DATA STRUCTURES AND ALGORITHMS ALGORITHMS IN C++

Jordi Petit   Salvador Roura   Albert Atserias

15 de febrer de 2017

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

1

STL Usage Examples

1.1 Reading the input line by line: `istringstream`

Reading the input line by line.
Reads a sequence of lines and, for each line, writes the sum of the numbers it contains.

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

int main() {
    string s;
    while (getline(cin, s)) {
        istringstream ss(s);
        int sum = 0;
        int x;
        while (ss >> x) sum += x;
        cout << sum << endl;
}   }
```

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();
}   }
```

1.5 Priority queues with inverted order

(An alternative method is to invert the sign)

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.

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

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

1.6 Sets: `set`

Sets.
Reads two sequences of numbers ended with 0 and writes their intersection.

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

int main() {
    set<int> s1, s2;
    int x;
    cin >> x;
    while (x != 0) {
        s1.insert(x);
        cin >> x;
    }
    cin >> x;
    while (x != 0) {
        s2.insert(x);
        cin >> x;
    }
    for (set<int>::iterator it = s1.begin(); it != s1.end(); ++it) {
        if (s2.find(*it) != s2.end()) cout << *it << endl;
}   }
```

1.7 Dictionaries: `map`

Maps.
Reads a sequence of words and, for each word, in alphabetical order, writes the number of times it appears.

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

int main() {
    map<string, int> m;
    string x;
    while (cin >> x) ++m[x];
    for (map<string, int>::iterator it = m.begin(); it != m.end(); ++it) {
        cout << it->first << " " << it->second << endl;
}   }
```

1.8 New features in C++11

New features of C++11.
In C++11 one can let the compiler ”guess”variable types. Moreover, the loop-construction `for` has been extended to allow easy iterations over collections. It is no longer necessary to add a blank in >> in templates, and it is possible to initialize complex types through lists of initial values.
To compile with C++11, a recent version of GCC is needed (for example gcc-4.7.0). Compilation requires the parameter `-std=c++11`. With slightly older versions of GCC (such as 4.6.2) the required parameter is `-std=c++0x`.

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

int main() {
    // read a vector from input
    vector<int> v;
    int x;
    while (cin >> x) v.push_back(x);
    // write all elements of v to output
    for (int y : v) cout << y << "endl";
    // double all elements of v (note the reference to modify the elements!!!)
    for (int& y : v) y *= 2;
    // write all elements of v again
    for (int y : v) cout << y << endl;

    // make a set of sets of integers and write it
    set<set<int>> S = {{2,3}, {5,1,5}, {}, {3}};
    for (auto s : S) {
        cout << "{";
        for (auto x : s) cout << x << ",";
        cout << "}" << endl;
}   }
```

1.9 Unordered sets: `unordered_set`

Unordered sets.
Reads two sequences of numbers ended with 0 and writes their intersection. This code uses C++11.

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

int main() {
    unordered_set<int> s1, s2;
    int x;
    cin >> x;
    while (x != 0) {
        s1.insert(x);
        cin >> x;
    }
    cin >> x;
    while (x != 0) {
        s2.insert(x);
        cin >> x;
    }
    for (auto x : s1) {
        if (s2.find(x) != s2.end()) cout << x << endl;
}   }
```

1.10 Unordered dictionaries: `unordered_map`

Unordered maps.
Reads a sequence of words and, for each word, prints the number of times it appears. Since an unordered map is used, the order of the output is undefined. This code uses C++11.

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

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

1.11 Definition of hash functions

Definition of hash functions.
In this case the hash function is the sum of the hash functions applied to all three fields. Defining the comparator operator for equality is required. This code uses C++11.

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

struct Point {
    int x, y, z;

    friend bool operator== (const Point& p1, const Point& p2) {
        return p1.x == p2.x and p1.y == p2.y and p1.z == p2.z;
    }

    struct Hash {
        size_t operator() (const Point& p) const {
            return hash<int>()(p.x) + hash<int>()(p.y) + hash<int>()(p.z);
        }
    };
};

int main() {
    unordered_set<Point, Point::Hash> cloud;
    cloud.insert({5,2,3});
}
```

2

Sorting

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;
}
```

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]);
}   }   }   }
```

2.6 Mergesort — 1

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];
}
```

2.7 Mergesort — 2

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);
}   }
```

2.8 Mergesort — 3

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));
}   }   }
```

2.9 Quicksort — 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]);
}   }
```

2.10 Quicksort — 2

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);
}   }
```

2.11 Quicksort — 3

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);
}   }
```

2.12 Heapsort — 1

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();
}   }
```

2.13 Heapsort — 2

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;
}
```

3 Dictionaries

3.1 Tables

Dictionary with a table (unordered).
Here n denotes the size (i.e. number of keys) in the dictionary. Implicit constructor, copy constructor, assignment operator, and destructor do the job; no need to implement them. The worst-case cost of the constructor is $\Theta(1)$, the worst-case cost of the copy constructor, the assignment operator, and the destructor is $\Theta(n)$.

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

template <typename Key, typename Info>
class Dictionary {

private:

    typedef pair<Key, Info> Pair;

    vector<Pair> t;

public:
```

Assigns `info` to `key`. If the key belongs to the dictionary already, the associated information is modified.
Worst-case cost: $\Theta(n)$.

```
    void assign (const Key& key, const Info& info) {
        int i = find(key);
        if (i < t.size()) {
            t[i].second = info;
        } else {
            t.push_back(Pair(key, info));
    }   }
```

Erases `key` and its associated information from the dictionary. If the key does not belong to the dictionary the dictionary does not change.
Worst-case cost: $\Theta(n)$.

```
    void erase (const Key& key) {
        int i = find(key);
        if (i < t.size()) {
            t[i] = t[t.size() - 1];
            t.pop_back();
    }   }
```

Returns a reference to the information associated to `key`. Throws a precondition exception if the key does not belong to the dictionary.
Worst-case cost: $\Theta(n)$.

```
    Info& query (const Key& key) {
        int i = find(key);
        if (i < t.size()) {
            return t[i].second;
        } else {
            throw "Key does not exist.";
    }   }
```

Determines if the dictionary contains `key` .
Worst-case cost: $\Theta(n)$.

```
    bool contains (const Key& key) {
        return find(key) < t.size();
    }
```

Returns the size of the dictionary (i.e. the number of keys that belong to it).
Worst-case cost: $\Theta(1)$.

```
    int size() {
        return t.size();
    }

private:
```

Returns the position of `key` in the table, or `t.size` if it is not there.
Worst-case cost: $\Theta(n)$.

```
    int find (const Key& key) {
        int i = 0;
        while (i < t.size() and t[i].first != key) ++i;
        return i;
    }

};
```

3.2 Lists

Dictionary with a list (unordered).
Assumes that `Key` is a comparable type.
Here n denotes the size (i.e. number of keys) of the dictionary and is stored explicitely to allow an implementation of `size()` with worst-case cost $\Theta(1)$ (note: STL's `size()` for lists takes linear time). Implicit copy constructor, assignment operator and destructor operator do the job; no need to implement them. Their worst-case cost is $\Theta(n)$.

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

template <typename Key, typename Info>
class Dictionary {

private:

    typedef pair<Key, Info> Pair;
    typedef list<Pair> List;
    typedef typename List::iterator iter;

    List li;            // the list
    int n;              // the number of keys

public:
```

Constructor. Creates an empty dictionary.
Worst-case cost: $\Theta(1)$.

```
    Dictionary () {
        n = 0;
    }
```

Assigns `info` to `key`. If the key belongs to the dictionary already, the associated information is modified.
Worst-case cost: $\Theta(n)$.

```
    void assign (const Key& key, const Info& info) {
        iter p = find(key);
        if (p != li.end()) {
            p->second = info;
        } else {
            li.push_back(Pair(key, info));
            ++n;
    }   }
```

Erases `key` and its associated information from the dictionary. If the key does not belong to the dictionary the dictionary does not change.
Worst-case cost: $\Theta(n)$.

```
    void erase (const Key& key) {
        iter p = find(key);
        if (p != li.end()) {
            li.erase(p);
            --n;
    }   }
```

Returns a reference to the information associated to `key`. Throws a precondition exception if the key does not belong to the dictionary.
Worst-case cost: $\Theta(n)$.

```
    Info& query (const Key& key) {
        iter p = find(key);
        if (p != li.end()) {
            return p->second;
        } else {
            throw "Key does not exist";
    }   }
```

Determines if the dictionary contains `key` .
Worst-case cost: $\Theta(n)$.

```
    bool contains (const Key& key) {
        return find(key) != li.end();
    }
```

Returns the size (i.e. number of keys) of the dictionary.
Worst-case cost: $\Theta(1)$.

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

private:
```

Returns the position of c in the list, or li.end() if it is not there.
Worst-case cost: $\Theta(n)$.

```
    iter find (const Key& c) {
        iter p = li.begin();
        while (p != li.end() and p->first != c) ++p;
        return p;
    }

};
```

3.3 Hash Tables

Dictionary with a hash table.

This implementation arranges the elements of the dictionary into M lists, where M is the value of M. The i-th list contains all the pairs key/info $\langle k, x \rangle$ such that hash$(k) = i$. A counter n maintains the number of keys that belong to the dictionary. We use n for the value of n.

The cost analysis is stated under two assumptions: 1) given a key, hash returns a natural number in time $\Theta(1)$, and 2) the hash function *hashes uniformly* at every new call with a different key.

Implicit copy constructor, assignment operator and destructor do the job; no need to implement them. Their worst-case and average-case cost is $\Theta(n + M)$.

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

using namespace std;

template <typename Key, typename Info>
class Dictionary {

private:

    typedef pair<Key, Info> Pair;
    typedef list<Pair> List;
    typedef typename List::iterator iter;

    vector<List> t;     // Hash table
    int n;              // Number of keys
    int M;              // Number of positions

public:
```

Constructor. Creates an empty dictionary.
Worst-case cost: $\Theta(M)$.

```
    Dictionary (int M = 1009)
```

```
:   t(M), n(0), M(M) { }
```

Assigns `info` to `key`. If the key belongs to the dictionary already the associated information is modified.
Worst-case cost: $\Theta(n)$.
Average-case cost: $\Theta(1 + n/M)$.

```
void assign (const Key& key, const Info& info) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end()) {
        p->second = info;
    } else {
        t[h].push_back(Pair(key, info));
        ++n;
}   }
```

Erases `key` and its associated information from the dictionary. If the key does not belong to the dictionary, nothing changes.
Worst-case cost: $\Theta(n)$.
Average-case cost: $\Theta(1 + n/M)$.

```
void erase (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end()) {
        t[h].erase(p);
        --n;
}   }
```

Returns a reference to the information associated to `key`. Throws a precondition exception if the key does not belong to the dictionary.
Worst-case cost: $\Theta(n)$.
Average-case cost: $\Theta(1 + n/M)$.

```
Info& query (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end()) {
        return p->second;
    } else {
        throw "Key does not exist";
}   }
```

Determines if the dictionary contains `key` .
Worst-case cost: $\Theta(n)$.
Average-case cost: $\Theta(1 + n/M)$.

```
bool contains (const Key& key) {
        int h = hash(key) % M;
        iter p = find(key, t[h]);
        return p != t[h].end();
    }
```

Returns the size (i.e. number of keys) of the dictionary.
Worst-case cost: $\Theta(1)$. Average-case cost: $\Theta(1)$.

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

private:
```

Finds `key` in the list and returns an iterator pointing to it, or pointing to `end()` if it is not there.

```
    static iter find (const Key& key, list<Pair>& L) {
        iter p = L.begin();
        while (p != L.end() and p->first != key) ++p;
        return p;
    }

};
```

3.4 Binary Search Trees

Dictionary with a binary search tree (BST).

Assumes that `Key` is a comparable type.

A BST is a binary tree whose nodes store pairs key/info satisfying the property that, for every node u, the key stored at u is bigger than all the keys stored at the left subtree of u and smaller than all the keys stored at the right subtree of u.

This implementation represents the BST through a pointer `root` to a `Node` which contains: a key, the associated information, a pointer to the root of the left subtree, and a pointer to the root of the right subtree. Empty trees are represented by the null pointer. A counter `n` is maintained with the number of keys in the dictionary. We use n for the value of `n`. We use h for the height of the BST and state the costs as a function of h whenever possible.

In a randomly generated BST (where keys are inserted at uniformly chosen positions in the order of the keys and never deleted), we have $h = \Theta(\log n)$ in expectation. Thus, a worst-case cost of $\Theta(h)$ translates to an average-case cost of $\Theta(\log n)$ (under this random model). In the worst-case scenario we have $h = n$.

```
template <typename Key, typename Info>
class Dictionary {

private:

    struct Node {
        Key key;
        Info info;
        Node* left;         // Pointer to left child
        Node* right;        // Pointer to right child

        Node (const Key& k, const Info& i, Node* l, Node* r)
        :   key(k), info(i), left(l), right(r) { }
    };

    int n;              // Number of keys
    Node* root;         // Pointer to the root of the BST

public:
```

Constructor. Creates an empty dictionary.
Worst-case cost: $\Theta(1)$.

```
    Dictionary () {
        n = 0;
        root = nullptr;
    }
```

Copy constructor. Makes a copy.
Worst-case cost: $\Theta(n)$.

```
    Dictionary (const Dictionary& d) {
        n = d.n;
        root = copy(d.root);
    }
```

Assignment operator.
Worst-case cost: $\Theta(n + \mathtt{d}.n)$.

```
    Dictionary& operator= (const Dictionary& d) {
        if (&d != this) {
            free(root);
            n = d.n;
            root = copy(d.root);
        }
        return *this;
    }
```

Destructor.
Worst-case cost: $\Theta(n)$.

```
    ~Dictionary () {
        free(root);
    }
```

Assign info to key.
Worst-case cost: $\Theta(h)$.

```
void assign (const Key& key, const Info& info) {
    assign(root, key, info);
}
```

Erases key and its associated information. If the key does not belong to the dictionary, nothing changes.
Worst-case cost: $\Theta(h)$.

```
void erase (const Key& key) {
    erase_3(root, key);
}
```

Returns a reference to the information associated to key. Throws a precondition exception if the key does not belong to the dictionary.
Worst-case cost: $\Theta(h)$.

```
Info& query (const Key& key) {
    if (Node* p = find(root, key)) {
        return p->info;
    } else {
        throw "Key does not exist";
}   }
```

Determines if the dictionary contains key.
Worst-case cost: $\Theta(h)$.

```
bool contains (const Key& key) {
    return find(root, key);
}
```

Returns the size (i.e. number of keys) of the dictionary.
Worst-case cost: $\Theta(1)$.

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

```
private:
```

Deletes the tree pointed by p.
Worst-case cost: $\Theta(s)$ where s is the number of nodes in the tree pointed by p.

```
static void free (Node* p) {
    if (p) {
        free(p->left);
        free(p->right);
        delete p;
}   }
```

Returns a pointer to a copy of the tree pointed by p.
Worst-case cost: $\Theta(s)$ where s is the number of nodes in the tree pointed by p.

```
static Node* copy (Node* p) {
    return p ? new Node(p->key, p->info, copy(p->left), copy(p->right)) : nullptr;
}
```

Returns a pointer to the node of the tree pointed by p that contains key, or nullptr if the key is not there.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
static Node* find (Node* p, const Key& key) {
    if (p) {
        if (key < p->key) {
            return find(p->left, key);
        } else if (key > p->key) {
            return find(p->right, key);
    }   }
    return p;
}
```

Assigns info to key if the key belongs to the tree pointed by p. Otherwise adds a new node into the tree pointed by p with the key and the associated information.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
void assign (Node*& p, const Key& key, const Info& info) {
    if (p) {
        if (key < p->key) {
            assign(p->left, key, info);
        } else if (key > p->key) {
            assign(p->right, key, info);
        } else {
            p->info = info;
        }
    } else {
        p = new Node(key, info, nullptr, nullptr);
        ++n;
}   }
```

Returns a pointer to the node that contains the minimum key in the tree rooted at p. Assumes that p is not nullptr.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
static Node* minimum (Node* p) {
    return p->left ? minimum(p->left) : p;
}
```

Returns a pointer to the node that contains the maximum key in the tree rooted at p. Assumes that p is not nullptr.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
static Node* maximum (Node* p) {
    return p->right ? maximum(p->right) : p;
}
```

Deletion version 1: Deleting a node with at least one empty child is easy. When both children are non-empty, hangs the left child as the new left child of the minimum of the right child.
Drawback: trees degrade quickly.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
void erase_1 (Node*& p, const Key& key) {
    if (p) {
        if (key < p->key) {
            erase_1(p->left, key);
        } else if (key > p->key) {
            erase_1(p->right, key);
        } else {
            Node* q = p;
            if (!p->left) p = p->right;
            else if (!p->right) p = p->left;
            else {
                Node* m = minimum(p->right);
                m->left = p->left;
                p = p->right;
            }
            delete q; --n;
}   }   }
```

Deletion version 2: Deleting a node with at least one empty child is easy. When both children are non-empty, copies the minimum of the right child to the node that should disappear and proceeds to deleting the minimum of the right child (which does not have left child for sure).
Drawback: copies keys and informations, not pointers.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
void erase_2 (Node*& p, const Key& key) {
    if (p) {
        if (key < p->key) {
            erase_2(p->left, key);
        } else if (key > p->key) {
            erase_2(p->right, key);
        } else if (!p->left) {
            Node* q = p;  p = p->right;
            delete q; --n;
        } else if (!p->right) {
            Node* q = p;  p = p->left;
            delete q; --n;
        } else {
            Node* m = minimum(p->right);
            p->key = m->key; p->info = m->info;
            erase_2(p->right, m->key);
}   }   }
```

Deletion version 3: Deleting a node with at least one empty child is easy. When both children are non-empty, calls `extract_minimum` which extracts the minimum of the right child and returns a pointer to it. This node is placed where the node that should disappear was, which is finally deleted.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
void erase_3 (Node*& p, const Key& key) {
    if (p) {
        if (key < p->key) {
            erase_3(p->left, key);
        } else if (key > p->key) {
            erase_3(p->right, key);
        } else {
            Node* q = p;
            if (!p->left) p = p->right;
            else if (!p->right) p = p->left;
            else {
                Node* m = extract_minimum(p->right);
                m->left = p->left;  m->right = p->right;
                p = m;
            }
            delete q; --n;
}   }   }
```

Extracts the node that contains the minimum element from the tree pointed by p. Returns a pointer to it.
Worst-case cost: $\Theta(t)$ where t is the height of the tree pointed by p.

```
Node* extract_minimum (Node*& p) {
    if (p->left) {
        return extract_minimum(p->left);
    } else {
        Node* q = p;
        p = p->right;
        return q;
}   }

};
```

3.5 AVL Trees

Dictionary with an Adel′son-Vel′skiĭ-Landis tree (AVL).
This implementation contains no documentation yet. Check the documentation for BSTs and your textbook.
The height of the null pointer is -1 and the height of a leaf is 0.

```
template <typename Key, typename Info>
class Dictionary {

private:

    struct Node {
        Key key;
        Info info;
        Node* left;      // Pointer to left child
        Node* right;     // Pointer to right child
        int height;      // Height of the tree

        Node (const Key& c, const Info& i, Node* l, Node* r, int h)
        :   key(c), info(i), left(l), right(r), height(h)
        { }
    };

    int n;           // Number of keys
    Node* root;      // Pointer to the root of the AVL

public:

    Dictionary () {
        n = 0;
        root = nullptr;
    }

    Dictionary (Dictionary& d) {
        n = d.n;
        root = copy(d.root);
    }

    Dictionary& operator= (const Dictionary& d) {
        if (&d != this) {
            free(root);
            n = d.n;
            root = copy(d.root);
        }
        return *this;
    }

    ~Dictionary () {
        free(root);
    }

    void assign (const Key& key, const Info& info) {
        assign(root, key, info);
    }

    void erase (const Key& key) {
        delete_avl(root, key);
    }

    Info& query (const Key& key) {
        if (Node* p=find(root, key)) {
            return p->info;
        } else {
            throw "Key does not exist";
    }   }

    bool contains (const Key& key) {
        return find(root, key);
    }

    int size() {
        return n;
    }

private:

    static void free (Node* p) {
        if (p) {
            free(p->left);
            free(p->right);
            delete p;
    }   }

    static Node* copy (Node* p) {
        return p ? new Node(p->key, p->info, copy(p->left), copy(p->right), p->height)
: nullptr;
    }

    static Node* find (Node* p, const Key& key) {
        if (p) {
            if (key < p->key) {
                return find(p->left, key);
            } else if (key > p->key) {
                return find(p->right, key);
        }   }
        return p;
    }

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

```
}

static void update_height (Node* p) {
    p->height = 1 + max(height(p->left), height(p->right));
}

static void LL (Node*& p) {
    Node* q = p;
    p = p->left;
    q->left = p->right;
    p->right = q;
    update_height(q);
    update_height(p);
}

static void RR (Node*& p) {
    Node* q = p;
    p = p->right;
    q->right = p->left;
    p->left = q;
    update_height(q);
    update_height(p);
}

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

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

void assign (Node*& p, const Key& key, const Info& info) {
    if (p) {
        if (key < p->key) {
            assign(p->left, key, info);
            if (height(p->left)-height(p->right) == 2) {
                if (key < p->left->key) LL(p);
                else LR(p);
            }
            update_height(p);
        } else if (key > p->key) {
            assign(p->right, key, info);
            if (height(p->right)-height(p->left) == 2) {
                if (key > p->right->key) RR(p);
                else RL(p);
            }
            update_height(p);
        } else {
            p->info = info;
        }
    } else {
        p = new Node(key, info, nullptr, nullptr, 0);
        ++n;
}   }

void delete_avl (Node*& p, const Key& key) {
    if (p) {
        if (key < p->key) {
            delete_avl(p->left, key);
            rebalance_left(p);
        } else if (key > p->key) {
            delete_avl(p->right, key);
            rebalance_right(p);
        } else {
            Node* old = p;
            if (p->height == 0) {
                p = 0;
            } else if (!p->left) {
                p = p->right;
            } else if (!p->right) {
                p = p->left;
            } else {
                Node* q = extract_minimum(p->right);
                q->left = p->left;  q->right = p->right;
                p = q;
                rebalance_right(p);
            }
            delete old; --n;
}   }   }

void rebalance_left (Node*& p) {
    if (height(p->right)-height(p->left)==2) {
        if (height(p->right->left) - height(p->right->right) == 1) {
            RL(p);
        } else {
            RR(p);
        }
    } else {
        update_height(p);
}   }

void rebalance_right (Node*& p) {
    if (height(p->left)-height(p->right)==2) {
        if (height(p->left->right) - height(p->left->left) == 1) {
            LR(p);
        } else {
            LL(p);
        }
    } else {
            update_height(p);
    }   }

    Node* extract_minimum (Node*& p) {
        if (p->left) {
            Node* q = extract_minimum(p->left);
            rebalance_left(p);
            return q;
        } else {
            Node* q = p;
            p = p->right;
            return q;
    }   }

};
```

4

Priority Queues

4.1 Recursive Implementation

Priority queue. Recursive implementation.

The priority queue is maintained as a heap in a dynamic table. A heap is a complete binary tree with the property that the value at every node is smaller or equal than the values at the nodes of its subtrees. A heap for n elements is easily implemented in positions 1 to n. An element at position i with $1 \leqslant i \leqslant n$ has its left child at position $2i$ if $2i \leqslant n$, its right child at position $2i+1$ if $2i+1 \leqslant n$, and its parent at position $\lceil i/2 \rceil$ if $i > 1$. For convenience, the table includes a position that is numbered 0 and is not used.

Below, n denotes the size of the priority queue. For the analysis, it is assumed that extending a table by a new element added at the end has constant cost (this is not quite true: it is constant amortized cost).

```
#include "eda.hh"

template <typename Elem>
class PriorityQueue {

private:

    vector<Elem> v;        // Table for the heap (position 0 is not used)

public:
```

Constructor. Creates an empty priority queue.
Cost: $\Theta(1)$.

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

Insert a new element.
Cost: $\Theta(\log n)$.

```
void insert (const Elem& x) {
    v.push_back(x);
    shift_up(size());
}
```

Remove and return the minimum elemenv.
Cost: $\Theta(\log n)$.

```
Elem remove_min () {
    if (empty()) throw "Priority queue is empty";
    Elem x = v[1];
    v[1] = v.back();
    v.pop_back();
    shift_down(1);
    return x;
}
```

Returns the minimum elemenv.
Cost: $\Theta(1)$.

```
Elem minimum () {
    if (empty()) throw "Priority queue is empty";
    return v[1];
}
```

Returns the size of the priority queue.
Cost: $\Theta(1)$.

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

Indicates if the priority queue is empty.
Cost: $\Theta(1)$.

```
bool empty () {
    return size() == 0;
}
```

```
private:
```

Shift a node up in the tree, as long as needed.

```
void shift_up (int i) {
        if (i != 1 and v[i/2] > v[i]) {
            swap(v[i], v[i/2]);
            shift_up(i/2);
    }   }
```

Shift a node down in the tree, as long as needed.

```
    void shift_down (int i) {
        int n = size();
        int c = 2*i;
        if (c <= n) {
            if (c+1 <= n and v[c+1] < v[c]) c++;
            if (v[i] > v[c]) {
                swap(v[i],v[c]);
                shift_down(c);
    }   }   }

};
```

4.2 Iterative Implementation

Priority queue. Iterative implementation.
Same as earlier version but with iterative implementation.

```
#include "eda.hh"

template <typename Elem>
class PriorityQueue {

private:

    vector<Elem> v;          // Vector for the heap (position 0 is not used)

public:
```

Constructor. Creates an empty priority queue.
Cost: $\Theta(1)$.

```
    PriorityQueue () {
        v.push_back(Elem());
    }
```

Inserts a new element.
Cost: $\Theta(\log n)$.

```
    void insert (const Elem& x) {
```

```
    v.push_back(x);
    int i = size();
    while (i != 1 and v[i/2] > x) {
        v[i] = v[i/2];
        i = i/2;
    }
    v[i] = x;
}
```

Removes and returns the minimum element.
Cost en el cas pitjor: $\Theta(\log n)$.

```
Elem remove_min () {
    if (empty()) throw "Priority queue is empty";
    int n = size();
    Elem e = v[1],  x = v[n];
    v.pop_back();  --n;
    int i = 1,  c = 2*i;
    while (c <= n) {
        if (c+1 <= n and v[c+1] < v[c]) ++c;
        if (x <= v[c]) break;
        v[i] = v[c];
        i = c;
        c = 2*i;
    }
    v[i] = x;
    return e;
}
```

Returns the minimum.
Cost: $\Theta(1)$.

```
Elem minimum () {
    if (empty()) throw "Priority queue is empty";
    return v[1];
}
```

Returns the size of the priority queue.
Cost: $\Theta(1)$.

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

Indicates if the queue is empty.
Cost: $\Theta(1)$.

```
bool empty () {
        return size() == 0;
    }

};
```

5

Graphs

5.1 Type Definitions for Graphs

Type definition for graphs.

Graphs are represented by a table of adjacency lists, assuming that the vertices are the integers between 0 and $|V| - 1$. These graphs are directed, but the type can also be used to represent undirected graphs by representing each edge as two arcs. The adjacency lists are implemented by dynamic vectors.

This code uses C++11.

```
#ifndef graf_hh
#define graf_hh

#include <vector>
#include <list>

using namespace std;

typedef vector<vector<int>> graph;

#endif
```

5.2 Depth First Search

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>
```

```
#include "eda.hh"
#include "graph.hh"
```

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;
}
```

5.3 Breadth First Search

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.
This code uses C++11.

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

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;
}
```

5.4 Topological Sort

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: $\Theta(|V| + |E|)$.
This code uses C++11.

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

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;
}
```

5.5 Shortest Paths: Dijkstra's Algorithm

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 $\Theta((|V| + |E|)\log(|V|))$.

This code uses C++11.

```
#include "eda.hh"

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}});
}   }   }   }
```

6

Backtracking

6.1 n Queens — 1

***n*-queens problem.**
Write an algorithm that writes all possible ways of placing n queens on an $n \times n$ chessboard in such a way that no queen threatens another.
First (naive) version.

```
#include "eda.hh"

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 ? "Q " : "* ") ;
            }
            cout << endl;
        }
        cout << endl;
    }

public:

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

Main program

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

6.2 *n* Queens — 2

***n*-queens problem.**
Write an algorithm that writes all possible ways of placing n queens in an $n \times n$ chessboard in such a way that no queens threatens another.
This is a less naive solution.

```
#include "eda.hh"

class NQueens {

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

    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) {
            write();
        } 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;
                    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);
        recursive(0);
    }
};
```

Main program

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

6.3 *n* Queens — 3

***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);
    }
};
```

Main program

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

6.4 Latin Squares

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);
    }
};
```

Main program

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

6.5 Knight Jumps

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;
}
```

6.6 Task Scheduling

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;
    }

public:

    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;
    }
};
```

Main program reads *n*, creates the time matrix, the solver, runs it, and writes the solution.

```
int main () {
    int n = readint();
    time_matrix M = randmatrix(n);
    cout << M;
    Scheduling tasks(M);
    cout << tasks.cost() << endl;
    cout << tasks.solution() << endl;
}
```

6.7 Hamiltonian Graph

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;
}   }
```

6.8 Traveling Salesman Problem

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;
}
```

6.9 Knapsack — 1

Knapsack problem.
We have got a knapsack that can hold up to C units of weight and up to n objects. The i-th object has weight $p[i]$ and value $v[i]$. Our goal is to choose which objects to place in the knapsack in such a way that the sum of their values is maximized subject to the constraint that the sum of their weights does not exceed C. Objects cannot be split.
Backtracking solution without lower bound.

```
#include "eda.hh"

class Knapsack {

    int n;                      // number of objects
    vector<double> p;           // weights
    vector<double> v;           // values
    double C;                   // weight capacity
    vector<boolean> s;          // current solution
    vector<boolean> sol;        // best solution so far
    double best;                // value of the best solution so far

    void recursive(int i, double val, double pes) {
        // i = currently handled object
        // val = accumulated value
        // pes = accumulated weight
        if (i == n) {
            if (val > best) {
                best = val;
                sol = s;
            }
        } else {
            // 1st option: take object i
            if (pes+p[i] <= C) {
                s[i] = true;
                recursive(i+1, val+v[i], pes+p[i]);
            }
            // 2st option: do not take object i
            s[i] = false;
            recursive(i+1, val, pes);
    }   }

public:

    Knapsack(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);
        best  = 0;
        recursive(0, 0, 0);
    }

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

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

Main program: reads the number of objects, sets the weights and values at random, sets the capacity to 0.4 the number of objects, creates the solver, runs it, and writes the solution.

```
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;

    Knapsack k(n, p, v, C);
    cout << k.value() << endl;
    cout << k.solution() << endl;
}
```

6.10 Knapsack — 2

Knapsack problem.

We have got a knapsack that can hold up to C units of weight and up to n objects. The i-th object has weight $p[i]$ and value $v[i]$. Our goal is to choose which objects to place in the knapsack in such a way that the sum of their values is maximized subject to the constraint that the sum of their weights does not exceed C. Objects cannot be split.

Backtracking solution with lower bound: we take into account the maximum value contribution that the remaining objects could bring (even if they exceed the capacity).

A clever sorting of the objects might improve the algorithm.

```
#include "eda.hh"

class Knapsack {

    int n;                      // number of objects
    vector<double> p;           // weights
    vector<double> v;           // values
    double C;                   // capacity
    vector<boolean> s;          // current solution
    vector<boolean> sol;        // best solution so far
    double best;                // value of the best solution so far
    vector<double> sv;          // sum of values for the lower bound

    void recursive(int i, double val, double pes) {
        // i = currently handled object
        // val = accumulated value
        // pes = accumulated weight
        if (i == n) {
            if (val > best) {
                best = val;
                sol = s;
            }
        } else {
            // 1st option: take object i
            if (pes+p[i] <= C and val+sv[i] > best) {
                s[i] = true;
                recursive(i+1, val+v[i], pes+p[i]);
            }
            // 2nd option: do not take object i
            if (val+sv[i+1] > best) {
                s[i] = false;
                recursive(i+1, val, pes);
    }   }   }

public:

    Knapsack(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);
        best    = 0;
        sv      = vector<double>(n+1);
        sv[n] = 0;
        for (int i = n-1; i >= 0; --i) {
            sv[i] = sv[i+1]+v[i];
        }
        recursive(0, 0, 0);
    }

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

    double value() {
        return best;
    }

};
```

Main program: reads the number of objects, sets the weights and values at random, sets the capacity to 0.4 the number of objects, creates the solver, runs it, and writes the solution.

```
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;

    Knapsack k(n,p,v,C);
    cout << k.value() << endl;
    cout << k.solution() << endl;
}
```

A

Macros

A.1 eda.hh

This is non-polished code.

Data Structures and Algorithms.
Jordi Petit Salvador Roura July 2010 Translated by AA in September 2013

```
#ifndef eda_hh
#define eda_hh
```

Standard inclusions

```
#include <vector>
#include <list>
#include <set>
#include <map>
#include <queue>
#include <iostream>
#include <limits>
#include <cstdlib>
#include <cassert>
#include <cmath>
#include <sys/resource.h>

using namespace std;
```

Definition of *boolean* type. The *boolean* type is an *int*, but it is used as if it were a *bool*. The purpose of this definition is to make vectors of *boolean*s that are fast. A *vector¡bool¿* is slow because it works at the bit level.

```
typedef int boolean;
```

Definition of the null pointer *null*. In C++ a null pointer is simply a zero.

```
#define null 0
```

Definition of the generic class *matrix*. Useful to make bidimensional tables without using vectors of vectors. The downside is that the numbers of rows and columns are fixed.

```
template <typename T>
class matrix {

    int r,c;
    vector<vector<T> > t;

public:

    matrix() :
        r(0), c(0) {  }

    matrix(int rows, int cols, T init=T()) :
        r(rows), c(cols), t(vector<vector<T> >(r, vector<T>(c, init))) {  }

    int rows() const {
        return r;
    }

    int cols() const {
        return c;
    }

    vector<T>& operator[](int i) {
        return t[i];
    }

    const vector<T>& operator[](int i) const {
        return t[i];
    }

    friend ostream& operator<<(ostream& s, matrix<T> m) {
        for (int j = 0; j < m.c; ++j) {
            for (int i = 0; i < m.r; ++i) {
                s << m.t[i][j] << " ";
            }
            s << endl;
        }
        return s;
    }
};
```

Operator to write the contents of a *vector*.

```
template <typename T> ostream& operator<<(ostream& s, vector<T> v) {
    for (int i = 0; i < int(v.size()); ++i) {
        s << v[i] << " ";
    }
    return s;
}
```

Functions to read basic types.

```
inline int      readint     ()  { int    n;  cin >> n;  return n; }
inline char     readchar    ()  { char   n;  cin >> n;  return n; }
inline bool     readbool    ()  { bool   n;  cin >> n;  return n; }
inline double   readdouble  ()  { double n;  cin >> n;  return n; }
```

Defines *infinity* as a double that is bigger than any other double (except, obviously *infinity*).

```
const double infinity = numeric_limits<double>::infinity();
```

Defines *maxint* as the largest representable integer.

```
const int maxint = numeric_limits<int>::max();
```

Functions to generate random numbers.

```
// Returns a random real in [0,1).
inline double randdouble() {
    return rand() / double(RAND_MAX);
}

// Returns a random integer in [a..b].
inline int randint(int a, int b) {
    return a + rand() % (b - a + 1);
}

// Returns a random integer in [0..n − 1].
inline double randint(int n) {
    return rand() % n;
}

// Returns true with probability p.
inline boolean randbit(double p) {
    return randdouble() < p;
}

// Returns an n × n matrix with random reals.
```

```
matrix<double> randmatrix(int n) {
    matrix<double> m(n, n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; j++) {
            m[i][j] = randdouble();
    }   }
    return m;
}

// Returns a vector of n random reals.
vector<double> randvector(int n) {
    vector<double> v(n);
    for (int i = 0; i < n; ++i) {
        v[i] = randdouble();
    }
    return v;
}

// Returns a vector with n random integers in [a..b].
vector<int> randvector(int n, int a, int b) {
    vector<int> v(n);
    for (int i = 0; i < n; ++i) {
        v[i] = randint(a, b);
    }
    return v;
}
```

Functions to measure CPU time.

```
inline double now() {
    // This works in Linux, I do not know for other systems.
    struct rusage u;
    getrusage(RUSAGE_SELF, &u);
    return
        u.ru_utime.tv_sec  + u.ru_stime.tv_sec
      +(u.ru_utime.tv_usec + u.ru_stime.tv_usec)/1000000.0;
}
```

Macro to iterate over all the elements in a container.

```
#define foreach(it,c) for (__typeof((c).begin()) it=(c).begin(); it!=(c).end(); ++it)
```

ErrorPrec and *ErrorImpl* are thrown exceptions. *Error* is a common basic class, useful to *catch* both. This is the only place where inheritance is used in the course.

```
class Error {
    private:
        string err;
    public:
        Error(string s) : err(s) {}
        string error() const {return err;}
};

class ErrorPrec: public Error {
    public:
        ErrorPrec(string s) : Error("Precondition error: " + s) {}
};

class ErrorImpl: public Error {
    public:
        ErrorImpl(string s) : Error("Implementation error: " + s) {}
};

#endif
```