorna end().

void erase ( iterator it )

Elimina l'element apuntat per it.

Cost: Theta(1) amortit

int erase ( const T& x ):

Si x pertany al conjunt, l'elimina i retorna 1.
Sinó retorna 0.

Cost: Theta(log n)

Ex. (llegeix dues seqüències d'enters acabades en zero i escriu seva
intersecció)

int main() {
  set<int> s1, s2;
  int x;
  while (cin >> x and x != 0) s1.insert(x);
  while (cin >> x and x != 0) s2.insert(x);

  for (set<int>::iterator it = s1.begin(); it != s1.end(); ++it)
    if (s2.find(*it) != s2.end())
      cout << *it << endl;
}
```

MAP
===

Un map<K,V> és un diccionari de claus K i valors V. Es comporta de manera semblant a un conjunt de parells (clau, valor) (és a dir, set<pair<K,V> >) on no es poden repetir claus.

Els parells estan ordenats per clau de menor a major.

(internament estan implementats amb arbres binaris de cerca balancejats)

Cal fer: #include <map>

Assumim que iterator és sinònim de map<K,V>::iterator.

Un iterator it es mou cap endavant amb ++it i cap endarrera amb --it
Per accedir al parell apuntat per it: *it
Per accedir a la clau apuntada per it: (*it).first o it->first
Per accedir al valor apuntat per it: (*it).second o it->second

....

```
pair<iterator,bool> insert ( const pair<K,V>& p );
```

Afegeix el parell p.

Si no hi havia cap parell amb aquesta clau, retorna un iterador que apunta on s'ha ficat p, i true.

Sinó retorna l'iterador que apunta al parell que ja hi havia amb la mateixa clau, i false.

Cost: Theta(log n)

```
iterator begin(): retorna l'iterador al parell amb clau més petita
iterator end  (): retorna l'iterador al següent al parell amb clau més gran
```

Cost: Theta(1)

```
iterator find ( const K& k ) const
```

Busca un parell amb clau k.
Si el troba, retorna un iterador que hi apunta.
Sinó retorna end().

```
void erase ( iterator it )
```

Elimina el parell apuntat per it.

Cost: Theta(1) amortit

```
int erase ( const K& k ):
```

Si hi ha un parell amb clau k, l'elimina i retorna 1.
Sinó retorna 0.

```
typedef pair<char,int> P;
typedef  map<char,int> M;

int main () {

  M m;

  m.insert( P('a', 10) );
  m.insert( make_pair('c', 30) );
  m['d'] = 40;

//    L'operador [ ] admet com a argument una clau k. Llavors:
//
//    Si ja hi havia un parell amb la clau, es retorna una referència al
//    camp second (valor) del parell que ja existia amb aquesta clau.
//
//    Sino, s'inserta un parell amb aquesta clau i el constructor per
//    defecte del tipus V (per ex., per a tipus bàsics de C++
//    numèrics, assigna a 0). Llavors es retorna una referència al
//    camp second (valor) d'aquest parell.

  m.erase('c');

  for (M::iterator it = m.begin(); it != m.end(); ++it)
    cout << it->first << " " << it->second << endl;
}
```

té sortida

```
a 10
d 40
```

UNORDERED_SET
=============

Com el set, però no es garanteix que recórrer el set des de begin() fins a end() respecti l'ordre dels elements.

insert, find, erase funcionen en temps lineal en el cas pitjor, però en temps constant en mitjana.

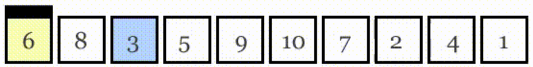
(internament estan implementats amb taules de hash)

```
#include <iostream>
#include <unordered_set>

using namespace std;

int main() {

  unordered_set<int> s1, s2;
  int x;

  while (cin >> x and x != 0)
    s1.insert(x);

  while (cin >> x and x != 0)
    s2.insert(x);

  for (auto y : s1)
    if (s2.find(y) != s2.end())
      cout << y << endl;
}
```

2.1 Selection Sort

Selection Sort.

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

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



Yellow is smallest number found
Blue is current item
Green is sorted list

6 5 3 1 8 7 2 4

2.2 Insertion Sort — 1

Insertion Sort (version 1).

```
template <typename elem>

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

2.3 Insertion Sort — 2

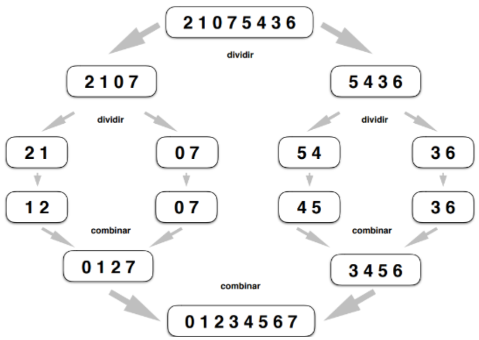
Insertion Sort (version 2).
Avoids swap-chaining: instead of doing swaps, the elements are shifted to the right (this improves from 3 assignments per iteration to 1).

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

2.4 Insertion Sort — 3

Insertion Sort (version 3).
To avoid the final test at each iteration, the smallest element is placed at the first position of the table.

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



2.5 Bubblesort

Bubblesort.

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

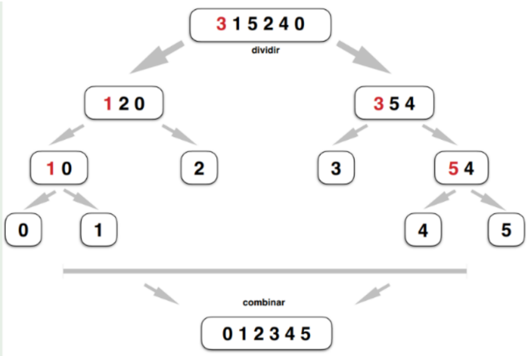
6 5 3 1 8 7 2 4

Mergesort (version 1).

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

template <typename elem>
void merge_sort_1 (vector<elem>& v, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        merge_sort_1(v, l, m);
        merge_sort_1(v, m + 1, r);
        merge(v, l, m, r);
}   }

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

Mergesort (version 2).
Cuts the recursion when the subvector is "small enough"at which point it uses insertion sort.

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

template <typename elem>
void merge_sort_2 (vector<elem>& v, int l, int r) {
    const int critical_size = 50;
    if (r - l < critical_size) {
        ins_sort(v, l, r);
    } else {
        int m = (l + r) / 2;
        merge_sort_2(v, l, m);
        merge_sort_2(v, m + 1, r);
        merge(v, l, m, r);
}   }
```



Mergesort with bottom-up merging.

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

Quicksort with Hoare's partition (version 1).

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

template <typename elem>
void quick_sort_1 (vector<elem>& v, int l, int r) {
    if (l < r) {
        int q = partition(v, l, r);
        quick_sort_1(v, l, q);
        quick_sort_1(v, q + 1, r);
}   }

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

Quicksort (version 2).
Pivot is selected at random.

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

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

Quicksort (version 3).
Stops sorting when the subvector is "small enough". At the end, a last pass is made with insertion sort.

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

template <typename elem>
void quick_psort_3 (vector<elem>& v, int l, int r) {
    const int critical_size = 100;
    if (r - l >= critical_size) {
        int q = partition(v, l, r);
        quick_psort_3(v, l, q);
        quick_psort_3(v, q + 1, r);
}   }
```

6 5 3 1 8 7 2 4

Heapsort with ADT.

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

Heapsort.

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

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

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

10	4	8	5	12	2	6	11	3	9	7	1

```
#include <vector>
#include <list>

using namespace std;

typedef vector<vector<int>> graph;
```

Depth-first search.

The function returns the list of vertices according to its order of visit in a depth-first search. This code uses C++11.

```
#include <stack>
```

Recursive version. Cost: $\Theta(|V| + |E|)$.

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

list<int> dfs_rec (const graph& G) {
    int n = G.size();
    list<int> L;
    vector<boolean> vis(n, false);
    for (int u = 0; u < n; ++u)  {
        dfs_rec(G, u, vis, L);
    }
    return L;
}
```

Iterative version: the order of visit is different than that of the recursive version because the neighbors leave the stack in reverse order. Cost: $\Theta(|V| + |E|)$.

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

    for (int u = 0; u < n; ++u) {
        S.push(u);
        while (not S.empty()) {
            int v = S.top();  S.pop();
            if (not vis[v]) {
                vis[v] = true;  L.push_back(v);
                for (int w : G[v]) {
                    S.push(w);
    }   }   }   }
    return L;
}
```

Breadth-first search

The function returns the list of the vertices according to the order in which they are visited in a breadth-first search.

Direct version: same as iterative dfs but with queue instead of stack; enqueues each vertex as many times as its indegree. Cost: $\Theta(|V| + |E|)$.

```
list<int> bfs_1 (const graph& G) {
    int n = G.size();
    list<int> L;
    queue<int> Q;
    vector<boolean> vis(n, false);

    for (int u = 0; u < n; ++u) {
        Q.push(u);
        while (not Q.empty()) {
            int v = Q.front(); Q.pop();
            if (not vis[v]) {
                vis[v] = true; L.push_back(v);
                for (int w : G[v])  {
                    Q.push(w);
    }   }   }   }
    return L;
}
```

Better version: avoids enqueuing a vertex more than once. Cost: $\Theta(|V| + |E|)$.

```
list<int> bfs_2 (const graph& G) {
    int n = G.size();
    list<int> L;
    queue<int> Q;
    vector<boolean> enc(n, false);

    for (int u = 0; u < n; ++u) {
        if (not enc[u]) {
            Q.push(u); enc[u] = true;
            while (not Q.empty()) {
                int v = Q.front(); Q.pop();
                L.push_back(v);
                for (int w : G[v])  {
                    if (not enc[w]) {
                        Q.push(w);  enc[w] = true;
    }  }  }  }  }

      return L;
  }
```

Topological sort.

Given a directed acyclic graph, returns a list with its vertices sorted in topological sort, that is, in such a way that a vertex v does not appear before a vertex u if there is a path from u to v. Cost: $\Theta(|V| + |E|)$.

```
list<int> topological_sort(const graph& G) {
    int n = G.size();
    vector<int> ge(n, 0);
    for (int u = 0; u < n; ++u) {
        for (int v : G[u]) {
            ++ge[v];
    }  }

    stack<int> S;
    for (int u = 0; u < n; ++u) {
        if (ge[u] == 0) {
            S.push(u);
    }  }

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

Dijkstra's algorithm.

Instead of decreasing the priority associated to a vertex, the algorithm reinserts that vertex with the new priority. A consequence of this is that each vertex can be inserted as many times as its indegree. This does not affect the asymptotic running time which is still $\Theta((|V| + |E|)\log(|V|))$.

```
typedef pair<double, int> WArc;            // weighted arc
typedef vector<vector<WArc>> WGraph;       // weighted digraf

void dijkstra(const WGraph& G, int s, vector<double>& d, vector<int>& p) {
    int n = G.size();
    d = vector<double>(n, infinit);  d[s] = 0;
    p = vector<int>(n, -1);
    vector<boolean> S(n, false);
    priority_queue<WArc, vector<WArc>, greater<WArc> > Q;
    Q.push(WArc(0, s));

    while (not Q.empty()) {
        int u = Q.top().second;  Q.pop();
        if (not S[u]) {
            S[u] = true;
            for (WArc a : G[u]) {
                int v = a.second;
                double c = a.first;
                if (d[v] > d[u] + c) {
                    d[v] = d[u] + c;
                    p[v] = u;
                    Q.push(WArc(d[v], v));
}  }  }  }  }
```

5.6 Minimum Spanning Tree: Prim's Algorithm

Prim's algorithm.

Edges are inserted in the priority queue with their signs reversed. Alternatively we could have redefined the order of the priority queue. Running time is $\Theta((|V| + |E|)\log(|V|))$.

```
#include "eda.hh"

typedef pair< double, pair<int, int> > WEdge;
typedef vector< vector< pair<double, int> > > WGraph;

void MST(const WGraph& G, vector<int>& parent) {
    vector<bool> used(G.size(), false);
    priority_queue<WEdge> Q;
    Q.push({0.0, {0, 0}});

    while (not Q.empty()) {
        double p = Q.top().first;
        int u = Q.top().second.first;
        int v = Q.top().second.second;
        Q.pop();
        if (not used[v]) {
            used[v] = true;
            parent[v] = u;
            for (auto e : G[v]) {
                double p = e.first;
                int w = e.second;
                Q.push({-p, {v, w}});
}   }   }   }
```

***n*-queens problem.**

Write an algorithm that writes a way of placing n queens on an $n \times n$ chessboard in such a way that no queen threatens another.

```
#include "eda.hh"

class NQueens {

    int n;                      // number of queens
    vector<int> T;              // current configuration
    bool found;                 // indicates if a solution has been found
    vector<boolean> mc;         // column labeling
    vector<boolean> md1;        // diagonal 1 labeling
    vector<boolean> md2;        // diagonal 2 labeling

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

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

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

    void write() {

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

public:

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

```
class NQueens {

    int n;                  // number of queens
    vector<int> T;          // current configuration

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

    // Indicates if the configuration with queens 0..i is legal
    // knowing that the configuration with queens 0..i − 1 is.
    bool legal(int i) {
        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 write() {
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    cout << (T[i]==j ? "O " : "* ") ;
                }
                cout << endl;
            }
            cout << endl;
        }

    public:

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

Latin square problem.

A latin square of order n is an $n \times n$ table in which every square is colored by one of n colors, in such a way that no row or column contains a repeated color. Write an algorithm that writes all the latin squares of order n.

```
#include "eda.hh"

class LatinSquare {

    int n;                    // number of rows and columns
    matrix<int> Q;            // the latin square
    matrix<boolean> F;        // F[i][c] = c is allowed in row i
    matrix<boolean> C;        // F[j][c] = c is allowed in column j

    void recursive(int cas) {
        if (cas == n*n) {
            cout << Q << endl;

        } else {
            int i = cas/n;
            int 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;
                    recursive(cas+1);
                    F[i][c] = C[j][c] = true;
    }   }   }   }

 public:

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

Knight-jumps problem.

A knight is placed on a given square of an $n \times n$ chessboard. Write an algorithm to determine if there is a way to visit every square of the board by moving it $n^2 - 1$ times.

```
#include "eda.hh"

class KnightJumps {

    typedef matrix<int> board;

    int n;                    // number of rows and columns
    int ox,oy;                // origin
    bool found;               // a solution is found
    board M;                  // current configuration
    board S;                  // solution (if found)

inline void try_it(int step, int x, int y) {
    if (not found and x >= 0 and x < n
        and y >= 0 and y < n and M[x][y] == -1) {
        M[x][y] = step + 1;
        recursive(step + 1, x, y);
        M[x][y] = -1;
}   }

void recursive(int step, int x, int y) {
    if (step == n*n-1) {
        found = true;
        S = M;
    } else {
        try_it(step, x+2, y-1); try_it(step, x+2, y+1);
        try_it(step, x+1, y+2); try_it(step, x-1, y+2);
        try_it(step, x-2, y+1); try_it(step, x-2, y-1);
        try_it(step, x-1, y-2); try_it(step, x+1, y-2);
}   }

public:

    KnightJumps(int n, int ox, int oy) {
        this->n   = n;
        this->ox  = ox;
        this->oy  = oy;
        found     = false;
        M         = board(n, n, -1);
        M[ox][oy] = 0;
        recursive(0, ox, oy);
    }

    bool has_a_solution() {
        return found;
    }
            board solution() {
                return S;
            }
        };
```

Main program. A solution for 6x6 can be found starting at 0,1 (it takes a while).

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

    KnightJumps kj(n,ox,oy);
    if (kj.has_a_solution()) cout << kj.has_a_solution() << endl;
}
```

Scheduling problem.

A boss has n workers for n tasks. The time that worker i takes to complete task j is given by $T[i][j]$. He wants to assign a task to each worker so as to minimize the total time span.

```
#include "eda.hh"

typedef matrix<double> time_matrix;

class Scheduling {

    time_matrix T;          // time matrix
    int n;                  // number of tasks and worksers
    vector<int> assig;      // assignment: each worker gets a task
    vector<boolean> done;   // for each tasks, indicates if taken
    vector<int> sol;        // best solution so far
    double best;            // cost of the best solution so far

    void recursive(int worker, double t) {
        // worker = index of the worker, t = accumulated time
        if (worker == n) {
            if (t < best) {
                best = t;
                sol = assig;
            }
        } else {
            for (int task = 0; task < n; ++task) {
                if (not done[task]) {
                    assig[worker] = task;
                    done[task] = true;
                    if (t + T[worker][task] + bound(worker, task) < best) {
                        recursive(worker+1, t + T[worker][task]);
                    }
                    done[task] = false;
                    assig[worker] = -1;
    }   }   }   }

    double bound(int worker, int task) {
        double f = 0;
        for (int i = worker+1; i < n; ++i) {
            double m = infinity;
            for (int j = 0; j < n; ++j) if (not done[j]) {
                m = min(m, T[i][j]);
            }
            f += m;
        }
        return f;

        Scheduling(time_matrix T) {
            this->T = T;
            n = T.rows();
            assig = vector<int>(n, -1);
            done = vector<boolean>(n, false);
            best = infinity;
            recursive(0, 0);
        }

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

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

Hamiltonian graph problem.

Write an algorithm to determin if a given graph is Hamiltonian.

Backtracking solution. It is assumed that the given graph is connected. It is assumed that the adjacency lists are sorted.

```
#include "eda.hh"
#include <algorithm>

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

class HamiltonianGraph {

    Graph G;            // the graph
    int n;              // number of vertices
    bool found;         // indicates if a cycle has been found
    vector<int> s;      // next of each vertex (−1 if not used)
    vector<int> S;      // solution (if found)

    void recursive(int v, int t) {
        // v = last vertex in the path, t = length of the path
        if (t == n) {
            // we need to check that the cycle can be closed
     if (not G[v].empty() and G[v][0] == 0) {
                s[v] = 0;
                found = true;
                S = s;
                s[v] = -1;
            }
        } else {
            for (int u : G[v]) {
                if (s[u] == -1) {
                    s[v] = u;
                    recursive(u, t+1);
                    s[v] = -1;
                    if (found) return;
    }   }   }   }

public:

    HamiltonianGraph(Graph G) {
        this->G = G;
        n = G.size();
        s = vector<int>(n, -1);
        found = false;
        recursive(0, 1);
    }
        bool has_a_solution() {
            return found;
        }

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

Reads the graph: first the number of vertices; next, for each vertex, its degree and its adjacency list.

```
Graph read_graph() {
    Graph G;
    int n = readint();

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

Main program: reads the graph, creates the solver, and runs it.

```
int main() {
    HamiltonianGraph ham(read_graph());
    if (ham.has_a_solution()) {
        vector<int> s = ham.solution();
        cout << 0 << " ";
        for (int u = s[0]; u != 0; u = s[u]) {
            cout << u << " ";
        }
        cout << endl;
}   }
```

Traveling salesman problem.

A salesman must visit the clients of n different cities. The distance between city i and city j is $D[i][j]$. The salesman wants to leave his own city, visit once and only once each other city, and return to the starting point. His goal is to do that and minimize the total distance of the journey.

```
#include "eda.hh"

typedef matrix<double> distance_matrix;

class TSP {

    distance_matrix M;  // distance matrix
    int n;              // number of cities
    vector<int> s;      // next of each city (−1 if not yet used)
    vector<int> sol;    // best solution so far
    double best;        // cost of best solution so far

    void recursive (int v, int t, double c) {
        // v = last vertex in the path
        // t = length of the path
        // c = cost so far

        if (t == n) {
            c += M[v][0];
            if (c < best) {
                best = 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] < best) {
                    s[v] = u;
                    recursive(u, t+1, c+M[v][u]);
                    s[v] = -1;
    }   }   }   }

 public:

    TSP(distance_matrix M) {
        this->M = M;
        n = M.rows();
        s = vector<int>(n, -1);
        sol = vector<int>(n);
        best = infinity;
        recursive(0, 1, 0);
    }

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

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

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

Main program reads n, creates a distance matrix with randomly placed cities, runs the traveling salesman problem, and writes the cost of the best solution.

```
int main () {
    int n = readint();
    vector<double> x = randvector(n);
    vector<double> y = randvector(n);
    distance_matrix M = distance_matrix(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();
    TSP tsp(M);
    t = now() - t;
    cout << "temps: " << t << endl;
    cout << tsp.cost() << endl;
    cout << tsp.solution() << endl;
}
```

Suposem que un lladre vol entrar en una botiga i carregar al seu sac una combinació d'objectes amb el màxim valor total.

```
class Motxilla {
  int n;                 // nombre d'objectes
  vector<double> p;      // pes de cada objecte
  vector<double> v;      // valor de cada objecte
  double C;              // capacitat de la motxilla
  vector<boolean> s;     // solucio activa
  vector<boolean> sol;// millor solucio provisional
  double millor;         // cost millor solucio provisional
  vector<double> sv;     // suma de 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:
  Motxilla (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; }
};

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;

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

Tresors en un mapa (3)

Feu un programa que, donat un mapa amb tresors i obstacles, digui a quants tresors es pot arribar des d'una posició inicial donada. Els moviments permesos són horitzontals o verticals, però no diagonals. Si cal, es pot passar per sobre dels tresors.

Entrada

L'entrada comença amb el nombre de files $n > 0$ i de columnes $m > 0$ del mapa. Segueixen n files amb m caràcters cadascuna. Un punt indica una posició buida, una 'x' indica un obstacle, i una 't' indica un tresor. Finalment, un parell de nombres f i c indiquen la fila i columna inicials (ambdues començant en 1) des de les quals cal començar a buscar tresors. Podeu suposar que f està entre 1 i n, que c està entre 1 i m, i que la posició inicial sempre està buida.

Sortida

Escriviu el nombre de tresors accessibles des de la posició inicial.

```
#include <iostream>
#include <vector>

using namespace std;


int num_tresors(vector<vector<char> >& M, int f, int c) {
        if(f >= 0 and c >= 0 and f < int(M.size()) and c < int(M[0].size()) and
        M[f][c] != 'X') {
                bool trobat;
                trobat = M[f][c] == 't';
                M[f][c] = 'X'; //mark cell as visited

                int cont = (num_tresors(M, f+1, c) +
                                        num_tresors(M, f-1, c) +
                                        num_tresors(M, f, c+1) +
                                        num_tresors(M, f, c-1));

                return cont + trobat;
        }
        return 0;

}

int main() {
        int f, c;
        cin >> f >> c;

        vector<vector<char> > M(f, vector<char>(c));
        for(int i = 0; i < f; ++i)
                for(int j = 0; j < c; ++j) cin >> M[i][j];

        int forig, corig;
        cin >> forig >> corig;
        cout << num_tresors(M, forig-1, corig-1) << endl;
}
```