

## Analysis of Algorithms

Complexity of an algorithm = computational resources it consumes: execution time, memory space.

Analysis of algorithms → Investigate the properties of the complexity of algorithms.

Rate of Growth											
$\lg n$	$<$	$\sqrt{n}$	$<$	$n$	$<$	$n \lg n$	$<$	$n^2$	$<$	$n^3$	$\theta(1)$
Logarithmic		linear		quadratic		square		cubic		constant	

$O(n)$  → Asymptotic upper bound (Big-Oh) →  $f$  grow no faster than  $O(n)$

$\Omega(n)$  → Asymptotic lower bound →  $f$  grow at least as fast as  $\Omega(n)$

$\Theta(n)$  → Asymptotically tight bound →  $f$  grow with the same rate of  $\Theta(n)$

Properties:

- If  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < +\infty \rightarrow g = O(f) \rightarrow$  To prove that an algorithm cost (Ex:  $f(n) = \Theta(n)$ )
- For all positive constants  $c > 0$ ,  $O(f) = O(c \cdot f)$
- Changing the basis of a logarithm  $\log_c x = \frac{\log_b x}{\log_b c}$
- Rule of sums:  $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- Rule of products:  $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$
- Pass a parameter by value →  $\Theta(n)$  Because the parameter is copied
- Pass a parameter by reference (&) →  $\Theta(1)$

Asymptotic because it matters for only large values of  $n$  and describe the limit of a function.

### Analysis of iterative algorithms

The cost of an elementary operation is  $\Theta(1)$

<code>if B then S1 else S2</code>	worst case → $O(\max\{f + h, g + h\})$	The cost of B is h The cost of S1 is f The cost of S2 is g
---------------------------------------	--	--

<code>while B {     S }</code>	$T(n) = \sum_{i=1}^g f(n) + h(n)$ If $p = \max\{f + h\} \rightarrow T = O(p \cdot g)$	The cost of B is f The cost of S is h $g = \text{number of iterations}$
--	--	---

### Analysis of recursive algorithm

$$T(n) \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \text{ (base case)} \\ a \cdot T(n - c) + f(n) & \text{if } n \geq n_0 \text{ (recursive case)} \end{cases} \rightarrow T(n) = \begin{cases} \Theta(n^k) & \text{If } a < 1 \\ \Theta(n^{k+1}) & \text{If } a = 1 \\ \Theta\left(a^{\frac{n}{c}}\right) & \text{If } a > 1 \end{cases}$$

Where  $c \geq 1$ ,  $g(n)$  is the cost of base case,  $a$  is the number of recursive calls and  $f(n) = \Theta(n^k)$  is the cost of the non-recursive part of the algorithm.

Examples: if  $f(n) = \Theta(1) \rightarrow n^k = 1 \rightarrow k = 0$ , if  $f(n) = \Theta(n) \rightarrow n^k = n \rightarrow k = 1$

$$T(n) \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \text{ (base case)} \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & \text{if } n \geq n_0 \text{ (recursive case)} \end{cases} \rightarrow T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{If } k < \log_b a \\ \Theta(n^k \cdot \log n) & \text{If } k = \log_b a \\ \Theta(n^k) & \text{If } k > \log_b a \end{cases}$$

Where  $a \geq 1$ ,  $b > 1$ ,  $g(n)$  is the cost of base case,  $a$  is the number of recursive calls and  $f(n) = \Theta(n^k)$  is the cost of the non-recursive part of the algorithm.

## Divide and conquer

If a given input is simple enough → find a simple solution.

Otherwise the given input in subproblems and solve each one independently and recursively and finally combine the solutions corresponding to the original input.

```

procedure DIVIDE_AND_CONQUER(x) {
    if x is simple return DIRECT_SOLUTION(x)
    else {
        ( $x_1, x_2, \dots, x_k$ ) := DIVIDE(x)
        for i := 1 to k {
             $y_i := \text{DIVIDE\_AND\_CONQUER}(x_i)$ 
        }
        return COMBINE( $y_1, y_2, \dots, y_k$ )
    }
}

```

### Cost of Divide and conquer algorithms

$$T(n) = \begin{cases} g(n) & \text{if } n \leq n_0 \text{ (base case)} \\ f(n) + a \cdot T\left(\frac{n}{b}\right) & \text{if } n > n_0 \text{ (recursive case)} \end{cases} \rightarrow T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{If } k < \log_b a \\ \Theta(f(n) \cdot \log n) & \text{If } k = \log_b a \\ \Theta(f(n)) & \text{If } k > \log_b a \end{cases}$$

Where  $a \geq 1$ ,  $b > 1$ ,  $g(n)$  is the cost of base case,  $a$  is the number of recursive calls and  $f(n) = \Theta(n^k)$  is the cost of the non-recursive part of the algorithm.

The cost of the sums and shifts (to multiply by  $2^k$ ) is  $\Theta(n)$

Matrix additions/subtractions have cost  $\Theta(n^2)$

**Karatsuba's Algorithm** → computes the product of two natural numbers of  $n$  bits

$$\rightarrow M(n) = \Theta(n^{\log_2 3})$$

$$x = \begin{array}{|c|c|} \hline 01\dots & 11\dots \\ \hline A & B \\ \hline \end{array} \quad y = \begin{array}{|c|c|} \hline 10\dots & 10\dots \\ \hline C & D \\ \hline \end{array}$$

1.  $x$  and  $y$  pass from decimal to binary
2. Split  $x$  and  $y$  in half, if  $x$  and  $y$  doesn't have the same numbers of bits ( $n$ ) or  $n$  is not a power of 2 → add 0's.
3. Calculate  $A \cdot C \cdot 2^n + [(A + B) \cdot (C + D) - (A \cdot C + B \cdot D)] \cdot 2^{\frac{n}{2}} + B \cdot D$

```

// Pre: x = X ^ y = Y
z = 0;
while (x != 0)
    // Inv: z + x - y = X - Y
    if (x % 2 == 0) { x /= 2; y *= 2; }
    else { x--; z += y; }
// Post: z = X - Y

```

**Strassen's Algorithm** → computes the product of two matrices of size  $n \times n \rightarrow M(n) = \Theta(n^{\log_2 7})$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} = M_2 + M_3 & C_{12} = M_1 + M_2 + M_5 + M_6 \\ C_{21} = M_1 + M_2 + M_4 - M_7 & C_{22} = M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

$$\begin{aligned}
 M_1 &= (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12}) \\
 M_2 &= A_{11} \cdot B_{11} \\
 M_3 &= A_{12} \cdot B_{21} \\
 M_4 &= (A_{11} - A_{21}) \cdot (B_{22} - B_{12}) \\
 M_5 &= (A_{21} + A_{22}) \cdot (B_{12} - B_{11}) \\
 M_6 &= (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22} \\
 M_7 &= A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})
 \end{aligned}$$

```

for i := 1 to n do
    for j := 1 to n do
        C[i; j] := 0
        for k := 1 to n do
            C[i; j] := C[i; j] + A[i; k] * B[k; j]
        end for
    end for
end for

```

**BinarySearch** → Return the position of a number in an array in increasing order →  $\Theta(\log n)$

1. Get the number of the middle position of all the array
2. Check if the number of the step 1 is bigger or lower than the number we are searching
  - a. If the number is bigger it will take the half upwards positions
  - b. If the number is lower it will take the half downwards positions
3. Repeat the same process until find the number that we are searching. If the number that we are looking for it's not in the array, return the closest above number.

**MergeSort** best case, average case and worst case →  $\Theta(n \log n)$

1. Divide the input sequence in two halves.
2. Sort each half recursively
3. Merge the two sorted sequences

**QuickSort** best case and worst case →  $\Theta(n \log n)$  worst case →  $\Theta(n^2)$

1. Take an element of the array and use it as a pivot
2. The elements bigger than the pivot go in one side and the lowers in the other side, the elements equal to the pivot can go wherever we want.
3. Divide the list in two (Biggers and lowers).
4. Repeat the process for all sub lists.
5. Combine

**QuickSelect** → Find the  $j - th$  smallest element in a given set not ordered average case →  $\Theta(n)$ , worst case →  $O(n^2)$

1. Take the middle position element in the set as a pivot (The first time is the first element).
2. All the elements bigger than pivot go to the right side of the set.
3. Take other element of the left side of the set as a pivot.
4. Repeat the process while pivot  $k > j$ .

**Rivest-Floyd's Worst-Case Linear Time Selection** →  $\Theta(n)$

1. Divide the set in  $q$  blocks → for each block take the median and put it in block of  $q$  elements.
2. Obtain the median recursively of the  $n/q$  medians of the previous step.
3. The median of medians (pseudo-median) is used as a pivot for partitioning the array
4. Do the same as quickSelect but use this pivot.

### Binary Search

```
// Initial call: int p = bsearch(A, x, -1, A.size());
template <class T>
int bsearch(const vector<T>& A, const T& x, int l,
int u) {
    if (l == u + 1)
        return u;
    int m = (l + u) / 2;
    if (x < A[m])
        return bsearch(A, x, l, m);
    else
        return bsearch(A, x, m, u);
}
```

### QuickSelect

```
procedure QUICKSELECT(A, l, j, u)
Ensure: Returns the  $(j + 1 - l)$ -th smallest element in
A[l..u],
 $l \leq j \leq u$ 
if  $l = u$  then
    return A[l]
end if
PARTITION(A, l, u, k)
if  $j = k$  then
    return A[k]
end if
if  $j < k$  then
    return QUICKSELECT(A; l; j; k - 1)
else
    return QUICKSELECT(A; k + 1; j; u)
end if
end procedure
```

## MergeSort

```

struct node_list {
    ELEM info;
    node_list* next;
};

typedef node_list* List;
void split(List& L, List& L2, int n) {
    node_list* p = L;
    while (n > 1) {
        p = p -> next;
        --n;
    }
    L2 = p -> next; p -> next = nullptr;
}

List merge(List L, List L2) {
    if (L == nullptr) return L2;
    if (L2 == nullptr) return L;
    if (L -> info <= L2 -> info) {
        L -> next = merge(L -> next, L2);
        return L;
    } else { // L -> info > L2 -> info
        L2 -> next = merge(L, L2 -> next);
        return L2;
    }
}
void mergesort(List& L, int n) {
    if (n > 1) {
        int m = n / 2;
        List L2;
        split(L, L2, m);
        mergesort(L, m);
        mergesort(L2, n-m);
        L = merge(L, L2);
    }
}

```

## QuickSort

```

template <class T>
void partition(vector<T>& v, int l, int u, int& k) {
    int i = l;
    int j = u + 1;
    T pv = v[i]; // simple choice for pivot
    do {
        while (v[i] < pv and i < u) ++i;
        while (pv < v[j] and l < j) --j;
        if (i <= j) {
            swap(v[i], v[j]);
            ++i; --j;
        }
    } while (i <= j);
    swap(v[l], v[j]);
    k = j;
}

template <class T>
void quicksort(vector<T>& v, int l, int u) {
    if (u - l + 1 > M) {
        int k;
        partition(v, l, u, k);
        quicksort(v, l, k-1);
        quicksort(v, k+1, u);
    }
}

template <class T>
void quicksort(vector<T>& v) {
    quicksort(v, 0, v.size()-1);
    insertion_sort(v, 0, v.size()-1);
}

```

**Insertion sort** best case  $\rightarrow \Theta(n)$ , worst case  $\rightarrow \Theta(n^2)$

## Dictionaries

BST (Binary Search Tree)  $\rightarrow T$  is a binary tree that is either empty, or it contains at least one element  $x$  at its root, and

- 1- The left and right subtrees of  $T$ ,  $L$  and  $R$ , respectively, are binary search trees.
- 2- For all elements  $y$  in  $L$ ,  $KEY(y) < KEY(x)$  and for all elements  $z$  in  $R$ ,  $KEY(z) > KEY(x)$ .

### Search

- If  $k = KEY(x) \rightarrow$  The search is successful.
- If  $k < KEY(x) \rightarrow$  If there exist an element in  $T$  with key  $k$  it must be stored in the left subtree of  $T \rightarrow$  We must make a recursive call on the left subtree of  $T$  to continue the search.
- If  $k > KEY(x) \rightarrow$  The search must recursively continue in the right subtree.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_lookup(node_bst* p, const Key& k) {
    if (p == nullptr or k == p->_k) return p;
    // p != nullptr and k != p->_k
    if (k < p->_k) return bst_lookup(p->_left, k);
    else return bst_lookup(p->_right, k); // p->_k < k
}
```

### Insertion

- If  $k = KEY(x) \rightarrow$  Overwrite value.
- If  $k < KEY(x) \rightarrow$  Insert in the left subtree.
- If  $k > KEY(x) \rightarrow$  Insert in the right subtree.

```
template <typename Key, typename Value>
void Dictionary::insert(const Key& k, const Value& v) {
    root = bst_insert(root, k, v);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_insert(node_bst* p, const Key& k, const Value& v) {
    if (p == nullptr) return new node_bst(k, v);
    // p != nullptr, continue the insertion in the appropriate subtree or update
    // the associated value if p->_k == k
    if (k < p->_k) p->_left = bst_insert(p->_left, k, v);
    if (p->_k < k) p->_right = bst_insert(p->_right, k, v);
    else p->_v = v; // p->_k == k
    return p;
}
```

### Deletions

- If the node that we want to delete is a leaf (both subtrees are empty)  $\rightarrow$  Remove the node.
- If the node have only one non-empty subtree  $\rightarrow$  Replace the non-empty subtree by father.
- If the node have both subtrees  $\rightarrow$ 
  - 1- Return pointer of the largest key in left subtree (inside this search in both subtrees).
  - 2- Replace this pointer by father, and join the left and right subtrees of the father.

```

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::join(node_bst* t1, node_bst* t2) {
    // trivial if one or two of the trees are empty
    if (t1 == nullptr) return t2;
    if (t2 == nullptr) return t1;

    t1 != nullptr and t2 != nullptr
    node_bst* z = relocate_max(t1);
    z -> _right = t2;
    return z;

    // alternative: z = relocate_min(t2);
    // z -> _left = t1;
    // return z;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::relocate_max(node_bst* p) {
    node_bst* f = nullptr;
    node_bst* orig_p = p;
    while (p -> _right != nullptr) {
        f = p;
        p = p -> _right;
    }
    if (f != nullptr) {
        f -> _right = p -> _left;
        p -> _left = orig_p;
    }
    return p;
}

template <typename Key, typename Value>
void Dictionary<Key,Value>::remove(const Key& k) {
    root = bst_remove(root, k);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_remove(node_bst* p, const Key& k) {
    // key k is not in the tree if it is empty
    if (p == nullptr) return p;

    if (k < p -> _k) p -> _left = bst_remove(p -> _left, k);
    else if (p -> k < k) p -> _right = bst_remove(p -> _right, k);
    else { // k == p -> k
        node_bst* to_kill = p;
        p = join(p -> _left, p -> _right);
        delete to_kill;
    }
    return p;
}

```

## Cost

The cost of the operations depends on the height of the tree

	Worst-case	Average-case
Size $n$ and height $h$	$h = \Theta(n)$	$h = \Theta(\log n)$

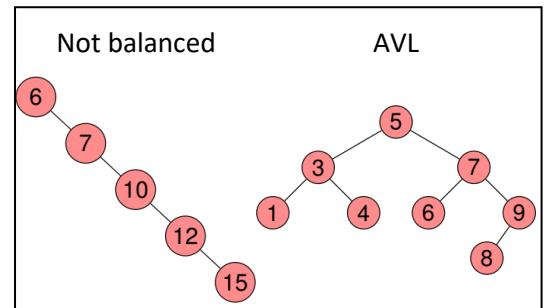
**Operation: In\_range** → Return a range between two keys → The cost is  $\Theta(\log n + F)$  where  $F$  is the length of the result.

## AVLs

An AVL  $T$  is a binary search tree that is either empty or

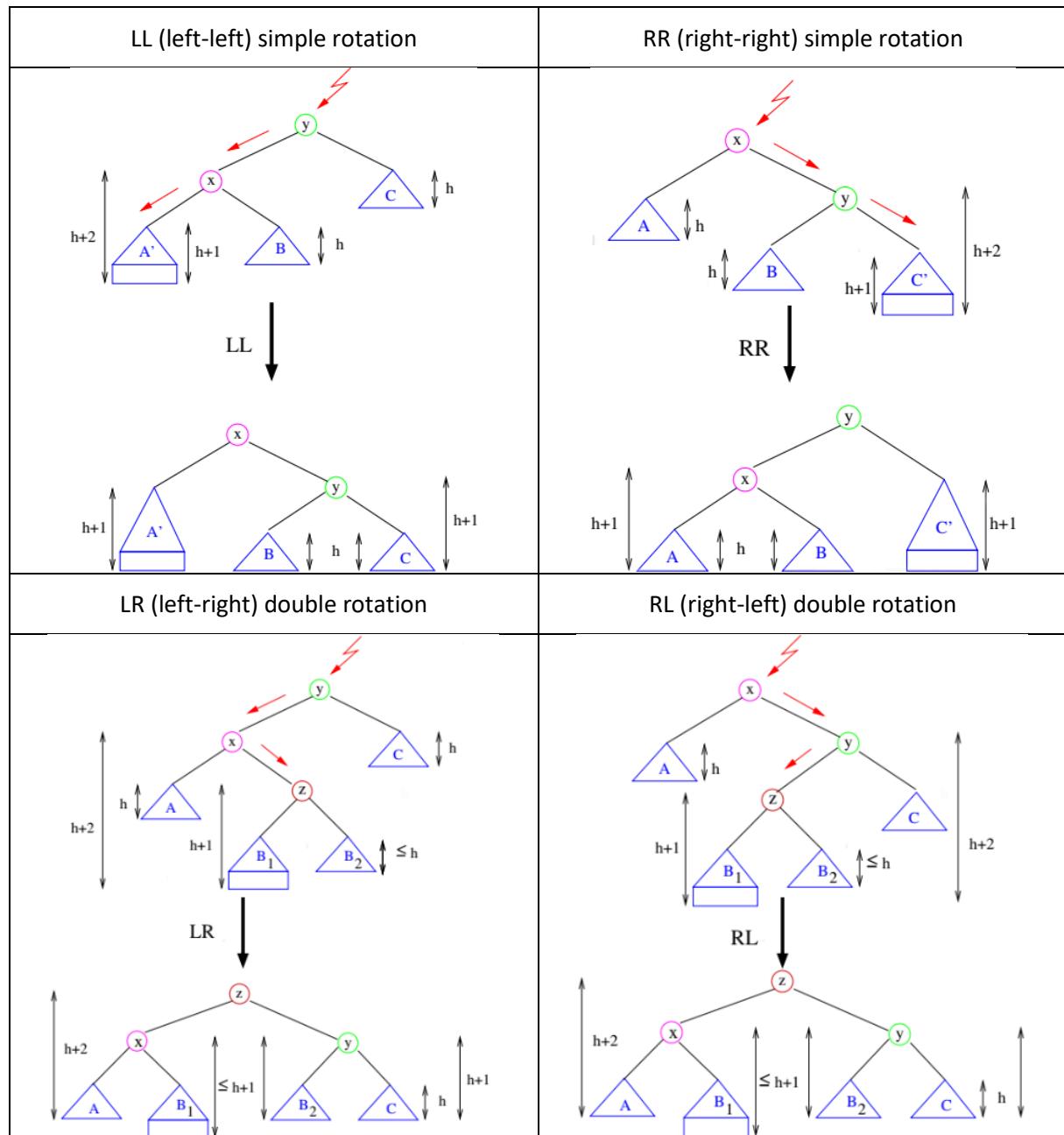
- 1- Its left and right subtrees,  $L$  and  $R$ , respectively are AVLs.
- 2- The height of  $L$  and  $R$  differs by one at most  
 $|height(L) - height(R)| \leq 1$ .

**Cost of search** → The height  $h(T)$  of an AVL  $T$  of size  $n$  is  
 $\Theta(\log n)$  → The worst case is  $\Theta(\log n)$ .



**Update AVLs** → The insertion and deletions operations act as their counterparts for BSTs, but after, we check if the tree is balanced (respect the rule of height of AVLs). To re-establish the balance invariant in a node where it didn't hold we will use **rotations**.

- 1- Update the `_height` →  $\Theta(1)$ , independent of tree's size.
- 2- We will need to perform a rotation at most to re-establish the AVL condition with cost  $\Theta(1)$ .
- 3- The worst-case cost of an insertion or deletion in an AVL is  $\Theta(\log n)$ .



```

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_LL(node_avl* p) {
    node_avl* q = p -> _left;
    p -> _left = q -> _right;
    q -> _right = p;
    update_height(p);
    update_height(q);
    return q;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_RR(node_avl* p) {
    node_avl* q = p -> _right;
    p -> _right = q -> _left;
    q -> _left = p;
    update_height(p);
    update_height(q);
    return q;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_LR(node_avl* p) {
    p -> _left = rotate_RR(p -> _left);
    return rotate_LL(p);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_RL(node_avl* p) {
    p -> _right = rotate_LL(p -> _right);
    return rotate_RR(p);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::avl_insert(node_avl* p, const Key& k, const Value& v) {
    if (p == nullptr) return new node_avl(k, v);

    if (k < p -> _k) {
        p -> _left = avl_insert(p -> _left, k, v);
        // chec balance at p and rotate if needed
        if (height(p -> _left) - height(p -> _right) == 2) {
            // p -> _left cannot be empty
            if (k < p -> _left -> _k) p = rotate_LL(p); // LL
            else p = rotate_LR(p); // LR
        }
    }

    else if (p -> _k < k) { // symmetric case
        p -> _right = avl_insert(p -> _right, k, v);
        if (height(p -> _right) - height(p -> _left) == 2) {
            if (p -> _right -> _k < k) p = rotate_RR(p); // RR
            else p = rotate_RL(p); // RL
        }
    }

    else p -> _v = v; // p -> _k == k
    update_height(p);
    return p;
}

```

```

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::avl_remove(node_avl* p, const Key& k) {
    if (p == nullptr) return p;
    if (k < p->_k) {
        p->_left = avl_remove(p->_left, k);
        // check balance and rotate if needed
        if (height(p->_right) - height(p->_left) == 2) {
            // p->_right cannot be empty
            if (height(p->_right->_left) - height(p->_right->_right) == 1)
                p = rotate_RL(p);
            else p = rotate_RR(p);
        }
    }
    else if (p->_k < k) { // symmetric case
        p->_right = avl_remove(p->_right, k);
        if (height(p->_left) - height(p->_right) == 2) {
            if (height(p->_left->_right) - height(p->_right->_left) == 1)
                p = rotate_LR(p);
            else p = rotate_LL(p);
        }
    }
    else { // p->_k == k
        node_avl* to_kill = p;
        p = join(p->_left, p->_right);
        delete to_kill;
    }
    update_height(p);
    return p;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::join(node_avl* t1, node_avl* t2) {
    // trivial, if one of the trees is empty
    if (t1 == nullptr) return t2;
    if (t2 == nullptr) return t1;
    // t1 != nullptr and t2 != nullptr
    node_avl* z;
    remove_min(t2, z);
    z->_left = t1;
    z->_right = t2;
    update_height(z);
    return z;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::remove_min( node_avl*& p, node_avl*& z) {
    if (p->_left != nullptr) {
        remove_min(p->_left, z);
        // check balance and rotate if needed
        if (height(p->_right) - height(p->_left) == 2) {
            // p->_right cannot be empty
            if (height(p->_right->_left) - height(p->_right->_right) == 1)
                p = rotate_RL(p);
            else p = rotate_RR(p);
        }
    } else {
        z = p;
        p = p->_right;
    }
    update_height(p);
}

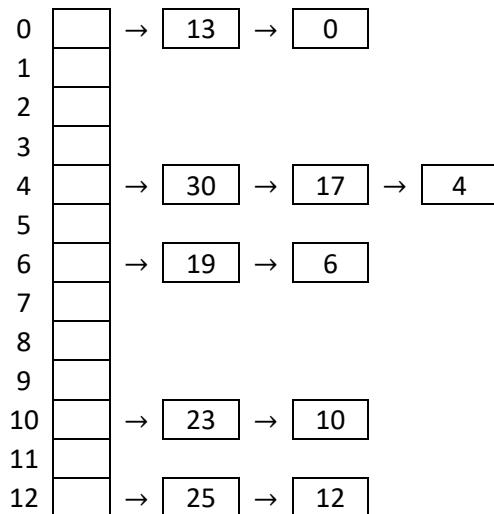
```

## Hash Tables

Allows us to store a set of elements (or pairs  $\langle \text{key}, \text{value} \rangle$ ) using a hash function, this would map every key to a distinct address of table, but we should expect **collisions** (different keys mapping to the same address  $h(x) = h(y)$ , the keys are **synonyms**) as soon as the number of elements stored in the table is  $n = \Omega(\sqrt{M})$ , for various theoretical reasons, it is a good idea that  $M$  is a prime number.

**Hash functions** → To obtain a valid position into the table an index between 0 and  $M - 1$ , the private method `hash` in class `Dictionary` computes  $h(x) \% M$  ( $h(x) = x \bmod M$ )

**Separate Chaining** → Each slot in the hash table has a pointer to a linked list of synonyms.



$$M = 13 \quad h(x) = x \bmod M$$

$$X = \{0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30\}$$

**Insertions:** Access to appropriate linked list using hash function:

- 1 - If a key was present modify associated value.
- 2 - If not add to the front of list a new node with the pair  $\langle \text{key}, \text{value} \rangle$ .

**Searching:** Access the appropriate linked list using the hash function and sequentially scan it to locate the key or report unsuccessful search.

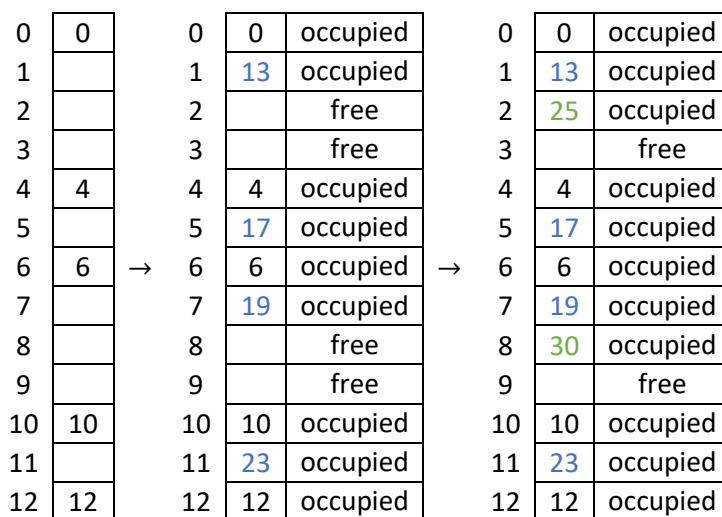
**Cost of update** →

**Cost of searching** →

**Open Addressing** → Synonyms are stored in the hash table.

Searches → probe a sequence of positions until the given key is found	The sequence of probes starts in position $i_0 = h(k)$ and continues with $i_1, i_2, \dots$
Insertions → Probe a sequence of positions until the given key or an empty slot is found	

There are different strategies for open addressing using different rules to define the sequence of probes. The simplest one is **linear probing**:  $i_1 = i_0 + 1, i_2 = i_1 + 1, \dots$



+{0, 4, 6, 10, 12}

+{13, 17, 19, 23}

+{25, 30}

## Priority queues

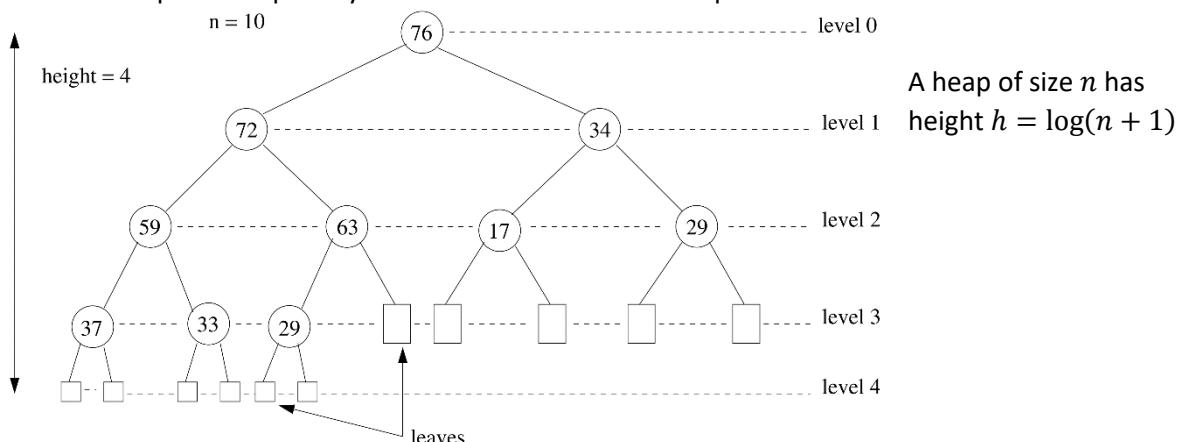
Stores a collection of elements, each one has a priority associated with it, and we order by this value. Priority queues support the insertions of new elements and the query and removal of an element of minimum (or maximum) priority.

Dictionaries techniques (not hash tables) can be also used for priority queues, AVLs can be used to implement a PQ with cost  $O(\log n)$ .

### Heap

Is a binary tree that:

- All empty subtrees are located in the last two levels of the tree.
- If a node has an empty left subtree then its right subtree is also empty.
- Max-heaps → The priority of an element is larger or equal than that of its descendants.
- Min-heaps → The priority of an element is smaller or equal than that of its descendants.



```

template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::insert(const Elemt& x, const Prio& p) {
    ++nelems;
    h.push_back(make_pair(x, p));
    siftup(nelems);
}

template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::remove_min() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    swap(h[1], h[nelems]);
    --nelems;
    h.pop_back();
    sink(1);
}

// Cost: O(log(n/j))
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::sink(int j) {
    // if j has no left child we are at the last level
    if (2 * j > nelems) return;
    int minchild = 2 * j;

    if (minchild < nelems and h[minchild].second > h[minchild + 1].second) ++minchild;

    // minchild is the index of the child with minimum priority
    if (h[j].second > h[minchild].second) {
        swap(h[j], h[minchild]);
        sink(minchild);
    }
}

```

```
// Cost: O(log j)
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::siftup(int j) {
    // if j is the root we are done
    if (j == 1) return;
    int father = j / 2;
    if (h[j].second < h[father].second) {
        swap(h[j], h[father]);
        siftup(father);
    }
}
```

## Heapsort

Sorts an array of  $n$  elements building a heap with the  $n$  elements and extracting them, one by one, from the heap. The cost is  $O(n \log n)$ .

```
// Establish (max) heap order in the
// array v[1..n] of Elemt's; Elemt == priorities
// this is a.k.a. as heapify
template <typename Elemt>
void make_heap(Elemt v[], int n) {
    for (int i = n/2; i > 0; --i)
        sink(v, n, i);
}

template <typename Elemt>
void sink(Elemt v[], int sz, int pos);

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elemt>
void heapsort(Elemt v[], int n) {
    make_heap(v, n);
    for (int i = n; i > 0; --i) {
        // extract largest element from the heap
        swap(v[1], v[i]);
        // sink the root to reestablish max-heap order
        // in the subarray v[1..i-1]
        sink(v, i-1, 1);
    }
}
```

# Graphs

## Basic Graph Theory

An **graph** (undirected graph) is a pair  $G = \langle V, E \rangle$  where  $V$  is a finite set of vertices (nodes) and  $E$  is a set of edges; each edge  $e \in E$  is an unordered pair  $(u, v)$  with  $u \neq v$  and  $u, v \in V$ .

$$\text{The number of edges } |E| = m; \quad 0 \leq m \leq \frac{n(n-1)}{2}$$

$$\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = |E|$$

A **digraph** (directed graph) is a graph but the edges have a direction associated with them.

$$\text{The number of arcs } m: \quad 0 \leq m \leq n(n - 1)$$

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|$$

For an arc  $e = (u, v)$ , vertex  $u$  is called the **source** and a vertex  $v$  the **target**. We say that  $v$  is a **successor** of  $u$ ; conversely,  $u$  is a **predecessor** of  $v$ . For an edge  $e = \{u, v\}$ , the vertices are called its **extremes** and we say  $u$  and  $v$  are **adjacent**. We also say that the edge  $e$  is **incident** to  $u$  and  $v$ .

**Degree:** number of edges incident to a vertex  $u$  (number of vertices  $v$  adjacent to  $u$ )  $\rightarrow \deg(v)$ .

- **In-degree:** number of successors of the vertex  $u \rightarrow \text{in-deg}(u)$ .
- **Out-degree:** number of predecessors of the vertex  $u \rightarrow \text{out-deg}(u)$ .

A graph is:

- **Dense** if the number of edges is close to the maximal number of edges,  $m = \Theta(n^2)$ . Ex: complete graphs.
- **Sparse** otherwise. Ex: cyclic graphs and  $d$ -regular graphs.
- A **path** is a graph such all the edges connected by a vertex (except the first and the last one).
- A **simple path** is a path that no vertex appears more than once in a sequence (except the first and the last one).
- A **cycle** is a simple path that the first and the last node are the same.
- **Acyclic** if does not have any cycle.
- **Hamiltonian** if contains at least a simple path that visits all vertices of the graph.
- **Eulerian** if only a path contains all edges/arcs of the graph.
- **Connected** if and only if there exists a path in  $G$  from  $u$  to  $v$  for all pair of vertices  $u, v \in V$ .

Is **subgraph** of  $G$  if the graph is the same but without some edges or vertex.

- **Subgraph induced** if is a subgraph with the same edges/arcs than  $G$ .
- **Spanning subgraph** if is a subgraph with the same vertices than  $G$ .

A **connected component**  $C$  of a graph  $G$  is a maximal connected induced subgraph of  $G$ . By maximal we mean that adding any vertex  $v$  to  $V(C)$  the resulting induced subgraph is not connected.

Trees

- A **(free) tree** is a connected acyclic graph. Then  $|E| = |V| - 1$ .
  - A **spanning tree** is a spanning subgraph and a tree.
- A **forest** is an acyclic graph. Then  $|E| = |V| - c$  ( $c$  = connected components).
  - A **spanning forest** is an acyclic spanning subgraph consisting of a set of trees, each a spanning tree for the corresponding connected components of  $G$ .

Often we will see (di)graphs in which each edge/arc bears a **label**, these are numeric (integers, rational, real numbers). When a graph is labelled then we call it **weighted graphs**, and the label of the edge is called its weight.

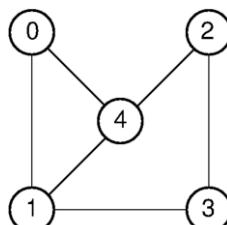
## Implementation of Graphs

**Adjacency matrices:** are too costly in space; if  $|V(G)| = n \rightarrow$  it needs space  $\Theta(n^2)$  to represent the graph. Are fine to represent dense graphs or when we need to answer very efficiently whether an edge  $(u, v) \in E$  or not.

- Entry  $A[i][j]$  is a Boolean indicating whether  $(i, j)$  is an edge/arc or not.
- For a weighted (di)graph  $A[i][j]$  stores the label assigned to the edge/arc  $(i, j)$ .

**Adjacency lists:** They require space  $\Theta(n + m)$  where  $n = |V|$  and  $m = |E|$ , in general rule use this implementation.

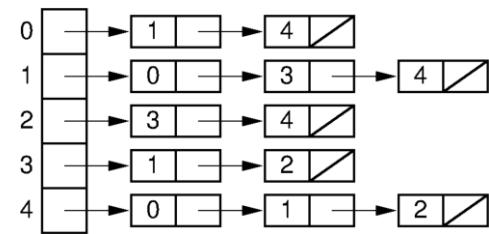
- We use an array or vector  $T$  such that for a vertex  $u$ ,  $T[u]$  points to a list of edges incident to  $u$  or a list of the vertices  $v \in V$  which are adjacent to  $u$ .
- For digraphs, we will have the lists of successors of a vertex  $u$ , for all vertices  $u$ , and, possibly, the list of predecessors of a vertex  $u$ , for all vertices  $u$ .



Graph

	0	1	2	3	4
0	1			1	
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

Adjacency matrix



Adjacency list

```

typedef int vertex;
typedef pair<vertex, vertex> edge;

class Graph {
// undirected graphs with V = {0, ..., n-1}
public:
// create an empty graph (no vertices and no edges)
Graph();
// create an empty graph with n vertices (but no edges)
Graph(int n);
// adds a new vertex to the graph; the new vertex will have identifier 'n' where n is
// the number of vertices in the graph, before adding the new vertex
void add_vertex();

// adds edge (u,v) to the graph, either giving an edge e=(u,v) or its extremes u and v
void add_edge(vertex u, vertex v);
void add_edge(edge e);

// return the number of vertices and edges, respectively
int nr_vertices() const;
int nr_edges() const;

// return the list of edges incident to vertex u
list<edge> adjacent(vertex u) const;
...

private:
    int n, m;
    vector<list<edge>> T;
    // alternatives: vector<list<vertex>> T;
    // vector<vector<vertex>> T;
    ...
}

```

**Traversals of graphs:** are the different ways to go through a graph, allows us to efficiently solve many problems on graphs:

- Decide whether a graph is connected or not.

- Find the connected components of an undirected graph.
- Find the strongly connected components of a digraph.
- Find whether a graph contains cycles or not.
- Decide if a graph is bipartite or not (equivalently, if a graph is 2-coloreable or not)
- Decide whether a graph is biconnected or not (a graph is biconnected if and only if the removal of any vertex and its incident edges does not disconnect the graph).
- Find the shortest path (with least number of edges/arcs) between any two given vertices

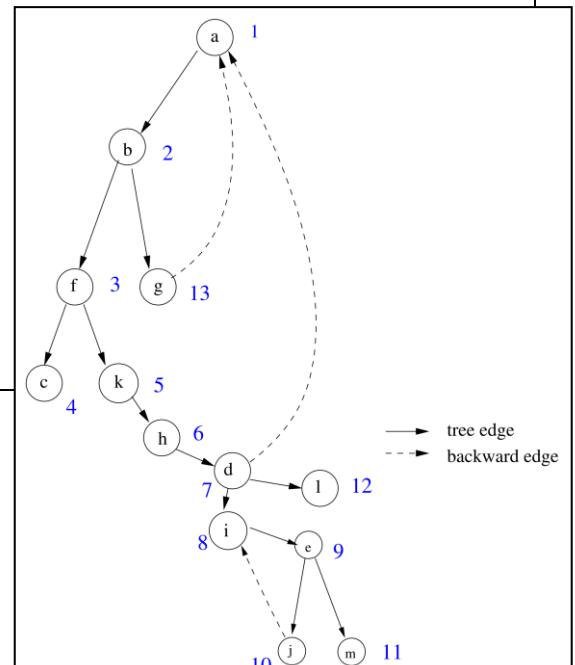
## DFS (Depth-First Search)

We visit a vertex  $v$  and from there we traverse recursively, each non-visited vertex  $w$  which is adjacent/a successor of  $v$ . A vertex remains **open** until the recursive traversal of all its adjacent/successors has been finished, after that the vertex gets **closed**.  $\Theta(n + m)$

- The **Direct** or **DFS number** of a vertex is the ordinal number in which the vertex is open the DFS. (Ex: If the DFS starts at vertex  $v$ , then the DFS number of  $v$  is 1).
- The **Inverse number** of a vertex is the ordinal number in which the vertex is closed by the DFS. The first vertex in the DFS for which all successors have been visited has inverse number 1.

```
// Are global variables, for simplicity
// visited -> bool vector of visited and not visited nodes
// ndfs -> int vector which contains the visited open position
// ninv -> int vector which contains the order of closed vertex
// num_dfs, num_inv -> int variables
procedure DFS(G)
    for  $v \in V(G)$  do
        visited[v] := false
        ndfs[v] := 0; ninv[v] := 0
    end for
    num_dfs := 0; num_inv := 0
    for  $v \in V(G)$  do
        if  $\neg$ visited[v] then
            DFS-REC(G, v, v)
        end if
    end for
end procedure

procedure DFS-REC(G, v, father)
    PRE-VISIT(v)
    visited[v] := true
    num_dfs := num_dfs + 1; ndfs[v] := num_dfs
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg$ visited[w] then
            PRE-VISIT-EDGE(v, w)
            DFS-REC(G, w, v)
            POST-VISIT-EDGE(v, w)
        else
            // If  $w \neq \text{father}$  there is a cycle (that contains edge  $(v, w)$ )
            end if
        end for
        POST-VISIT(v)
        num_inv := num_inv + 1; ninv[v] := num_inv
    end procedure
```



All edges in the connected component (CC) can be classified in two types:

- **Tree edges**: the normal edges that DFS follows.
- **Backward edges**: edges that can return to a previous visited edge without follow previous edges.

**DFS in digraphs** we not only visit strongly connected component (SCC), we visit all accessible (not visible) vertices from  $v$ . Each call to  $\text{DFS}(v, \dots)$  induces a directed tree, with  $v$  as a root. DFS numbers and inverse numbers are defined as in DFS for undirected graphs. But we have different types of arcs:

- **Tree edges**: from currently visited vertex  $v$  to a non-visited vertex  $w$ .
- **Backward edges**: from currently visited vertex  $v$  to an ascendant (from above)  $w$  that still open,  $\text{ndfs}[v] < \text{ndfs}[w]$  and  $\text{ndfs}[w]$  is open.
- **Forward edges**: from currently visited vertex  $v$  to a previously visited descendant (from below)  $w$ ;  $\text{ndfs}[v] < \text{ndfs}[w]$ .
- **Cross edges**: from currently visited vertex  $v$  to a previously visited vertex  $w$  in the same DFS tree or a different traversal tree;  $\text{ndfs}[w] < \text{ndfs}[v]$ , but  $w$  is already closed ( $\text{ninv}[w] \neq 0$ ).

**DAG** (Directed Acyclic Graph): where in a large complex software system, each node is a subsystem and each arc  $(A, B)$  indicates that subsystem  $A$  uses subsystem  $B$ .

A **Topological sort** of a DAG  $G$  is a sequence of all its vertices such that for any vertex  $w$ , if  $v$  precedes  $w$  in the sequence then  $(w, v) \notin E$  (no vertex is visited until all its predecessors have been visited).

```
// By definition we know that there are no cycles in the graph
// pred[v] => number of predecessors of v that have not been visited yet

procedure TOPOLOGICALSORT(G)
    for  $v \in G$  do  $\text{pred}[v] := 0$ 
    end for
    for  $v \in G$  do
        for  $w \in G.\text{SUCCESSORS}(v)$  do  $\text{pred}[w] := \text{pred}[w] + 1$ 
        end for
    end for
     $Q := \emptyset$ 
    for  $v \in G$  do
        if  $\text{pred}[v] = 0$  then
             $Q.\text{PUSH\_BACK}(v)$ 
        end if
    end for
    . . .
end procedure
```

## BFS (Breadth-First Search)

Given a vertex  $s$  we visit all vertices in the connected component of  $s$  in increasing distance from  $s$ . When a vertex is visited, all its adjacent non-visited vertices are put into a queue of vertices yet to be visited.  $\Theta(|V| + |E|)$

```
procedure BFS(G, s, D)
    // Assumes G is connected; all vertices will be visited
    // After execution, D[v] = distance from s to v
    for  $v \in G$  do
         $D[v] := 1$  // D[v] = 1 indicates that v hasn't been visited
    end for
     $Q := \emptyset$ ;  $Q.\text{PUSH}(s)$ ;  $D[s] := 0$ 
    while  $\neg Q.\text{EMPTY}()$  do
         $v := Q.\text{POP}()$ 
        VISIT( $v$ )
        for  $w \in G.\text{ADJACENT}(v)$  do
            if  $D[w] = -1$  then
                 $D[w] := D[v] + 1$ 
                 $Q.\text{PUSH}(w)$ 
            end if
        end for
    end while
end procedure
```

**Diameter** of a graph: Is the maximal distance between a pair of vertices.

- To compute the diameter we need to perform a BFS starting from each vertex  $v$  in  $G$ .  $\Theta(n \cdot m)$

**Center** of a graph: is the vertex such that the maximal distance to any other vertex is minimal .

## Dijkstra's Algorithm

Find all the shortest paths from a given vertex to all other vertices in the digraph.

The worst-case cost is  $\Theta((m + n) \log n)$

```
template <class Elemt, class Prio>
class DijkstraPrioQueue<Elemt,Prio> {
public:
    DijkstraPrioQueue(int n = 0);
    void insert(const Elemt& e, const Prio& prio);
    Elemt min() const;
    void remove_min();
    Prio prio(const Elemt& e) const; // returns the priority of element v
    bool contains(const Elemt& e) const;
    void decrease_prio(const Elemt& e, const Prio& newprio);
    bool empty() const;
    ...
};

typedef vector<list<pair<int,double>>> graph;
...
void Dijkstra(const graph& G, int s, ... ) {
    DijkstraPrioQueue<int,double> cand(G.size());
    for (int v = 0; v < G.size(); ++v)
        cand.insert(v, INFINITY);

    cand.decrease_prio(s, 0);

    while(not cand.empty()) {
        int u = cand.min(); double du = cand.prio(u);
        cand.remove_min();
        for (auto e : G[u]) {
            // e is a pair <v, weight(u,v)>
            int v = e.first;
            double d = du + e.second;
            if (d < cand.prio(v))
                cand.decrease_prio(v, d);
        }
    }
}
```

## Minimum Spanning Trees: Prim's Algorithm

Obtain a subgraph with the minimum weight that contains all the edges of the graph with no-cycles.

Procedure: start by the lightest vertex and look for the weightless edge, once you are on the second edge we look in all visited edges for the lighter edge without a visited vertex. Do the same until all edges of the original graph are in our subgraph.

The cost if we use a priority queue giving a started bound is  $O(n \cdot m)$

```
typedef pair<int,int> edge;
typedef pair<double,edge> weighted_edge;
typedef vector< list<weighted_edge> > graph;

void Prim(const graph& G, list<edge>& MST, double& MST_weight) {
    vector<bool> visited(G.size(), false);
    priority_queue< weighted_edge, vector<weighted_edge>,
                    greater<weighted_edge> > Candidates;
    visited[0] = true;
    for (weighted_edge x : G[0]) Candidates.push(x);
    MST_weight = 0.0;
    while (MST.size() != G.size()-1) {
        int weight = Candidates.top().first;
        edge uv = Candidates.top().second;
        int v = uv.second;
        Candidates.pop();
        if (not visited[v]) {
            visited[v] = true;
            MST.push_back(uv); MST_weight += weight;
            for (weighted_edge y : G[v]) Candidates.push(y);
        }
    }
}
```

## Exhaustive Search and Generation

The computational problems in which solutions are n-tuples (tuple: mathematical object with structures, that are allow to be discompose in a number of components, in this case a graph can be discompose in some vertices and edges) which satisfy some additional constrains. We look for:

- 1- Finding all solutions
- 2- Finding if exist at least one solution (and return such a solution)
- 3- Finding an optimal solution according to some criterion (maximizing a benefit/minimizing a cost)

## Backtracking

Is a blind search scheme, since the order of exploration of the configuration trees is fixed beforehand

```
// x is the current partial solution
procedure BACKTRACK(k)
    if IS_SOLUTION(x) then
        PROCESS(x)
    else
        for v ∈ Dk do
            // exit from the loop if we are looking
            // for one solution and it has been found
            x[k] := v
            MARK(x; v)
            if IS_FEASIBLE(x; k) then
                BACKTRACK(k + 1)
            else // feasibility pruning + CBSP if optimizing
            end if
            UNMARK(x; v)
        end for
    end if
end procedure
```

## Branch & Bound

Branch & Bound is an informed search scheme since it explores the configuration tree in order of estimated costs (or estimated benefits). The most promising areas of the configuration tree are explored before other areas. Since B&B is usually finding better solutions before Backtracking does, but B&B is more costly in memory usage.

```
procedure BRANCH-AND-BOUND(z)
    Alive: a priority queue of nodes
    y := ROOT_NODE(z)
    Alive.INSERT(y, ESTIMATED-COST(y))
    Initialize best_solution and best_cost, e.g., best_cost := +1
    while ¬Alive:IS_EMPTY() do
        y := Alive.MIN_ELEM(); Alive.REMOVE_MIN()
        for y' ∈ SUCCESSORS(y, z) do
            if IS_SOLUTION(y', z) then
                if COST(y') < best_cost then
                    best_cost := COST(y')
                    best_solution := y'
                    Purge nodes with larger priority = cost
                end if
            else
                feasible := IS_FEASIBLE(y', z) ^
                    ESTIMATED-COST(y') < best_cost
                if feasible then
                    Alive.INSERT(y', ESTIMATED-COST(y'))
                end if
            end if
        end for
    end while
end procedure
```

## Notions of Interactability

**Complexity Theory** aims at finding the complexity of computational problems and classify them according to their complexity. The **analysis of algorithms** studies the amount of resources needed by an algorithm to solve a problem, instead the complexity theory consider the possible algorithms to solve a problem. Classifications of problems:

- A **decisional problem** is a computational problem in which for every possible input  $x$ , one must determine if a certain property is satisfied or not (the output is “yes/no”, “true/false”, “0/1”).
- The solution of a decisional problems often can be used for an “efficient” solution to the **non-decisional problem**.

A decisional problem  $P$  is **decidable in time  $t$** , because it depends if the problem can be solved.

- **Polynomial time:** a problem belongs to  $P$  if is decidable in time  $n^k$  by some  $k$
- **Exponential time:** a problem is EXP if it is decidable in time  $2^{n^k}$  by some  $k$ .

Examples:

- CONNECTIVITY  $\in P$
- REACHABILITY  $\in P$
- PRIMALITY  $\in P$
- SHORTEST-PATH  $\in P$
- 2-COLORABILITY  $\in P$
- 3-COLORABILITY  $\in EXP$ ; also  $\in P$ ? (not known to be  $P$ )
- LONGEST-PATH  $\in EXP$ ; also  $\in P$ ? (not known to be  $P$ )
- TSP  $\in EXP$ ; also  $\in P$ ? (not known to be  $P$ )
- GENERALIZED-CHESS  $\in EXP$
- GENERALIZED-CHECKERS  $\in EXP$
- GENERALIZED-GO  $\in EXP$

The algorithms seen so far are **deterministic algorithms**: the computation path from the input to the output is unique .

The **non-deterministic algorithms** have many distinct computational paths, forming a tree. These start the algorithm as deterministic algorithm until the first function  $CHOOSE(y)$  which returns a value between 0 and  $y$ , each value corresponding to one possible solution.

- amb un algorisme d'ordenació bàsic (bombolla, inserció):  $O(n^2)$
- amb un algorisme d'ordenació eficient:  $O(n \log n)$

## Mida de l'entrada i cost

### Mida

La **mida** (o **talla**) d'una entrada  $x$  és el nombre de símbols necessari per codificar-la. Es representa amb  $|x|$ .

### Convencions segons el tipus d'entrada

- Nombres naturals** → codificació en binari

$$|27| = 5 \text{ perquè } \langle 27 \rangle_2 = 11011$$

- Llistes, vectors** → nombre de components

$$|(23, 1, 7, 0, 12, 500, 2, 11)| = 8$$

### Notació

A partir d'ara, escriurem **log** en lloc de **log<sub>2</sub>**.

### Notació O gran

Donada una funció  $g$ ,  $O(g)$  és la classe de funcions  $f$  que “no creixen més de pressa que  $g$ ”. Formalment,  $f \in O(g)$  si existeixen  $c > 0$  i  $n_0 \in \mathbb{N}$  tals que

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

En lloc de  $f \in O(g)$ , s'escriu sovint “ $f$  és  $O(g)$ ” o, també,  $f = O(g)$ .

### Exemple

Sigui  $f(n) = 3n^3 + 5n^2 - 7n + 41$ . Llavors, podem afirmar que  $f \in O(n^3)$ .

Per justificar-ho, només cal trobar constants  $c$  i  $n_0$  tals que

$$\forall n \geq n_0 \quad f(n) \leq cn^3.$$

Però  $3n^3 + 5n^2 - 7n + 41 \leq 8n^3 + 41$ . Triem  $c = 9$ . Llavors,

$$8n^3 + 41 \leq 9n^3 \iff 41 \leq n^3,$$

que es compleix a partir de  $n_0 = 4$ . Per tant,  $\forall n \geq 4 \quad f(n) \leq 9n^3$  i, llavors,  $f(n) = O(n^3)$  amb  $c = 9$  i  $n_0 = 4$ .

## Notació asimptòtica: definicions

### Notació $\Theta$ ((a): fita exacta asimptòtica)

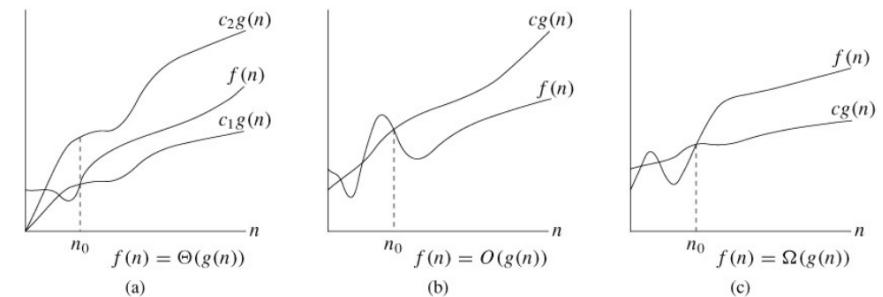
$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad c_1g(n) \leq f(n) \leq c_2g(n)\}$$

### Notació O gran ((b): fita superior asimptòtica)

$$O(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)\}$$

### Notació $\Omega$ ((c): fita inferior asimptòtica )

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)\}$$



## Notació asimptòtica: propietats

### Relacions entre $O$ , $\Omega$ i $\Theta$

Donades dues funcions  $f$  i  $g$ :

- $f \in \Omega(g) \iff g \in O(f)$
- $\Theta(f) = O(f) \cap \Omega(f)$
- $O(f) = O(g) \iff \Omega(f) = \Omega(g) \iff \Theta(f) = \Theta(g)$

### Regla del límit

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$  però  $g \notin O(f)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow g \in O(f)$  però  $f \notin O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ , on  $0 < c < \infty \Rightarrow O(f) = O(g)$

## Notació asimptòtica: propietats

### Propietats de l'O gran

Donades les funcions  $f, f_1, f_2, g, g_1, g_2$  i  $h$ :

- **Reflexivitat.**  $f \in O(f)$
- **Transitivitat.**  $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- **Caracterització.**  $f \in O(g) \iff O(f) \subseteq O(g)$
- **Regla de la suma.**  $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max(g_1, g_2))$
- **Regla del producte.**  $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- **Invariança multiplicativa.** Per a tota constant  $c \in \mathbb{R}^+$ ,  $O(f) = O(c \cdot f)$

### Exercici

Feu servir la regla del límit per demostrar la transitivitat de l'O gran, és a dir, que si  $f, g, h$  són funcions, llavors  $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$ .

Suposant que  $f \in O(g)$  i  $g \in O(h)$ , tenim que

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \quad \wedge \quad \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} < \infty.$$

Aleshores,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \lim_{n \rightarrow \infty} \frac{f(n) \cdot g(n)}{g(n) \cdot h(n)} = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \cdot \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} < \infty$$

i, per tant,  $f \in O(h)$ .

### Exercici

Argumenteu per què l'affirmació  $f \in O(g)$  és equivalent a

$$\exists c \in \mathbb{R}^+ \quad \forall n \quad f(n) \leq c \cdot g(n).$$

Recordeu que, per definició,  $f \in O(g)$  si

$$\exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

### Nota

La notació  $\forall^\infty n P(n)$  representa que  $P(n)$  es compleix per a tots els valors de  $n$  excepte per a un nombre finit.

## Propietats de $\Theta$

Mateixes que O gran menys caracterització per :

- **Simetria.**  $f \in \Theta(g) \iff g \in \Theta(f) \iff \Theta(f) = \Theta(g)$

## Regles de la suma i el producte (segona versió)

Donades dues funcions  $f$  i  $g$ :

- $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$
- $O(f) \cdot O(g) = O(f \cdot g)$
- $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$

## Formes de creixement

Costos freqüents

- **Constant:**  $\Theta(1)$ 
  - Decidir la paritat
  - Sumar dues variables numèriques
- **Logarítmic:**  $\Theta(\log n)$ 
  - Cerca binària
- **Lineal:**  $\Theta(n)$ 
  - Recorregut seqüencial  
(p. ex., calcular el màxim, el mínim, la mitjana)
- **Quasilineal:**  $\Theta(n \log n)$ 
  - Ordenacions per fusió (*Mergesort*) i ràpida (*Quicksort*)
- **Quadràtic:**  $\Theta(n^2)$ 
  - Suma de dues matrius quadrades
  - Ordenació per selecció i bombolla
- **Cúbic:**  $\Theta(n^3)$ 
  - Producte de dues matrius quadrades
  - Enumeració de triples
- **Polinòmic:**  $\Theta(n^k)$ , per a  $k \geq 1$  constant
  - Enumerar combinacions
  - Test de primalitat  
(amb variants de l'algorisme AKS que van de  $\Theta(n^{12})$  a  $\Theta(n^6)$ )
- **Exponencial:**  $\Theta(k^n)$ , per a  $k > 1$  constant
  - Cerca en un espai de configuracions (d'amplada  $k$  i profunditat  $n$ )
- **Altres funcions:**  $\Theta(\sqrt{n}), \Theta(n!), \Theta(n^n)$

# Formes de creixement

## Notació

Donades dues funcions  $f$  i  $g$ , escrivim  $f \prec g$  per indicar que  $f \in O(g)$  però  $g \notin O(f)$ .

## Exercici

Trobeu dos costos  $f, g$  de l'escala anterior per als quals  $f \prec \sqrt{n} \prec g$ .

## Solució

Triem  $f(n) = \log n$  i  $g(n) = n$  i apliquem la regla del límit:

①  $\log n \prec \sqrt{n}$ . Per la regla de L'Hôpital,

$$\lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{1/(\ln 2 \cdot n)}{1/2 \cdot n^{-1/2}} = \frac{2}{\ln 2} \cdot \lim_{n \rightarrow \infty} \frac{n^{1/2}}{n} = \frac{2}{\ln 2} \cdot \lim_{n \rightarrow \infty} \frac{1}{n^{1/2}} = 0.$$

②  $\sqrt{n} \prec n$ . Trivialment,  $\lim_{n \rightarrow \infty} \frac{\sqrt{n}}{n} = \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$ .

# Algorismes iteratius

## Càlcul del cost:

- El cost d'una **operació elemental** és  $\Theta(1)$ . Això inclou:
  - una assignació entre tipus bàsics (`int`, `bool`, `double`,...)
  - una lectura o escriptura d'un tipus bàsic
  - una comparació
  - una operació aritmètica
  - l'accés a un component d'un vector
  - el pas d'un paràmetre per referència
- Avaluar una **expressió** té cost igual a la suma dels costos de les operacions que s'hi fan (incloses les crides a les funcions, si n'hi ha).
- El cost de **construir o copiar un vector** de mida  $n$  (assignació, pas per valor, return) és  $\Theta(n)$ .

## Càlcul del cost:

- Si el cost de  $F$  durant la  $k$ -èsima iteració és  $C_k$ , el d'avaluar  $B$  és  $D_k$  i el nombre d'iteracions és  $N$ , llavors el cost de la **composició iterativa**

while (B) F;

és  $(\sum_{k=1}^N C_k + D_k) + D_{N+1}$ .

## Càlcul del cost:

- Si el cost d'un fragment  $F$  és  $C$  i el cost d'avaluar  $B$  és  $D$ , llavors el cost de la **composició alternativa d'una branca**

if (B) F;

és  $\leq D + C$ .

- Si el cost d'un fragment  $F_1$  és  $C_1$ , el d'un fragment  $F_2$  és  $C_2$  i el d'avaluar  $B$  és  $D$ , llavors el cost de la **composició alternativa de dues branques**

if (B) F<sub>1</sub>; else F<sub>2</sub>;

és  $\leq D + \max(C_1, C_2)$ .

## Exemple d'ordenació per selecció

Passos per ordenar la seqüència 5, 6, 1, 2, 0, 7, 4, 3 segons l'algorisme de selecció. En vermell, els elements ja ordenats. En blau, els elements intercanviats pel màxim.

5	6	1	2	0	7	4	3
5	6	1	2	0	3	4	7
5	4	1	2	0	3	6	7
3	4	1	2	0	5	6	7
3	0	1	2	4	5	6	7
2	0	1	3	4	5	6	7
1	0	2	3	4	5	6	7
0	1	2	3	4	5	6	7

## Ordenació per selecció

```
0 int posicio_maxim(const vector<int>& v, int m) {
1   int k = 0;
2   for (int i = 1; i <= m; ++i)
3     if (v[i] > v[k]) k = i;
4   return k;
5
6 void ordena_seleccio (vector<int>& v, int n) {
7   for (int i = n-1; i > 0; --i) {
8     int k = posicio_maxim(v, i);
9     swap(v[k], v[i]);
10 }
```

2, 6 Iteracions bucles:  $m - 1 + 1 = m \in \Theta(m)$ ,  $(n - 1) - 1 + 1 = n - 1 \in \Theta(n)$ .  
7 Cost  $\Theta(i)$ .

altres Instruccions de cost constant:  $\Theta(1)$ .

$$t_{sel}(n) = \Theta(1) + \sum_{i=1}^{n-1} (\Theta(i) + \Theta(1)) = \Theta(\sum_{i=1}^{n-1} i) = \Theta(\frac{(n-1)n}{2}) = \Theta(n^2)$$

## Exemple d'ordenació per inserció

Passos per ordenar la seqüència 5, 6, 1, 2, 0, 7, 4, 3 segons l'algorisme d'inserció. En vermell, els elements ja ordenats. En blau, el nombre de posicions que s'ha desplaçat l'element insertat.

5	6	1	2	0	7	4	3	(0)
5	6	1	2	0	7	4	3	(0)
1	5	6	2	0	7	4	3	(2)
1	2	5	6	0	7	4	3	(2)
0	1	2	5	6	7	4	3	(4)
0	1	2	5	6	7	4	3	(0)
0	1	2	4	5	6	7	3	(3)
0	1	2	3	4	5	6	7	(4)

## Ordenació per inserció

```
0 void ordena_insercio(vector<int>& v, int n) {
1     for (int k = 1; k <= n-1; ++k) {
2         int t = k-1;
3         while (t >= 0 and v[t+1] < v[t]) {
4             swap(v[t], v[t+1]);
5             --t; }}
```

0 Pas de paràmetres:  $\Theta(1)$ .

1 Iteracions bucle:  $(n - 1) - 1 + 1 = n - 1 \in \Theta(n)$ .

1,2 Condició d'iteració i línia 2:  $\Theta(1)$ .

3 Iteracions bucle: entre 0  $\in \Theta(1)$  i  $k - 1 - 0 + 1 = k \in \Theta(k)$ .

4,5 Assignacions amb cost  $\Theta(1)$ .

$$\Theta(1) + (\Theta(n) \times \Theta(1)) \leq t_{ins}(n) \leq \Theta(1) + \sum_{k=1}^{n-1} \Theta(k)$$

Però sabem que

$$\sum_{k=1}^{n-1} k = 1 + 2 + \dots + (n - 1) = \frac{(n - 1)n}{2}.$$

Aleshores,

$$\sum_{k=1}^{n-1} \Theta(k) = \Theta\left(\sum_{k=1}^{n-1} k\right) = \Theta\left(\frac{(n - 1)n}{2}\right) = \Theta(n^2)$$

i, per tant,

$$\Theta(n) \leq t_{ins}(n) \leq \Theta(n^2).$$

## Algorismes recursius

### Exemple

$$C(n) = \begin{cases} 1, & \text{si } n = 1 \\ C(n - 1) + n, & \text{si } n \geq 2 \end{cases}$$

### Idea

Podem observar que

- $C(1) = 1$
- $C(2) = 1 + 2 = 3$
- $C(3) = 3 + 3 = 6$
- $C(n) = C(n - 1) + n = C(n - 2) + (n - 1) + n = \dots$

### Solució

$$\begin{aligned} C(n) &= C(n - 1) + n \\ &= C(n - 2) + (n - 1) + n \\ &= C(n - 3) + (n - 2) + (n - 1) + n \\ &\vdots \\ &= C(1) + 2 + \dots + (n - 2) + (n - 1) + n \\ &= 1 + 2 + \dots + n \\ &= \sum_{i=1}^n i = \frac{n(n + 1)}{2} \in \Theta(n^2). \end{aligned}$$

### Cerca lineal recursiva

Comprovar si un nombre  $x$  apareix en un vector  $v$  entre les posicions 0 i  $n - 1$  comparant-lo amb  $v[0], v[1], \dots, v[n - 1]$ .

Si es troba  $x$ , retornar la seva posició en  $v$ . Altrament, retornar -1.

```
int cerca_lineal(const vector<int>& v, int n, int x) {
    if (n == 0) return -1;
    else if (v[n-1] == x) return n-1;
    else return cerca_lineal(v, n-1, x);}
```

El paràmetre de recursió és  $n$ , la mida del vector. Definim la recurrència  $T(n)$  que representa el cost (en cas pitjor) de l'algorisme:

$$T(n) = T(n - 1) + \Theta(1)$$

## Recurrència per a la cerca lineal

$$\begin{aligned}T(n) &= T(n-1) + \Theta(1) \text{ per a } n \geq 1, \text{ i} \\T(0) &= \Theta(1).\end{aligned}$$

## Solució

$$\begin{aligned}T(n) &= T(n-1) + \Theta(1) \\&= T(n-2) + 2 \cdot \Theta(1) \\&= T(n-3) + 3 \cdot \Theta(1) \\&\vdots \\&= T(0) + n \cdot \Theta(1) \\&= (n+1) \cdot \Theta(1) \\&= \Theta(n+1) = \Theta(n).\end{aligned}$$

## Recurrència per a la cerca binària

$$\begin{aligned}T(n) &= T(n/2) + \Theta(1) \text{ per a } n \geq 1, \text{ i} \\T(0) &= \Theta(1).\end{aligned}$$

## Solució

$$\begin{aligned}T(n) &= T(n/2) + \Theta(1) \\&= T(n/4) + 2 \cdot \Theta(1) \\&= T(n/8) + 3 \cdot \Theta(1) \\&\vdots \\&= T(n/2^{\log n}) + \log n \cdot \Theta(1) \\&= T(1) + \log n \cdot \Theta(1) \\&= T(0) + (\log n + 1) \cdot \Theta(1) \\&= (\log n + 2) \cdot \Theta(1) = \Theta(\log n + 2) = \Theta(\log n).\end{aligned}$$

## Cerca binària recursiva

Comprovar si un nombre  $x$  apareix en un vector ordenat  $v$  entre les posicions  $i$  i  $j$  per cerca binària.

Si es troba  $x$ , retornar la seva posició en  $v$ . Altrament, retornar  $-1$ .

```
int cerca_binaria(const vector<int>& v, int i, int j, int x)
{ if (i <= j) {
    int k = (i + j) / 2;
    if (x == v[k])
        return k;
    else if (x < v[k])
        return cerca_binaria(v, i, k-1, x);
    else
        return cerca_binaria(v, k+1, j, x);
}
else return -1;
}
```

El paràmetre de recursió és  $n = j - i$ , la mida de l'interval a explorar. Definim la recurrència  $T(n)$ , el cost (en cas pitjor) de l'algorisme:

$$T(n) = T(n/2) + \Theta(1)$$

## Teoremes mestres

Per sistematitzar l'anàlisi del cost dels algoritmes recursius, els classifiquem en dos grups en funció de com divideixen el problema d'entrada en subproblems en les crides recursives.

Sigui  $A$  un algorisme que, amb una entrada de mida  $n$ , fa  $a$  crides recursives i una feina addicional no recursiva de cost  $g(n)$ . Llavors, si en les crides recursives els subproblems tenen mida

- $n - c$ , el cost d' $A$  ve descrit per la recurrència

$$T(n) = a \cdot T(n - c) + g(n)$$

- $n/b$ , el cost d' $A$  ve descrit per la recurrència

$$T(n) = a \cdot T(n/b) + g(n)$$

Les dues menes de recurrències anteriors:

- **subtractives**:  $T(n) = a \cdot T(n - c) + g(n)$

- **divisores**:  $T(n) = a \cdot T(n/b) + g(n)$

es poden resoldre amb els **teoremes mestres** que veurem a continuació.

### Teorema mestre de recurrències subtractives

$$\text{Sigui } T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n - c) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on  $n_0 \in \mathbb{N}$ ,  $c \geq 1$ ,  $f$  és una funció arbitrària i  $g \in \Theta(n^k)$  per a  $k \geq 0$ .

Aleshores

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } a < 1 \\ \Theta(n^{k+1}), & \text{si } a = 1 \\ \Theta(a^{n/c}), & \text{si } a > 1 \end{cases}$$

### Exemple 1

Hem vist que el cost de l'algorisme recursiu de **cerca lineal** es pot descriure amb la recurrència  $T(n) = T(n-1) + \Theta(1)$  per a  $n \geq 1$ , i  $T(0) = \Theta(1)$ .

Per tant,  $n_0 = 1$ ,  $a = 1$ ,  $c = 1$ ,  $k = 0$ . Llavors,  $T(n)$  pertany al segon cas:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n).$$

## Exemple 2

En la recurrència  $T(n) = T(n - 1) + \Theta(n)$ , tenim els valors

$$a = 1, c = 1, k = 1.$$

Llavors,  $T(n)$  pertany al segon cas:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n^2).$$

## Exemple 3

En la recurrència  $T(n) = 2 \cdot T(n - 1) + \Theta(n)$ , tenim els valors

$$a = 2, c = 1, k = 1.$$

Llavors,  $T(n)$  pertany al tercer cas:

$$T(n) \in \Theta(2^n).$$

## Exemple 4

Els nombres de Fibonacci estan definits per la recurrència  $f(k) = f(k - 1) + f(k - 2)$  per a  $k \geq 2$ , amb  $f(0) = f(1) = 1$ .

La solució recursiva és evident.

```
int fibonacci (int k) {
    if (k <= 1) return 1;
    else return fibonacci(k-1) + fibonacci(k-2);
}
```

- El cost segueix la recurrència  $T(k) = T(k - 1) + T(k - 2) + \Theta(1)$
- No podem aplicar directament el teorema mestre per resoldre-la!

Podem aplicar el teorema mestre a dues fites de  $T(k)$ :

- $T(k) = T(k - 1) + T(k - 2) + \Theta(1) \leq 2T(k - 1) + \Theta(1)$  dona  $T(k) \in O(2^k)$
- $T(k) = T(k - 1) + T(k - 2) + \Theta(1) \geq 2T(k - 2) + \Theta(1)$  dona  $T(k) \in \Omega(2^{k/2}) = \Omega(\sqrt{2}^k)$

Es pot demostrar que  $T(k) = \Theta(\phi^k)$ , on  $\phi = \frac{1+\sqrt{5}}{2}$  (*nombre d'or*). Noteu que  $\sqrt{2} = 1.414213562\dots$  i  $\phi = 1.618033988\dots$

## Teorema mestre de recurrències divisores

$$\text{Sigui } T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on  $n_0 \in \mathbb{N}$ ,  $b > 1$ ,  $f$  és una funció arbitrària i  $g \in \Theta(n^k)$  per a  $k \geq 0$ .

Sigui  $\alpha = \log_b(a)$ . Aleshores,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } \alpha < k \\ \Theta(n^k \log n), & \text{si } \alpha = k \\ \Theta(n^\alpha), & \text{si } \alpha > k \end{cases}$$

## Exemple 1

Hem vist que el cost de l'algorisme recursiu de **cerca binària** es pot descriure amb  $T(n) = T(n/2) + \Theta(1)$ ,  $n \geq 1$ , i  $T(0) = \Theta(1)$ .

Per tant,  $n_0 = 1$ ,  $a = 1$ ,  $b = 2$ ,  $k = 0$ ,  $\alpha = 0$ . Llavors,  $T(n)$  pertany al 2n cas:

$$T(n) \in \Theta(n^k \log n) = \Theta(\log n).$$

## Exemple 2

Funció principal de l'ordenació per fusió (*mergesort*)

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

Tenint en compte que el cost de la crida `merge(v, e, m, d)` és  $\Theta(n)$  (on  $n = d - e + 1$ ), el cost total es pot expressar amb la recurrència:

$$T(n) = 2T(n/2) + \Theta(n) \text{ per a } n \geq 2, \text{ i } T(1) = \Theta(1).$$

## Exemple 2

Hem vist que el cost de l'**ordenació per fusió** es pot descriure amb la recurrència  $T(n) = 2T(n/2) + \Theta(n)$  per a  $n \geq 2$  i  $T(1) = \Theta(1)$ .

Per tant,  $n_0 = 2$ ,  $a = 2$ ,  $b = 2$ ,  $k = 1$ ,  $\alpha = 1$ . Llavors,  $T(n)$  pertany al 2n cas:

$$T(n) \in \Theta(n^k \log n) = \Theta(n \log n).$$

## Exercici 1

Resoleu la recurrència  $T(n) = T(\sqrt{n}) + 1$ .

### Solució

Fem el canvi de variable  $m = \log n$ . Aleshores,

$$T(n) = T(2^m) = T(2^{m/2}) + 1.$$

Definim  $S(m) = T(2^m)$ , que compleix

$$S(m) = S(m/2) + 1.$$

Pel segon teorema mestre, tenim que  $S(m) \in \Theta(\log m)$  i, per tant:

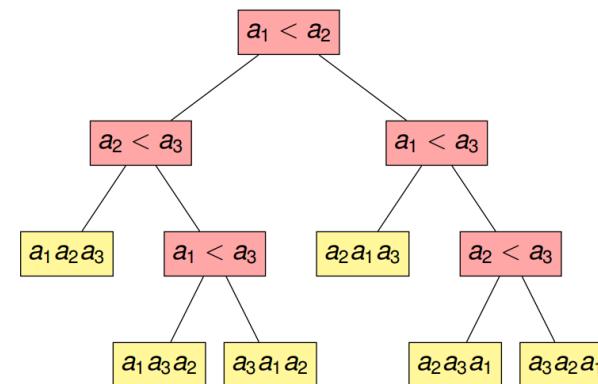
$$T(n) = T(2^m) = S(m) \in \Theta(\log m) = \Theta(\log \log n).$$

## Fita inferior dels algorismes d'ordenació

### Proposició

Tot algorisme d'ordenació basat en comparacions té cost  $\Omega(n \log n)$ .

Suposem que volem ordenar  $a_1, a_2$  i  $a_3$ . Si  $a_1 < a_2$ , seguim per la branca esquerra; si no, per la dreta. Els rectangles grocs representen les ordenacions trobades. **L'alçària de l'arbre és el cost en cas pitjor.**



- com que hi ha  $n!$  permutacions de  $n$  elements, l'arbre té  $\geq n!$  fulles
- tot arbre binari amb  $\geq k$  fulles té alçària  $\geq \log k$
- per tant, **l'alçària del nostre arbre és almenys de  $\log n!$**

El cost de l'algorisme representat per l'arbre és, per tant,  $\Omega(\log n!)$ . Com que

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

tenim que

$$\log n! \geq \log(n/2)^{(n/2)} = \frac{n}{2} \log(n/2) \in \Omega(n \log n).$$

## Algorisme de fusió bàsic

L'**ordenació per fusió**, o *mergesort*, és un bon exemple de l'esquema dividir i vèncer que fa servir un nombre de comparacions gairebé òptim.

Donat un vector  $T$  de talla  $\geq 2$ , l'algorisme consisteix a:

- ➊ Partir  $T$  en dues meitats
- ➋ Ordenar recursivament la meitats de  $T$  per separat
- ➌ Retornar la fusió de les dues meitats

L'operació clau (punt 3) consisteix a **fusionar** dos vectors ordenats en un.

### Exemple de fusió

entrada	E	X	E	M	P	L	E	F	U	S	I	O
ordenar 1a meitat	E	E	L	M	P	X	E	F	U	S	I	O
ordenar 2a meitat	E	E	L	M	P	X	E	F	I	O	S	U
resultat fusió	E	E	E	F	I	L	M	O	P	S	U	X

## Algorisme de fusió bàsic

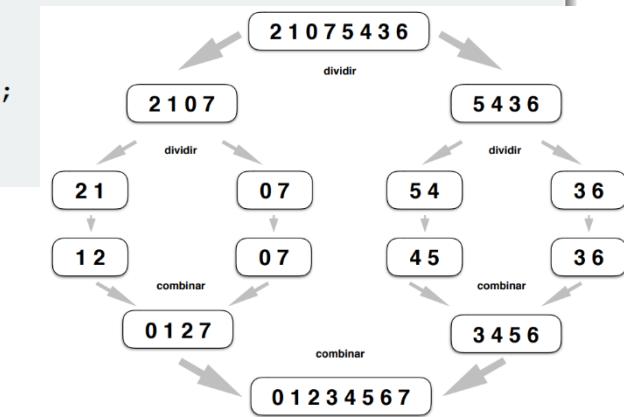
### Ordenació per fusió (Algorismes en C++, EDA)

```

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

```

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



## Algorisme de fusió bàsic

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

### Exemple

0	1	2	7
3	4	5	6

0
---

### Exemple

0	1	2	7
3	4	5	6

0	1	2	3	4	5	6
---	---	---	---	---	---	---

### Observació

Cada comparació afegeix un element a la taula  $B$  excepte l'última, que n'afegeix almenys dos.

- Per tant, el nombre de **comparacions** de tipus `elem` és  $< n = d - e + 1$
- El nombre d'**assignacions** de tipus `elem` és  $2n$
- El cost és **lineal** (assumint que assignar un `elem` és  $\Theta(1)$ )

Donat que el procediment `merge` és lineal, el cost de l'ordenació per fusió es pot expressar fàcilment amb la recurrència

$$T(n) = \begin{cases} \Theta(1), & \text{si } n = 1 \\ 2T(n/2) + \Theta(n), & \text{si } n > 1 \end{cases}$$

i, aplicant el teorema mestre de recurrències divisores, tenim que

$$T(n) \in \Theta(n \log n).$$

## Variants

Ordenació per fusió amb inserció per a vectors petits (*Alg. en C++, EDA*)

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

Les dues **versions iteratives** que veurem parteixen del fet que les fusions només comencen al final de la recursió, de manera que:

- comencen directament pels elements a ordenar i
- arriben al vector ordenat mitjançant fusions.

## Variants

Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject (Q, e)`: afegir l'element `e` a la cua `Q` i
- `eject (Q)`: funció que extreu i retorna l'últim element de `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (cua buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

3	0	2	6	5	1	4
---	---	---	---	---	---	---

2	6	5	1	4	03
---	---	---	---	---	----

5	1	4	03	26
---	---	---	----	----

4	03	26	15
---	----	----	----

26	15	034
----	----	-----

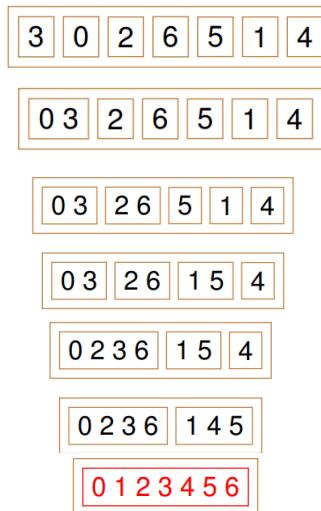
034	1256
-----	------

0123456
---------

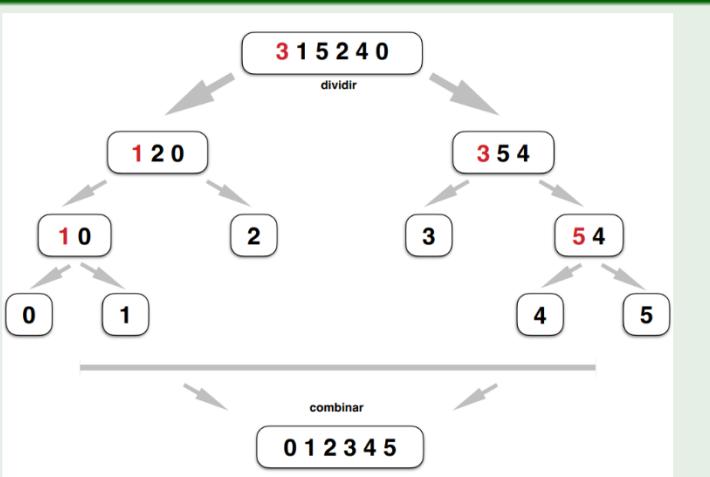
034	1256
-----	------

## Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

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



### Exemple



## Algorisme general

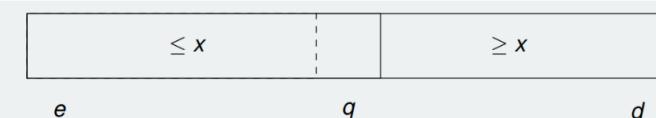
### Ordenació ràpida (Algorismes en C++, EDA)

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

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

$q = \text{partition}(T, e, d)$

- Precondició:  $0 \leq e \leq d \leq T.size() - 1$
- Postcondició:  $\exists x \text{ (pivot)} \forall i$ 
  - si  $e \leq i \leq q$ , tenim que  $T[i] \leq x$
  - si  $q < i \leq d$ , tenim que  $T[i] \geq x$



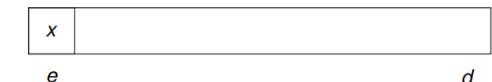
## Partició de Hoare

Partició original de Hoare amb el primer element com a pivot.

### Partició de Hoare (Algorismes en C++, EDA)

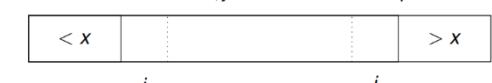
```
template <typename elem>
int partition (vector<elem>& T, int e, int d) {
    elem x = T[e];
    int i = e - 1;
    int j = d + 1;
    for (;;) {
        while (x < T[--j]);
        while (T[++i] < x);
        if (i >= j) return j;
        swap(T[i], T[j]);
    } }
```

- Inici de la funció  $\text{partition}(T, e, d)$ :

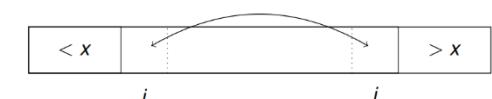


- Bucle principal:

- es troben els valors de  $i, j$  més centrals tals que



- s'intercanvien els continguts de les posicions  $i, j$



## Variants

Ordenació ràpida amb la mediana de tres com a pivot

```

template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int centre = (e + d) / 2;
        if (T[e] < T[centre]) swap(T[centre], T[e]);
        if (T[d] < T[centre]) swap(T[centre], T[d]);
        if (T[d] < T[e]) swap(T[e], T[d]);
        // el pivot es a la posició e
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}

```

Després de les línies 1, 2 i 3, tenim

$$T[\text{centre}] \leq T[e], T[\text{centre}] \leq T[d], T[e] \leq T[d].$$

Per tant,  $T[\text{centre}] \leq T[e] \leq T[d]$  i la mediana és  $T[e]$ .

Per a vectors molt petits, l'ordenació per **inserció** es comporta millor que **quicksort**. Una bona solució, doncs, és tallar la recursió quan el vector és més petit que una certa mida (normalment, entre 5 i 20).

Ordenació ràpida amb inserció per a vectors petits

```

template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    const int talla_critica = 20;
    if (d - e < talla_critica)
        ordena_insercio(T, e, d);
    else {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}

```

Recurrència

Sigui  $T(n)$  el cost de l'algorisme d'ordenació ràpida amb  $n$  elements.  
Llavors,

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

on  $i$  és el nombre d'elements de la primera meitat i  $n - i$  de la segona.

Cas pitjor

$$T(n) = T(n-1) + \Theta(n)$$

$$T(n) \in \Theta(n^2).$$

## Algorisme de Karatsuba

Conjectura (Kolmogorov, 1952)

Qualsevol algorisme per multiplicar dos nombres de  $n$  díigits té cost  $\Omega(n^2)$ .

Un estudiant de 23 anys de Kolmogorov, Anatolii Alexeevitch Karatsuba, va trobar un algorisme de cost  $\Theta(n^{1.585})$ .

Refutació (Karatsuba, 1960)

Hi ha un algorisme que multiplica dos nombres de  $n$  díigits en temps

$$\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

Exemple

Si  $x = 10010111_2$  i  $y = 11001010_2$  (el subíndex 2 vol dir "en binari"), llavors

$$x = \boxed{x_E} \boxed{x_D} = \boxed{1001}_2 \boxed{0111}_2$$

$$y = \boxed{y_E} \boxed{y_D} = \boxed{1100}_2 \boxed{1010}_2$$

Com abans, observem que

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

Si ara anomenem

$$a = x_E y_E, \quad b = x_D y_D, \quad c = (x_E + x_D)(y_E + y_D),$$

llavors el producte per a  $n$  parell (l'equació 11)

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

es pot reescriure com

$$2^n a + 2^{n/2}(c - a - b) + b$$

que només depèn de 3 subproductes (com el cas de  $n$  senar).

La nova expressió dona lloc a un algorisme de cost

$$T(n) = 3T(n/2) + \Theta(n)$$

i, pel teorema mestre de recurrències divisores, sabem que

$$T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

## Algorisme de Karatsuba

### Solució de Karatsuba (cas parell, base 10)

Donat  $n$  parell,  $x = (10^{n/2}x_E + x_D)$  i  $y = (10^{n/2}y_E + y_D)$ , tenim que

$$xy = 10^n a + 10^{n/2}(c - a - b) + b$$

on  $a = x_E y_E$ ,  $b = x_D y_D$ ,  $c = (x_E + x_D)(y_E + y_D)$ .

Exemple: calcular  $1234 * 4321$

- Problema: calcular  $x * y$  per a  $x = 1234$ ,  $y = 4321$

- Subproblemes:

- ① Calcular  $a = 12 * 43$
- ② Calcular  $b = 34 * 21$
- ③ Calcular  $c = (12 + 34) * (43 + 21) = 46 * 64$

Subproblema 1: calcular  $a = 12 * 43$

- Subproblemes:

- ① Calcular  $a_1 = 1 * 4 = 4$
  - ② Calcular  $b_1 = 2 * 3 = 6$
  - ③ Calcular  $c_1 = (1 + 2) * (4 + 3) = 21$
- Solució:  $a = 10^2 * 4 + 10 * (21 - 4 - 6) + 6 = 516$

Subproblema 2: calcular  $b = 34 * 21$

- Subproblemes:

- ① Calcular  $a_2 = 3 * 2 = 6$
  - ② Calcular  $b_2 = 4 * 1 = 4$
  - ③ Calcular  $c_2 = (3 + 4) * (2 + 1) = 21$
- Solució:  $b = 10^2 * 6 + 10 * (21 - 6 - 4) + 4 = 714$

Subproblema 3: calcular  $c = 46 * 64$

- Subproblemes:

- ① Calcular  $a_3 = 4 * 6 = 24$
  - ② Calcular  $b_3 = 6 * 4 = 24$
  - ③ Calcular  $c_3 = (4 + 6) * (6 + 4) = 100$
- Solució:  $c = 10^2 * 24 + 10 * (100 - 24 - 24) + 24 = 2944$

### Resultat

- Problema: calcular  $x * y$  per a  $x = 1234$ ,  $y = 4321$

- Subproblemes:

- ①  $a = 12 * 43 = 516$
- ②  $b = 34 * 21 = 714$
- ③  $c = 46 * 64 = 2944$

- Solució:  $10^4 * 516 + 10^2 * (2944 - 516 - 714) + 714 = 5332114$

## Exponenciació ràpida

Definició recursiva de  $x^n$

$$x^n = \begin{cases} 1, & \text{si } n = 0 \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor}, & \text{si } n > 0 \text{ i parell} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x, & \text{si } n \text{ és senar} \end{cases}$$

### Exemple

Amb un algorisme basat en la definició anterior, tindriem

$$x^{62} = (\cancel{x^{31}})^2, x^{31} = (\cancel{x^{15}})^2 \cdot x, x^{15} = (\cancel{x^7})^2 \cdot x$$

$$x^7 = (\cancel{x^3})^2 \cdot x, x^3 = (\cancel{x^1})^2 \cdot x, x^1 = (\cancel{x^0})^2 \cdot x, x^0 = 1$$

(en blau, els valors calculats recursivament)

### Exponenciació ràpida

```
double potencia (double x, int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        double y = potencia (x, n / 2);  
        if (n % 2 == 0) return y * y;  
        else return y * y * x;  
    } }
```

El càlcul del cost és directe i ve donat per la recurrència

$$T(n) = T(n/2) + \Theta(1)$$

que, segons el teorema mestre de recurrències divisores, implica

$$T(n) \in \Theta(\log n).$$

## Algorisme de Strassen

Algorisme  $\Theta(n^3)$ , adaptat de *Data Structures and Alg. Analysis in C++*, M.A. Weiss.

```
matrix<int> producte_matrius  
(const matrix<int>& X, const matrix<int>& Y)  
{  
    int n = X.numrows();  
    matrix<int> Z(n, n, 0);  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            for (int k = 0; k < n; k++)  
                Z[i][j] += X[i][k] * Y[k][j];  
    return Z;  
}
```

$Z_{ij}$  és el producte de la fila  $i$ -èsima de  $X$  per la columna  $j$ -èsima de  $Y$ :

Una primera idea és que el producte de matrius es pot fer *per blocs*.

Dividim  $X$  i  $Y$  en quatre quadrants cadascuna:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Llavors, es pot veure que

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

### Exemple

Per fer el producte de  $X$  i  $Y$

$$X = \begin{bmatrix} 4 & 3 & 1 & 6 \\ 1 & 5 & 2 & 7 \\ 2 & 1 & 5 & 9 \\ 3 & 4 & 2 & 6 \end{bmatrix}, Y = \begin{bmatrix} 2 & 6 & 9 & 4 \\ 3 & 2 & 4 & 1 \\ 1 & 1 & 8 & 3 \\ 2 & 1 & 3 & 1 \end{bmatrix},$$

definim les vuit matrius  $2 \times 2$

$$\begin{aligned} A &= \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix}, E = \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix}, F = \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix}, \\ C &= \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}, D = \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix}, G = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, H = \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix}. \end{aligned}$$

i l'expressem en termes de les submatrius

$$\begin{aligned} XY &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix} = \\ &= \left[ \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \right] \\ &\quad \left[ \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \right] \end{aligned}$$

El cost  $T(n)$  és la suma de fer:

- vuit productes de matrius de mida  $n/2$ :  $8T(n/2)$
- quatre sumes de matrius de mida  $n/2$ :  $\Theta(n^2)$

Per tant, tenim la recurrència

$$T(n) = 8T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dona

$$T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3).$$

## Algorisme de Strassen

Però el nombre de productes es pot reduir a 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

on

$$P_1 = A(F - H), P_4 = D(G - E)$$

$$P_2 = (A + B)H, P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Fent servir la descomposició de Strassen, obtenim un algorisme amb cost

$$T(n) = 7T(n/2) + \Theta(n^2) \quad T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

## Torres de Hanoi

Algorisme

```
void hanoi(int n, int a, int b, int c){
    // descriu els moviments de n discs des d'a fins a b
    // fent servir c com a auxiliar
    if (n > 0) {
        hanoi(n-1, a, c, b);
        cout << a, b;
        hanoi(n-1, c, b, a);
    } }
```

Cost

$$T(n) = \text{nombre de moviments que fa } \text{hanoi}(n, 1, 2, 3).$$

Llavors,

$$T(n) = \begin{cases} 0, & \text{si } n = 0 \\ 2T(n-1) + 1, & \text{si } n > 0 \end{cases}$$

Solució asimptòtica

Aplicant el teorema mestre de recurrències subtractives, obtenim  $T \in \Theta(2^n)$ .

Solució exacta

Definim  $S(n) = T(n) + 1$  i l'escrivim sense dependre de  $T(n)$ :

- $S(0) = 1$
- $S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \text{ si } n > 0$

Ara,  $S(n)$  es resol directament i dona:  $S(n) = 2^n$  per a tot  $n \geq 0$ .

Per tant,

$$T(n) = 2^n - 1 \text{ per a tot } n \geq 0.$$

# Mediana

## Definició

La **mediana** d'una llista  $S$  de  $n$  nombres és l'element  $\lfloor n/2 \rfloor$ -èsim de  $\text{SORT}(S)$ , on  $\text{SORT}(S)$  és la llista dels elements de  $S$  ordenada creixentment.

## Exemple

La mediana de  $[15, 3, 34, 5, 10]$  és 10 perquè quan s'escriuen en ordre, és el nombre que queda al mig:

3 5 10 15 34

La mediana de  $[15, 3, 12, 34, 5, 10]$  també és 10 perquè la llista és

3 5 10 12 15 34

Característiques de la mediana:

- Sempre és un dels valors del conjunt de dades
- És menys sensible a les observacions atípiques (*outliers*). Per exemple:
  - 1 Donats els nombres 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 100  
(10 vegades 1 i una vegada 100),
    - la mitjana és 10
    - la mediana és 1
  - 2 Donats 2, 4, 6, 8, 10.000,
    - la mitjana és 2004
    - la mediana és 6

Per calcular la mediana, n'hi ha prou a ordenar els elements.

Es pot fer en temps  $\Theta(n \log n)$

$[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]$

$\Downarrow$

$[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]$

Però es fa més feina de la necessària:

només volem l'element del mig i no caldria ordenar la resta

$\{0, 3, 5, 3, 2, 1\}, 6, \{8, 10, 7, 12, 7, 9\}$

## Definició

Si  $S$  és una llista i  $k$  tal que  $1 \leq k \leq |S|$ , anomenem

$\text{SEL}(S, k)$

al  $k$ -èsim element més petit de  $S$ .

## Problema de selecció

Donada una llista  $S$  i un natural  $k$ ,  $1 \leq k \leq |S|$ , determinar  $\text{SEL}(S, k)$ .

La mediana d'una llista  $S$  de  $n$  nombres és  $\text{SEL}(S, \lfloor n/2 \rfloor)$ .

## Idea per a un algorisme

Per a cada nombre  $x$ , dividim la llista en 3 conjunts d'elements:

- els més petits que  $x$
- els que són iguals a  $x$
- els més grans que  $x$

Si tenim el vector

$$S = [2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1]$$

per a  $x = 5$  el dividim en

$$S_E = [2 \ 4 \ 1] \quad S_x = [5 \ 5] \quad S_D = [36 \ 21 \ 8 \ 13 \ 11 \ 20]$$

# Mediana

## Idea per a un algorisme

$$S_E = [2 \ 4 \ 1] \quad S_x = [5 \ 5] \quad S_D = [36 \ 21 \ 8 \ 13 \ 11 \ 20]$$

Suposem ara que volem el 8è element de  $S$

$$S = [2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1]$$

Sabem que serà el 3r element de  $S_D$  perquè  $|S_E| + |S_x| = 5$ .

$$S_E = [2 \ 4 \ 1] \quad S_x = [5 \ 5] \quad S_D = [36 \ 21 \ 8 \ 13 \ 11 \ 20]$$

Podem definir l'operador  $\text{SEL}(S, k)$  de manera recursiva:

$$\text{SEL}(S, k) = \begin{cases} \text{SEL}(S_E, k), & \text{si } k \leq |S_E| \\ x, & \text{si } |S_E| < k \leq |S_E| + |S_x| \\ \text{SEL}(S_D, k - |S_E| - |S_x|), & \text{si } k > |S_E| + |S_x| \end{cases}$$

# TAD Diccionari

## Diccionari

Anomenem **diccionari** (també *taula de símbols*, *associative array* o *map*) a una estructura de dades que conté un conjunt finit d'elements, cadascun dels quals amb un identificador únic anomenat **clau**.

### Operacions bàsiques:

- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element hi és

Cada element del diccionari és un parell:

element = (clau, informació)

## Nombre d'elements consultats en les operacions de diccionaris

cas pitjor/mitjà	assignar	esborrar	consultar
vector no ordenat	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
vector ordenat	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(\log n)$ , $\Theta(\log n)$
llista no ordenada	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
llista ordenada	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
taula de dispersió	$\Theta(n)$ , $\Theta(1)$	$\Theta(n)$ , $\Theta(1)$	$\Theta(n)$ , $\Theta(1)$
AVL	$\Theta(\log n)$ , $\Theta(\log n)$	$\Theta(\log n)$ , $\Theta(\log n)$	$\Theta(\log n)$ , $\Theta(\log n)$

En vectors, cal desplaçar els elements.

En estructures no ordenades, cal comprovar repetits.

## Exemple: escriure equivalències en anglès de paraules en català

L'iterador `it` apunta a un parell `<clau, valor>`, on la clau és una paraula en català i el valor, la traducció a l'anglès.

```
map<string, string> CatEng;
...
for (auto it : CatEng)
    cout << it -> first << ' ' = '' << it -> second << endl;
```

## Operacions

- **assignar**: afegir un element (clau, informació) al diccionari; si existia un element amb la mateixa clau, se sobreescrivia la informació
- **esborrar**: donada una clau, s'esborra l'element que té aquella clau; si no hi ha cap element amb la clau, no es fa res
- **consultar**: donada una clau, retorna una referència a la informació associada a la clau
- **talla**: retorna la talla del diccionari

## Variants de consultar

Considerarem variants de l'operació

- **consultar**: donada una clau, retorna una **referència a la informació** associada a la clau
- com ara
- **present**: donada una clau, retorna un **booleà** que indica si hi ha un element amb aquella clau
- **cerca**: donada una clau, retorna una **referència a l'element** amb aquella clau

En general, però, afegint operacions més complexes s'obtenen noves estructures de dades. Per exemple, afegint l'operació d'**extreure el mínim** dona lloc a les **cues amb prioritat**.

## Taules d'accés directe

En l'**adreçament directe**:

- Cada posició correspon a una clau
- Les operacions seran  $\Theta(1)$  en cas pitjor
- Es pot guardar la informació directament en les posicions de la taula corresponents a la seva clau, però cal indicar si l'espai és buit

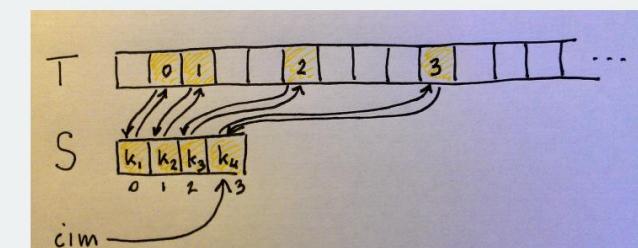
## Solució: idea

Definim un vector enorme  $T$  accessible per clau i un vector  $S$  amb tantes entrades com claus utilitzades ( $cim + 1$ ) tal que per a una clau  $k$ :

- $T[k]$  conté l'índex  $j$  d'una entrada vàlida de  $S$
- $S[j]$  conté  $k$

És a dir,  $S[T[k]] = k$  i  $T[S[j]] = j$  (en diem **cicle de validació**).

Definim també un vector  $S'$  que conté els objectes (la informació). Quan la clau  $k$  defineix un cicle de validació,  $S'[T[k]]$  conté l'objecte.



## Solució: operacions

- **inicialitzar**

```
cim = -1;
```

- **consultar.** Donada una clau  $k$ ,

```
if (S[T[k]] == k)
    return S'[T[k]];
else
    return NULL;
```

- **assignar.** Donat un objecte  $x$  amb clau  $k$ , si la clau no hi és,

```
++cim;
S[cim] = k;
S'[cim] = x;
T[k] = cim;
```

- **esborrar.** Donada una clau  $k$  (suposant que la clau hi és), hem d'assegurar que no queda un "forat" a  $S$ .

```
S[T[k]] = S[cim];
S'[T[k]] = S'[cim];
T[S[T[k]]] = T[k];
T[k] = 0;
--cim;
```

## Taules de dispersió

Les **taules de dispersió** (*hash tables*) són estructures de dades eficients per implementar els diccionaris.

Quan cal abandonar les taules d'accés directe.

- les claus no són nombres naturals o
- el nombre de claus utilitzades és petit en relació al nombre possible de claus o
- el nombre possible de claus és enorme,

En lloc de fer servir la clau com a índex per accedir a la taula, **l'índex es calcula a partir de la clau**.

## Qüestió

Per què els dos últims díigits i no els dos primers?

La biblioteca podria rebre la visita d'un grup d'amics amb números de carnet semblants, com ara

001523  
001525  
001531  
001570

Però no és tan probable que en algun moment molts usuaris tinguin els dos últims díigits idèntics.

Suposem que volem emmagatzemar un màxim de  $n$  claus. Declarem una taula  $T$  de  $m \leq n$  posicions.

- Si  $m = n$  i les claus són  $\{0, 1, \dots, m-1\}$ , usem **adreçament directe**: l'element de clau  $k$  va a l'espai  $T[k]$ .
- Si  $m < n$ , usem **taules de dispersió**: l'element de clau  $k$  va a l'espai  $T[h(k)]$ , on

$$h : K \rightarrow \{0, 1, \dots, m-1\}$$

és la **funció de dispersió** (*hash function*) i  $h(k)$ , el **valor de dispersió** de  $k$ .

La situació en què dues claus tenen el mateix valor de dispersió i, per tant, coincideixen en el mateix espai (com  $k_2$  i  $k_5$ ) se'n diu **col·lisió**.

## Col·lisions

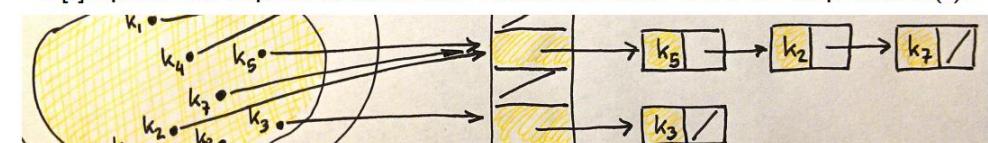
Resolució de col·lisions per **encadenament**. Cada entrada  $T[j]$  conté una **llista encadenada** amb les claus amb valor de dispersió  $j$ .

Per exemple,  $h(k_1) = h(k_4)$  i, per tant,  $T[h(k_1)]$  apunta a  $k_1$  seguit de  $k_4$ .

## Encadenament

En l'**encadenament**, els elements que tenen el mateix valor de dispersió es posen en una llista encadenada:

$T[i]$  apunta al cap de la llista dels elements amb valor de dispersió  $h(i)$ .



El cost de fer **una consulta** és

- $\Theta(n)$  en el **cas pitjor**. És el cas en què tots els elements tenen el mateix valor de dispersió i formen una sola llista.
- $\Theta(1)$  en el **cas mitjà** assumint que:
  - a cada crida amb una nova clau,  $h$  retorna un natural entre  $0$  i  $m-1$  a l'atzar, independent de les crides anteriors i
  - el cost de **calcular  $h$**  és  $\Theta(1)$ .

```

class Dictionary {
    private:

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

    vector<List> t; // Taula de dispersio
    int n;          // Nombre de claus
    int M;          // Nombre de posicions

    Classe Dictionary: implementació

    public:

    Dictionary (int M = 1009)
    : t(M), n(0), M(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;
        }
    }

    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;
        }
    }

    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";
    }

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

    int size () {
        return n;
    }
}

```

El cas pitjor de les operacions assign, erase, query i contains és  $\Theta(n)$ .  
 El cost mitjà és  $\Theta(1 + n/M)$ .

## Encadenament: anàlisi del cas pitjor

### Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

- $|K| > m?$  2
- $|K| > 2m?$  3
- $|K| > 3m?$  4
- $|K| > nm?$   $n+1$

### Proposició

El cost en cas pitjor de fer una cerca (find) en l'esquema d'encadenament és  $\Theta(n)$ .

### Corol·lari

El cost en cas pitjor de fer una assignació, un esborrat o una consulta (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(n)$ .

## Encadenament

### Exemple

Volem emmagatzemar la informació de matrícula dels alumnes:

- Si un nom té  $\leq 20$  caràcters, l'espai de possibles claus és de  $\approx 27^{20}$
- El nombre màxim d'alumnes nous és de  $n = 300$

Definim un vector de  $m = 300$  posicions, de manera que, en mitjana, cada llista de sinònims tindrà talla  $\approx 1$  i les operacions, cost  $\Theta(1)$ .

Però com triem la funció de dispersió?

Els costos previs depenen del cost de fer una cerca, que és  $\Theta(n)$  en cas pitjor i  $\Theta(1 + n/M)$  en mitjana.

### private:

```

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

```

## Funcions de dispersió

### Exemple

Donada una cadena de caràcters, la interpretarem com un natural.

Donada la cadena CLRS:

- valors en ASCII: C= 67, L= 76, R= 82, S= 83
- hi ha 128 caràcters en ASCII
- per tant, CLRS es transforma en el natural

$$67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 \\ = 141.764.947$$

### El mètode de la divisió

Dispersem una clau  $k$  en una de les  $m$  posicions a través de la funció

$$h(k) = k \bmod m$$

- **Exemple:** si la taula té mida  $m = 12$  i  $k = 100$ , llavors  $h(k) = 4$
- **Avantatge:** és força ràpid
- **Desavantatge:** cal evitar certs valors de  $m$  com  $2^i$
- **Bones tries per a  $m$ :** primers no gaire propers a potències de 2

### Exemple 1

Volem una taula de dispersió amb les col·lisions resoltres per encadenament, que emmagatzemi unes  $n = 2000$  cadenes de caràcters. No ens fa res examinar uns 3 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 701.$$

Triem 701 perquè és un primer  $\approx 2000/3$  i no és proper a una potència de 2.

La funció de dispersió serà

$$h(k) = k \bmod 701.$$

### El mètode de la multiplicació

Per dispersar una clau  $k$  en una de les  $m$  posicions:

- 1 multipliquem  $k$  per una constant  $A$  t.q.  $0 < A < 1$  i n'extraiem la part fraccionaria
- 2 llavors, multipliquem el valor per  $m$  i prenem la part baixa

$$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- **Avantatge:** el valor de  $m$  no és crític
- **Desavantatge:** és més lent que el mètode de divisió
- **Bones tries per a  $m$ :** potències de 2 (fan la implementació fàcil)
- **Bona tria per a  $A$ :** Knuth suggerix  $A \approx (\sqrt{5} - 1)/2 = 0,6180339887\dots$

## Adreçament obert

Una alternativa a l'encadenament és l'**adreçament obert**:

- tots els elements s'emmagatzemen en la mateixa taula
- quan cerquem un element, examinem les posicions de manera sistemàtica fins a trobar-lo
- la funció de dispersió té dos paràmetres: la clau i la "prova" de posició

$$h : K \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

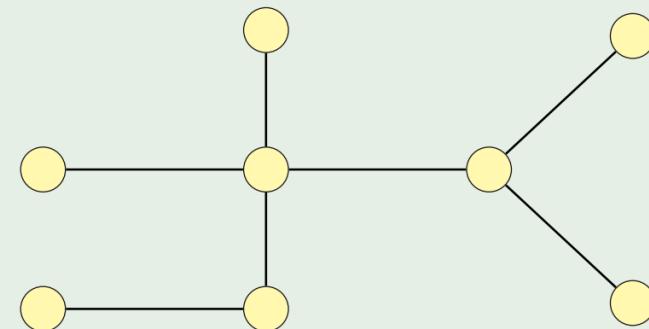
Per a una clau  $k$ , la seqüència de proves és

$$(h(k, 0), h(k, 1), \dots, h(k, m-1)).$$

### Definició

Un **arbre** és un graf connex i acíclic.

### Exemple: arbre



### Exemple

En una taula de dispersió amb  $m = 1024$ ,  $k = 123$  i  $A = 0,61803399$ , tenim

$$h(k) = \lfloor 1024 \cdot (123 \cdot A - \lfloor 123 \cdot A \rfloor) \rfloor$$

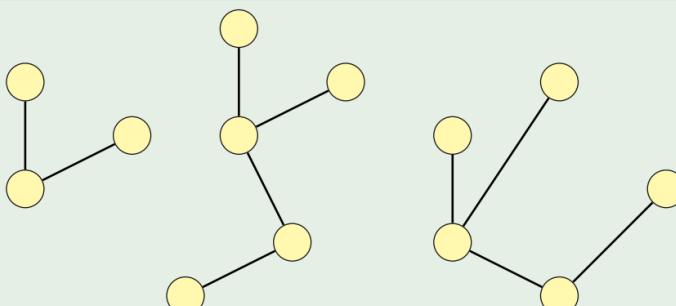
$$= \lfloor 1024 \cdot (76,0181808 - 76) \rfloor$$

$$= \lfloor 1024 \cdot 0,0181808 \rfloor = 18.$$

## Definició

Un **bosc** és un graf acíclic.

Exemple: bosc



## Teorema

Sigui  $G = (V, E)$  un graf i siguin  $n = |V|$  i  $m = |E|$ .

Les afirmacions següents són equivalents:

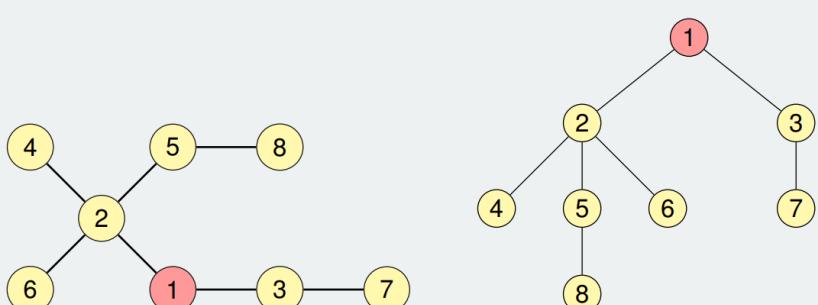
- ①  $G$  és un **arbre**
- ② Tot parell de vèrtexos de  $G$  estan units per un **camí únic**
- ③  $G$  és connex i  $m = n - 1$
- ④  $G$  és connex però, si s'hi elimina una aresta, s'obté un graf inconnex
- ⑤  $G$  és acíclic i  $m = n - 1$
- ⑥  $G$  és acíclic però, si s'hi afegeix una aresta, s'obté un graf cíclic

## Definició

Un **arbre arrelat** és un arbre amb un vèrtex distingit (**l'arrel**).

## Representació

Representarem els arbres arrelats amb **l'arrel a dalt**.

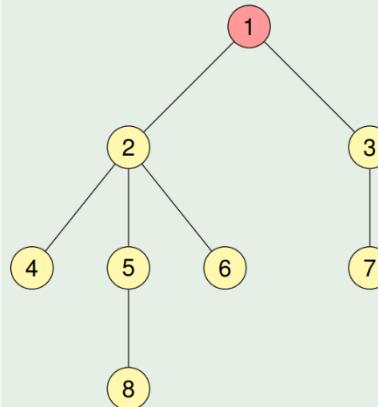


## Terminologia

En aquest tema, per **arbre** entendrem **arbre arrelat**.

## Definicions i terminologia

Exemple



- l'**arrel** és 1
- 3 és un **node intern**
- 7 és una **fulla**
- 2 és **pare** de 5
- 8 és **nét** de 2
- els **predecessors** de 8 són 5, 2 i 1
- els **successors** de 2 són 4, 5, 6 i 8
- 4, 5, 6 són **germans**
- l'**arbre** té **alçària** 3

## Introducció

La propietat dels ABC permet implementar altres operacions com

- ① fer la **llista ordenada** dels elements en temps  $\Theta(n)$
- ② trobar el **mínim** o el **màxim** en temps mitjà  $\Theta(\log n)$
- ③ trobar l'**anterior** o el **següent** d'un element donat en temps mitjà  $\Theta(\log n)$

## Definició

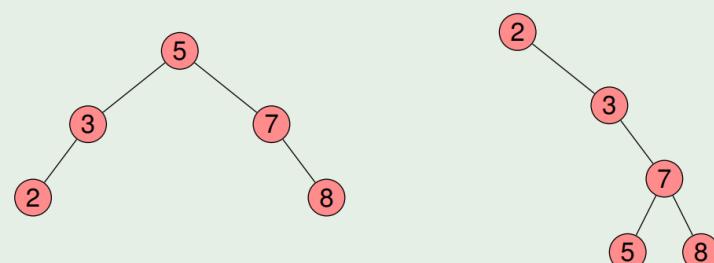
Un **arbre binari** és un arbre arrelat on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

## Definició

Un **arbre binari de cerca** (ABC, *Binary Search Tree* o BST en anglès) és un arbre binari que té una clau associada a cada node i que compleix la propietat que la clau de cada node és

- més gran que la de tots els nodes del seu subarbre esquerre i
- més petita que la de tots els nodes del seu subarbre dret

Exemple



## Operacions dels diccionaris en els ABC:

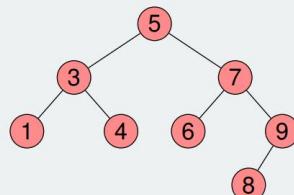
- Les **operacions bàsiques dels diccionaris** (**consultar, assignar, esborrar**) es poden fer en temps proporcional a l'alçària
- L'**alçària esperada** d'un ABC de  $n$  nodes és de  $\Theta(\log n)$
- Per tant, les operacions bàsiques tenen un **cost mitjà** de  $\Theta(\log n)$
- L'**alçària màxima** d'un ABC de  $n$  nodes és de  $\Theta(n)$
- Per tant, les operacions bàsiques tenen un cost  **$\Theta(n)$  en cas pitjor**

### Llista ordenada dels elements d'un ABC

- 1 Per fer la llista ordenada dels elements d'un ABC, n'hi ha prou a fer un recorregut inordre de l'arbre:

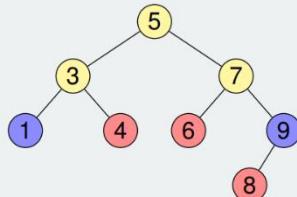
```
RECORREGUT-INORDRE(x)
  if x ≠ NUL llavors
    RECORREGUT-INORDRE(esquerre(x))
    escriure clau(x)
    RECORREGUT-INORDRE(dret(x))
```

El recorregut inordre de l'ABC següent és: 1, 3, 4, 5, 6, 7, 8, 9.



Com que cada node es visita un cop, el cost és  $\Theta(n)$ .

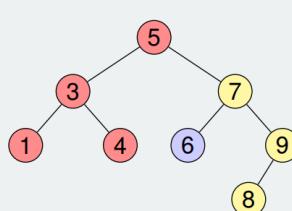
- 2 Mínim o màxim



El cost mitjà correspon a l'alçària esperada de l'arbre:  $\Theta(\log n)$ .

### Trobar l'anterior o el següent d'un element en un ABC

- 3 Següent de 5: el mínim del seu subarbre dret



Llavors, el cost mitjà és  $\Theta(\log n)$ .

## Implementació (ABC)

### Definició de Diccionari i Node

Un **Node** emmagatzema una clau, la seva informació associada i apuntadors a dos altres nodes.

```
template <typename Clau, typename Info>
class Diccionari {
private:
  struct Node {
    Clau clau;
    Info info;
    Node* fesq; // Apuntador al fill esquerre
    Node* fdre; // Apuntador al fill dret
  };
  Node (const Clau& c, const Info& i, Node* fe, Node* fd) : clau(c), info(i), fesq(fe), fdre(fd) { }
};

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

### Constructores de creació i còpia / Destructora

Crear Diccionari:  $\Theta(1)$ .

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

Fer una còpia:  $\Theta(n)$ .

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

Destructor:  $\Theta(n)$ .

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

### Constructora d'assignació

Redefinició de l'assignació:  $\Theta(n + d.n)$ .

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

## Assignar i esborrar

Assignar a clau el valor info:  $\Theta(n)$ .

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

Esborrar clau i la informació associada (si no hi és, no canvia):  $\Theta(n)$ .

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

## Consultes

Donada una clau, retornar la referència a la informació associada:  $\Theta(n)$ .

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

Indicar si la clau hi és o no present:  $\Theta(n)$ .

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

Retornar la talla del diccionari:  $\Theta(1)$ .

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

## Implementació (ABC): funcions privades

Suposarem que l'arbre a tractar (apuntat per p) té

- s nodes
- alçària h

Els costos en cas pitjor vindran donats per  $\Theta(s)$  o  $\Theta(h)$ .

## Esborrar

Eliminar l'arbre apuntat per p:  $\Theta(s)$ .

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

## Copiar

Retornar un apuntador a una còpia de l'arbre apuntat per p:  $\Theta(s)$ .

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

## Cerca

Retornar un apuntador al node de l'arbre apuntat per p que conté clau (o null si no hi és):  $\Theta(h)$ .

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

La cerca es fa descendint pels subarbres adequats fent ús de la propietat dels ABC.

## Assignar

Assignar info a clau si la clau és al subarbre apuntat per p; si no hi és, afegir un nou node amb clau i info:  $\Theta(h)$ .

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

## Mínim (recursiu)

Retornar un apuntador al node que conté el valor mínim en el subarbre apuntat per p (suposant que p no és nul):  $\Theta(h)$ .

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

## Màxim (iteratiu)

Retornar un apuntador al node que conté el valor màxim en el subarbre apuntat per p (suposant que p no és nul):  $\Theta(h)$ .

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

## Esborrar (1)

Esborrar el node que conté clau en el subarbre apuntat per p:  $\Theta(h)$ .  
Es penja el subarbre esquerre del mínim del subarbre dret.

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

## Esborrar (2)

Inconvenient: es copien claus i informació, més costós que copiar apuntadors.

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

## Esborrar (3)

Esborrar el node que conté clau en el subarbre apuntat per p:  $\Theta(h)$ .  
Es copia el mínim del subarbre dret al node esborrat i després s'esborra.

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

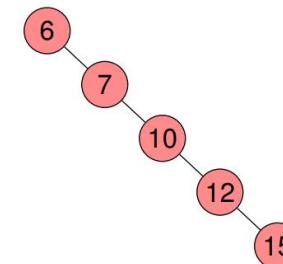
## Esborrar mínim

Esborrar i retornar el node que conté l'element mínim de p:  $\Theta(h)$ .

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

## Introducció

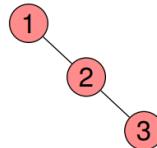
En els **arbres binaris de cerca**, hem vist que el cost de les operacions és  $\Theta(h)$ , on  $h$  és l'alçària. Però  $h$  pot coincidir amb el nombre de nodes:



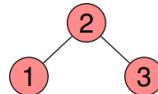
Per tant, el cost en cas pitjor és de  $\Theta(n)$ , on  $n$  és el nombre de nodes.

Per millorar el cost lineal es poden fer dues coses:

- demostrar que si en un ABC s'insereixen els elements de forma aleatòria, l'alçària és  $\approx \log n$
- fer les insercions i els esborrats de manera que l'alçària es mantingui en  $\approx \log n$ . En comptes de tenir



tindriem



Com mantenir els subarbres equilibrats?

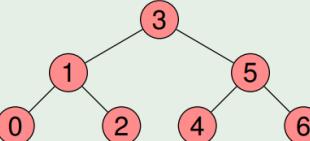
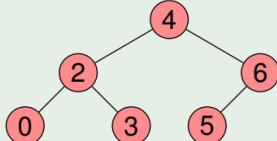
- 1 Forçant que l'ABC sigui **complet**.

Un arbre complet és aquell que té tots els nivells plens tret, potser, de l'últim, on els nodes són el més a l'esquerra possible.

Però afegir un element pot ser massa costós.

#### Exemple

Per afegir l'element 1 en el primer arbre, cal canviar-ho gairebé tot.



Com mantenir els subarbres equilibrats?

- 3 Permetent un **petit desequilibri** en les alçàries dels subarbres esquerre i dret: una diferència màxima d'1.

Però cal fer-ho per a tots els nodes!

#### Definició

Un arbre binari està **equilibrat** si

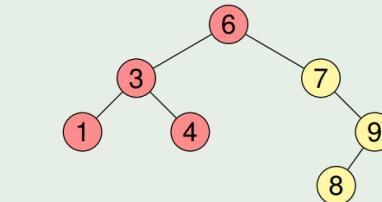
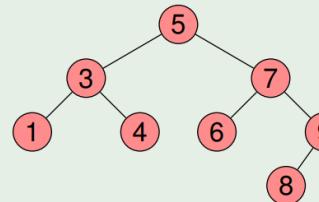
- és buit o
- la diferència d'alçàries dels seus subarbres és com a màxim d'1 i els seus subarbres també estan equilibrats

## Introduction

Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

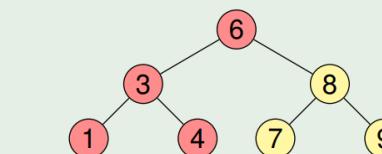
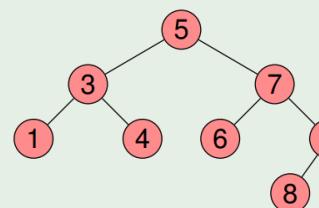
#### Exemple

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat.



Subarbre desequilibrat en groc

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat. **Els AVL usen "rotacions" per solucionar-ho.**



Rotació aplicada al subarbre groc

## Implementació (AVL)

```
class Diccionari {  
private:  
    struct Node {  
        Clau clau;  
        Info info;  
        Node* fesq; // Apuntador al fill esquerre  
        Node* fdre; // Apuntador al fill dret  
        int alc; // Alçària de l'arbre  
        Node (const Clau& c, const Info& i,  
              Node* fe, Node* fd, int a)  
            : clau(c), info(i), fesq(fe), fdre(fd), alc(a) {} };  
  
    int n; // Nombre d'elements en l'AVL  
    Node* arrel; // Apuntador a l'arrel de l'AVL
```

Les funcions públiques i les privades **alliberar**, **copia** i **cerca** són iguals que en els ABC.

En els AVL necessitarem dues funcions senzilles referents a l'alçària, totes dues de cost  $\Theta(1)$ .

Retornar i actualitzar l'alçària

```
static int alcaria (Node* p) {  
    return p ? p->alc : -1;  
}  
  
static void actualitzar_alcaria (Node* p) {  
    p->alc = 1 + max(alcaria(p->fesq), alcaria(p->fdre));  
}
```

Per tal de mantenir equilibrat un arbre després d'una assignació o esborrat, considerem quatre **rotacions** de l'arbre: EE, DD, ED i DE.

Totes quatre tenen cost  $\Theta(1)$ .

Assignar

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

En **mitjana**, es fa una rotació cada dues assignacions.

En el **cas pitjor**:

- **assignar** i **esborrar** fan  $\Theta(\log n)$  rotacions, on una rotació costa  $\Theta(1)$   
 $\implies \Theta(\log n)$
- el cost de **consultar**, com en els ABC, és  $\Theta(\log h)$ , on  $h$  és l'alçària de l'arbre  $\implies \Theta(\log n)$

Proposició

En els AVL, les operacions **assignar**, **esborrar** i **consultar** tenen cost en cas pitjor  $\Theta(\log n)$ .

## Alçària d'un AVL

Veurem el fet següent sobre AVLs.

Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

Per demostrar-lo, definim

$$n_k = \text{mínim nombre de nodes d'un AVL d'alçària } k$$

Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{k \text{ senar}} \geq \underbrace{2^{\frac{k}{2}}}_{k \text{ parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

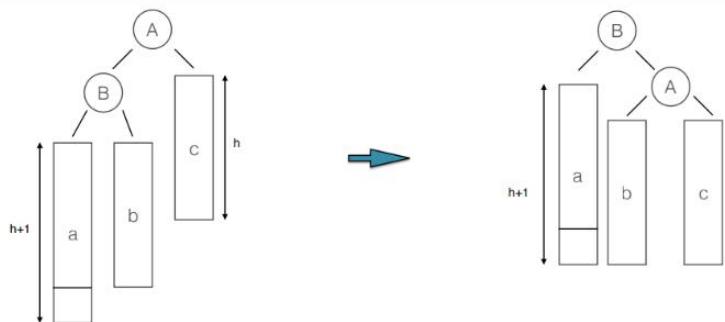
$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

La definició dels nombres de Fibonacci és:

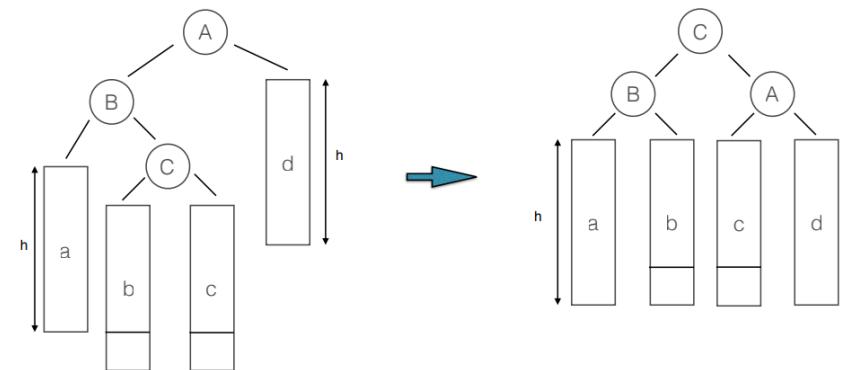
- $F_0 = 1$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$  for  $k > 1$

## Implementació (AVL)



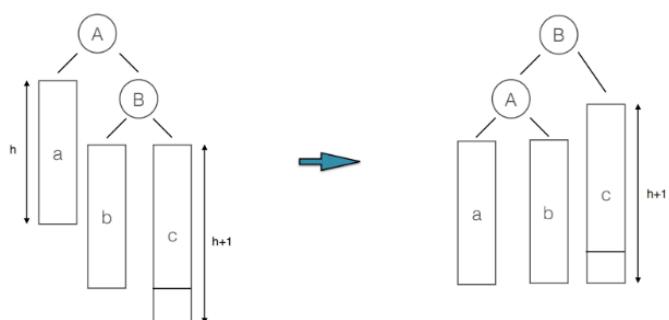
Rotació EE (LL, en anglès)

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



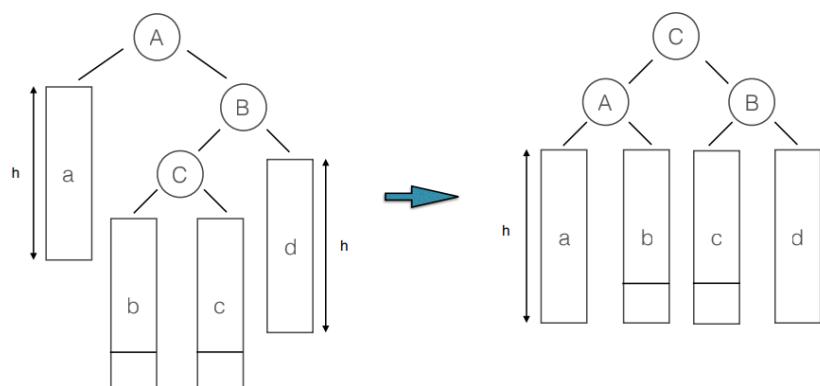
Rotació ED (LR, en anglès)

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



Rotació DD (RR, en anglès)

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



Rotació DE (RL, en anglès)

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

(68 of unbalanced node, key we add)

2. if  $b < -1$  and key  
LEFT(root)

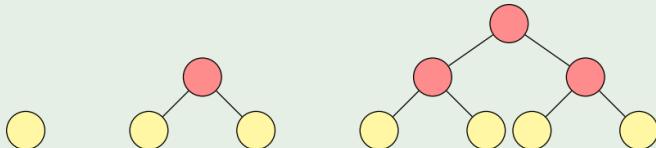
LEFT:

# Arbres binaris perfectes

## Definicions

- El **nivell** d'un node en un arbre és la distància de l'arrel al node.
- Un **arbre binari és perfecte** si totes les fulles són al mateix nivell (per tant, tots els nodes interns tenen dos fills).

## Exemples



## Definicions

- L'**alçària** d'un node és la distància màxima del node a una fulla.
- L'**alçària** d'un arbre és l'alçària de l'arrel (o el nivell màxim dels nodes).

## Proposició

Un arbre binari perfecte d'alçària  $h$  té  $2^{h+1} - 1$  nodes.

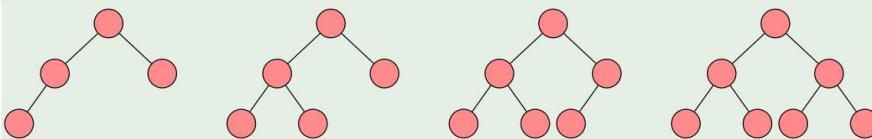
# Arbres binaris complets

## Definició

Un **arbre binari** d'alçària  $h$  és **complet** si

- 1 hi ha tots els nodes possibles amb nivells  $0 \dots h - 1$
- 2 tots els nodes de nivell  $h$  són el màxim a l'esquerra

## Exemple: arbres binaris complets d'alçària 2



## Proposició

Un arbre binari complet d'alçària  $h$  té entre  $2^h$  i  $2^{h+1} - 1$  nodes.

## Corol·lari

L'alçària d'un arbre binari complet de  $n$  nodes és  $\lfloor \log n \rfloor \in \Theta(\log n)$ .

## Definició

Una **cua amb prioritat** és una estructura de dades que disposa de dues operacions bàsiques:

- **afegir**: inserir un element (clau més informació)
- **treure\_min (treure\_max)**: esborrar i retornar l'element amb la clau més petita (més gran)

## Descripció

- La implementació fa servir *heaps*
- Per defecte, *max-heaps* (clau més alta disponible amb cost  $\Theta(1)$ )
- Mètodes: push, pop, top, empty, size

## Exemple: max-heap

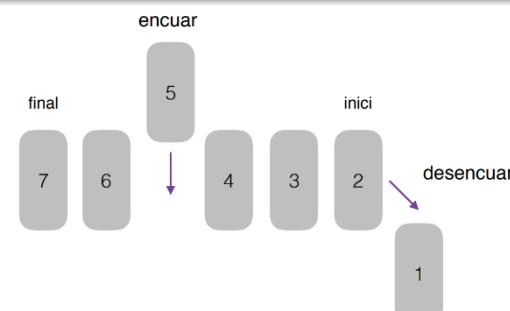
```
#include <queue>
int main() {
    priority_queue<int> Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

S'obté 5 al canal de sortida.

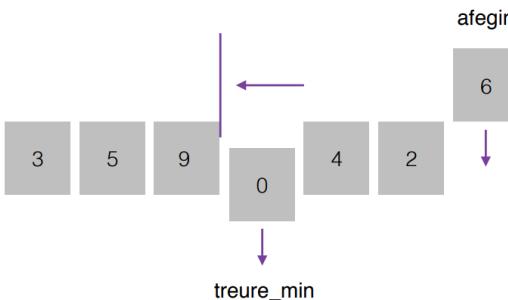
## Exemple: min-heap

```
#include <queue>
int main() {
    priority_queue<int, vector<int>, greater<int> > Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

S'obté 3 al canal de sortida.



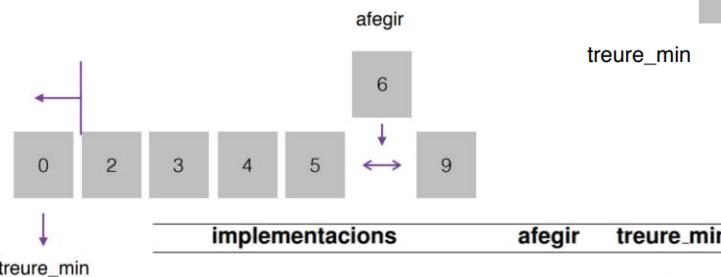
implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



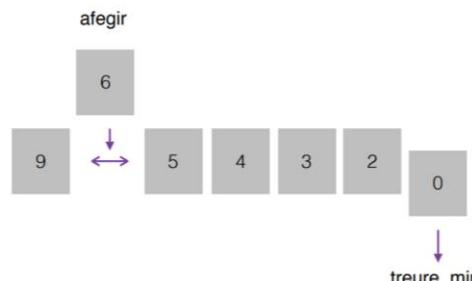
implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$

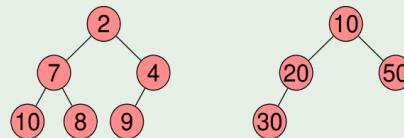


### Definició

Un *min-heap* és un arbre binari complet on la clau d'un node és sempre més petita que les claus dels seus fills.

- Quan parlem de *heaps* sense especificar res més, ens referirem als *min-heaps*

Són *min-heaps*:



No són *min-heaps*:

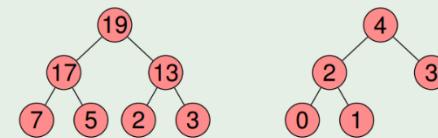


### Definició

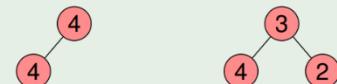
Un *max-heap* és un arbre binari complet on la clau d'un node és sempre més gran que les claus dels seus fills.

### Exemples

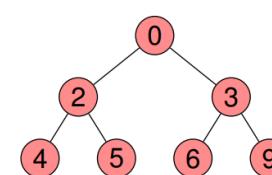
Són *max-heaps*:



No són *max-heaps*:

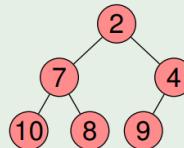


implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



## Representació d'un *heap* mitjançant vectors

El *heap*



es representa amb el vector

	2	7	4	10	8	9		
0	1	2	3	4	5	6	7	8

No calen apuntadors perquè per a un node en posició  $i$ :

- el **pare** és a la posició  $[i/2]$
- el **fill esquerre** és a la posició  $2i$
- el **fill dret** és a la posició  $2i + 1$

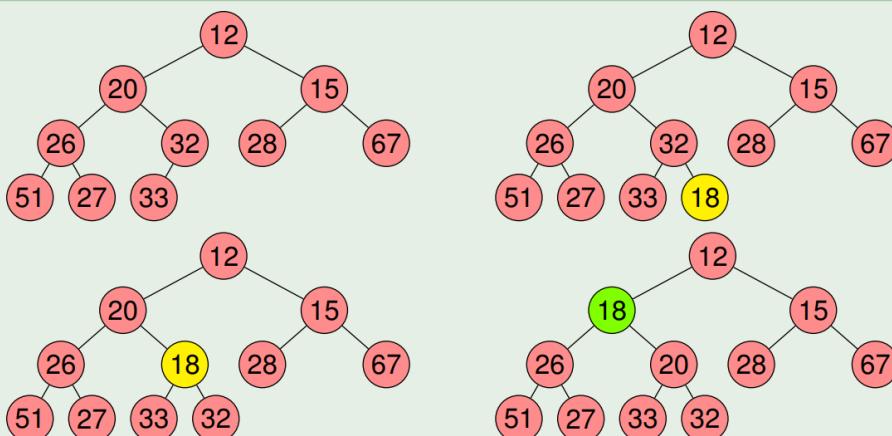
## Costos de les operacions en *heaps*

operacions	cas pitjor	cas mitjà
afegir	$\Theta(\log n)$	$\Theta(1)$
treure_min	$\Theta(\log n)$	$\Theta(\log n)$

## Operació afegir

S'afegeix l'element en la **següent posició lliure** del vector i **es fa ascendir** fins la posició en què es torna a complir la propietat del *heap*.

Exemple: afegir la clau 18



El *heap* es forma en la taula  $t$ . La posició 0 no s'utilitza.

```

template <typename Elemt>
class CuaPrio {
private:
vector<Elemt> t;
Crea una cua amb prioritat buida. Cost:  $\Theta(1)$ .
CuaPrio () {
    t.push_back(Elemt());
}
  
```

Retorna la talla de la cua amb prioritat. Cost:  $\Theta(1)$ .

```

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

Indica si la cua amb prioritat és buida. Cost:  $\Theta(1)$ .

```

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

Retorna un element amb prioritat mínima. Cost:  $\Theta(1)$ .

```

Elemt minim () {
    if (buida()) throw "CuaPrio_buida";
    return t[1];
}
  
```

Els costos asymptòtics no canvien:  $\Theta(\log n)$ .

## afegir

```

void afegir (Elemt& x) {
    t.push_back(x);
    int i = talla();
    while (i != 1 and t[i/2] > x) {
        t[i] = t[i/2];
        i = i/2;
    }
    t[i] = x;
}
  
```

## treure\_min

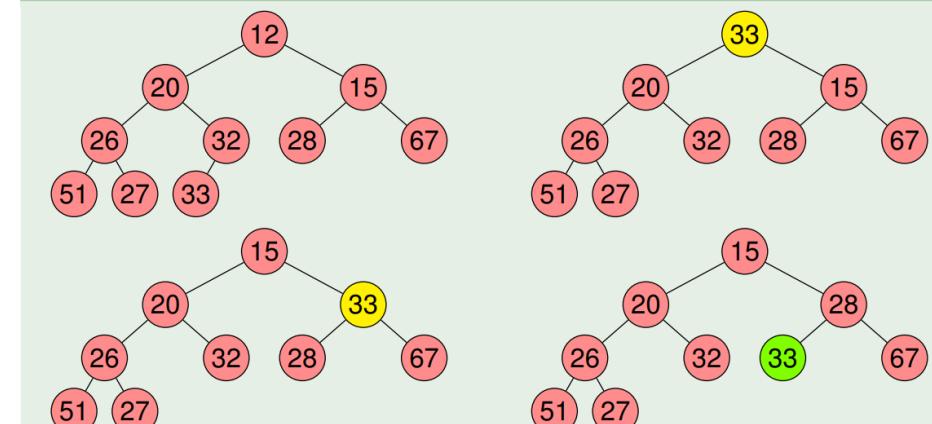
```

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

## Operació treure-min

L'element en l'**última posició** del vector **es trasllada** a la **primera** i **es fa descendir** fins que troba la seva posició. Es retorna l'antiga arrel.

Exemple: esborrar el mínim



**Heapsort** és un algorisme d'ordenació basat en les cues amb prioritat.

Donat un vector de  $n$  elements,

- ① afegeix els  $n$  elements a un *heap*:  $\Theta(n \log n)$
- ② fa  $n$  operacions **treure\_min** per construir un vector ordenat:  $\Theta(n \log n)$

El temps total és  $\Theta(n \log n)$ , el mínim asimptòtic per a un algorisme d'ordenació.

```
void heapsort (vector<elem>& v) {  
    n = v.size();  
    CuaPrio<elem> h;  
    for (int i = 0; i < n; ++i)  
        h.afegir(v[i]);  
    for (int i = 0; i < n; ++i)  
        v[i] = h.treure_min();  
}
```

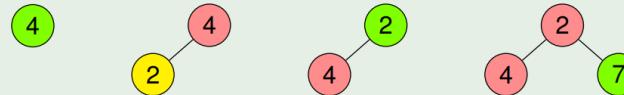
### Exemple

Suposem que partim del vector:

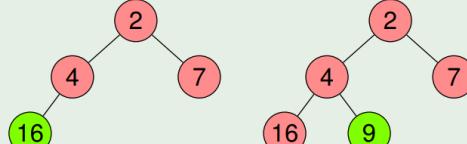
4	2	7	16	9	3	1	5
1	2	3	4	5	6	7	8

i afegim els elements a un *heap*, un per un.

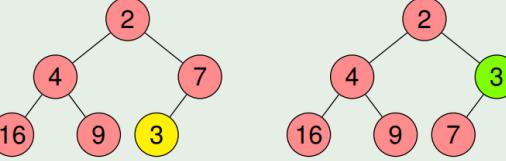
+4, +2, +7:



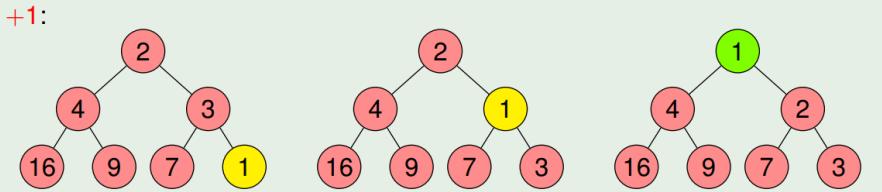
+16, +9:



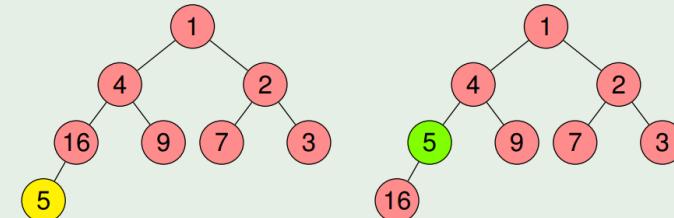
+3:



+1:



+5:

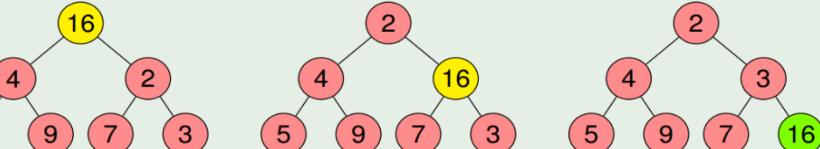


El *heap* resultant s'emmagaixza en el vector:

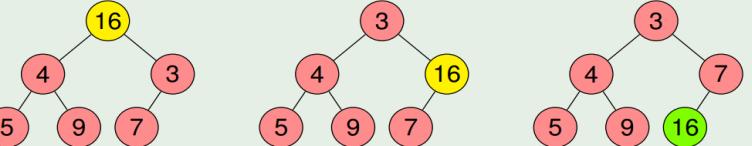
1	4	2	5	9	7	3	16
1	2	3	4	5	6	7	8

### Algorisme bàsic

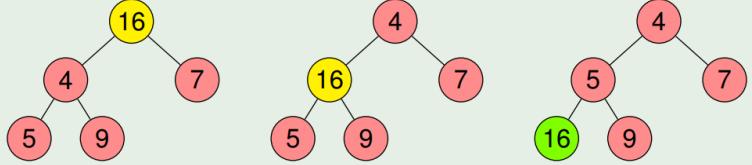
-1:



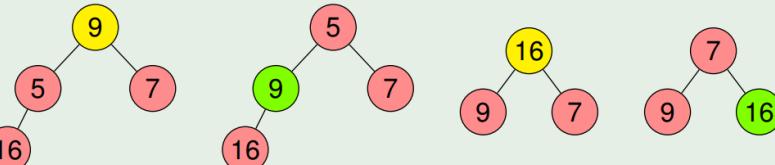
-2:



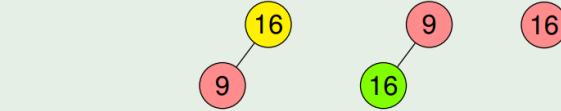
-3:



-4, -5:



-7, -9, -16:



## Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida								
	2	7	16	9	3	1	5	
1	2	3	4	5	6	7	8	
4								
heap								

entrada/sortida								
				3	1	5		
1	2	3	4	5	6	7	8	
2	4	7	16	9				
heap								

entrada/sortida								
						1	5	
1	2	3	4	5	6	7	8	
2	4	3	16	9	7			
heap								

## Exemple: evolució dels vectors (operació **treure\_min**)

entrada/sortida								
1	2	3	4	5	6	7	8	
1	4	2	5	9	7	3	16	
heap								

entrada/sortida								
1	2	3	4	5	7	9	16	
1	2	3	4	5	6	7	8	
heap								

## Funció buildHeap

Construir el *heap* en temps  $\Theta(n)$  en cas pitjor en lloc de  $\Theta(n \log n)$ :

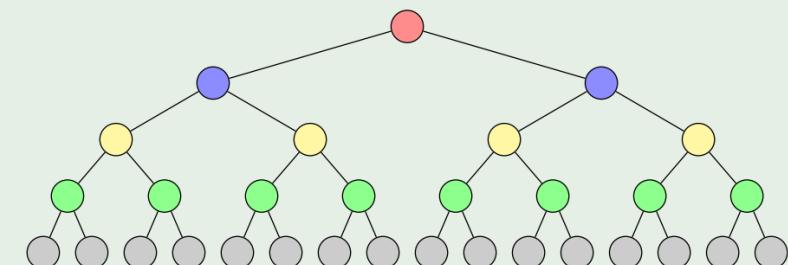
- ➊ Introduir els elements en el *heap* en qualsevol ordre (i temps lineal)
- ➋ Si el *heap* té  $h$  nivells, per a  $i = h - 1, h - 2, \dots, 1$ :
  - **enfonsar** tots els elements del nivell  $i$

El fet que la majoria de *subheaps* tractats siguin petits fa que el nombre d'intercanvis fets per **enfonsar** sigui lineal.

## Exemple

Per a un *heap* de 31 nodes, hi ha

- 8 *heaps* de mida 3 (arrel en verd)
- 4 *heaps* de mida 7 (arrel en groc)
- 2 *heaps* de mida 15 (arrel en blau)
- 1 *heap* de mida 31 (arrel en vermell)



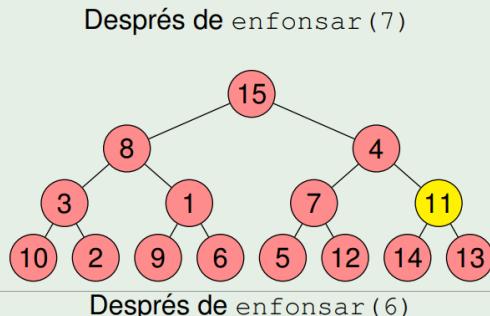
Constructor que pren els elements d'un vector com a entrada.

```
explicit PrioQueue (const vector<Elem>& v)
: t(v.size() + 1) {
  for (int i = 0; i < v.size(); ++i)
    t[i + 1] = v[i];
  buildHeap();
}
```

Establir propietat d'ordre del *heap* a partir d'una ordenació arbitrària d'ítems.

```
void buildHeap () {
  for (int i = size() / 2; i > 0; --i)
    enfonsar(i);
}
```

## Exemple de buildHeap



El temps de càlcul de `buildHeap` està fitat per la suma de les alçàries de tots els nodes.

Volem demostrar que aquesta suma és  $O(n)$ .

### Teorema

Per a l'arbre binari perfecte d'alçària  $h$  i  $2^{h+1} - 1$  nodes, la suma de les alçàries dels seus nodes és  $2^{h+1} - 1 - (h + 1)$ .

### Corol·lari

La suma d'alçàries d'un arbre binari complet de  $n$  nodes és  $O(n)$ .

### Problema de selecció

Donada una llista  $S$  de naturals i un  $k \in \mathbb{N}$ , determinar el  $k$ -èsim element més petit de  $S$ .

Fent servir *heaps*, podem trobar un nou algorisme:

- 1 Construir un *min-heap* a partir de  $S \rightarrow \Theta(n)$
- 2 Efectuar  $k$  operacions **treure\_min** del *min-heap*  $\rightarrow \Theta(k \log n)$
- 3 Retornar l'últim element extret  $\rightarrow \Theta(1)$

Cost total:  $\Theta(n + k \log n)$ .

La **mediana** correspon a  $k = n/2$ . Cost:  $\Theta(n \log n)$ .

En el cas  $k = \frac{n}{\log n}$ , el cost és  $\Theta(n)$ .

## Exemple: acolorir un mapa amb el mínim nombre de colors

Quin és el **mínim nombre de colors** necessari per acolorir un mapa de manera que els països veïns tinguin colors diferents?  
Però tot mapa el podem representar com un graf:

- Un **vèrtex** correspon a un país
- Una **aresta** correspon a una frontera

De fet, el graf és **planar**: es pot dibuixar sense que les arestes es tallin.

### Teorema dels quatre colors (Appel/Haken, 1976)

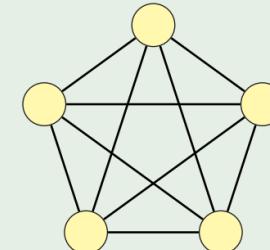
Tot graf planar es pot acolorir amb 4 colors.

Per tant, tot mapa es pot acolorir amb 4 colors

### Exemple: connectar cinc objectes en el pla

No es poden connectar 5 objectes sobre el pla sense creuar connexions.

En teoria de grafs, és el mateix que dir que el graf  $K_5$

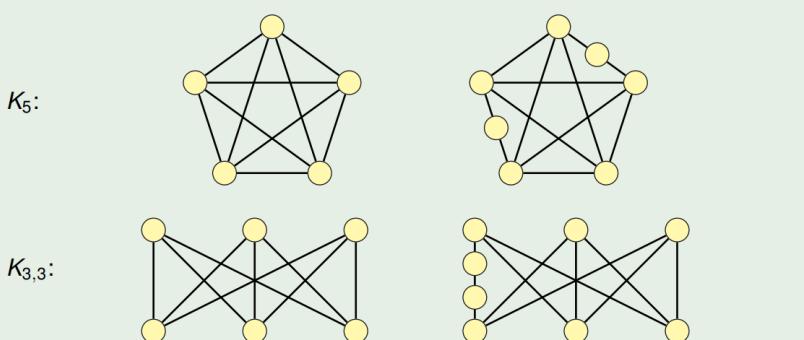


no es pot dibuixar en el pla sense creuar arestes (no és **planar**).

### Definició

Una **subdivisió** d'un graf és un graf format subdividint les seves arestes en camins d'una o més arestes.

### Exemple: subdivisions de $K_5$ i $K_{3,3}$

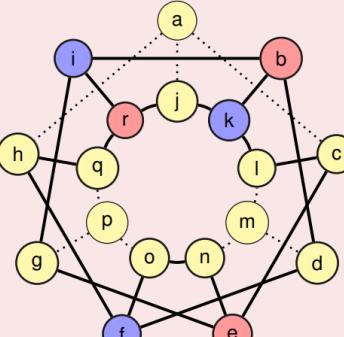


## Teorema (Kuratowski, 1922)

Un graf és planar si i només si no conté cap subgraf que sigui una subdivisió de  $K_5$  o  $K_{3,3}$ .

### Solució

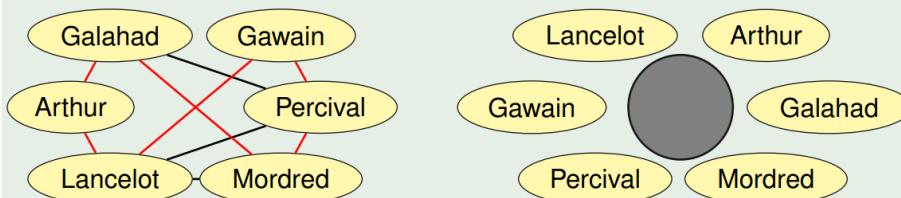
El subgraf destacat és una subdivisió de  $K_{3,3}$  (en blau i vermell).



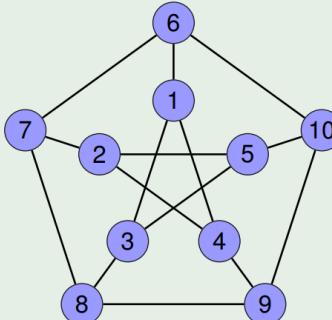
### Exemple: el rei Artur i la taula rodona

El rei Artur vol asseure els seus cavallers a la taula rodona però ha d'evitar que alguns d'ells seguin colze contra colze.

S'introduceix el graf en un algorisme que troba un cicle que passa per tots els vèrtexs (cicle Hamiltonià): aquest serà l'ordre en què hauran de seure.



### Exemple: graf de Petersen

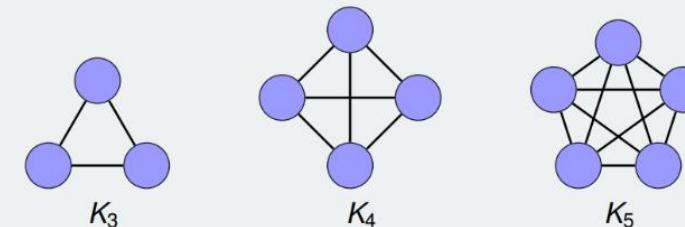


Formalment, és  $GP(5,2) = (V, E)$  on

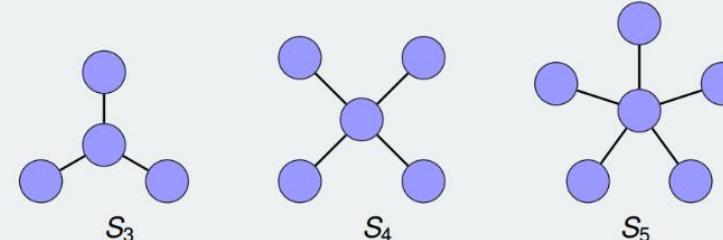
- $V = \{1, \dots, 10\}$
- $E = \{\{1,3\}, \{1,4\}, \{1,6\}, \{2,4\}, \{2,5\}, \{2,7\}, \{3,5\}, \{3,8\}, \{4,9\}, \{5,10\}, \{6,7\}, \{6,10\}, \{7,8\}, \{8,9\}, \{9,10\}\}$

## Tipus de grafs

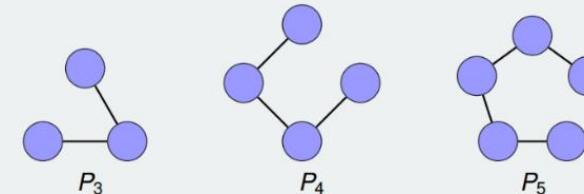
- **Complets:**  $K_i$  és el graf complet de  $i$  vèrtexs.



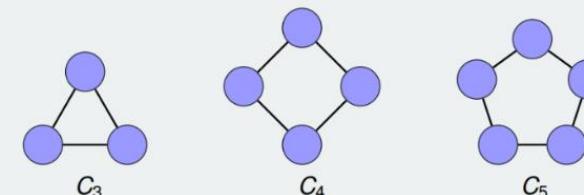
- **Estrelles:**  $S_i$  és l'estrella amb  $i + 1$  vèrtexs.



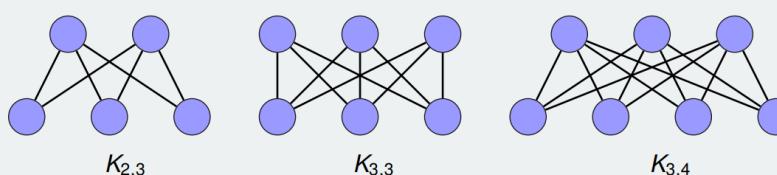
- **Camins:**  $P_i$  és el camí de  $i$  vèrtexs.



- **Cicles:**  $C_i$  és el cicle de  $i$  vèrtexs.



- **Bipartits complets:**  $K_{i,j}$  és el bipartit complet amb  $i$  vèrtexs connectats a  $j$  vèrtexs.

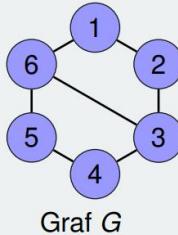


### Propietat

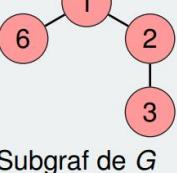
Tot graf de  $n$  vèrtexs té un màxim de  $\frac{n(n-1)}{2}$  arestes.

## Adjacència i subgrafs

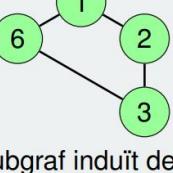
- dos vèrtexs  $u, v$  són **adjacents** si  $\{u, v\}$  és una aresta
- una aresta  $\{u, v\}$  es diu que és **incident** a  $u$  i  $v$
- **grau** d'un vèrtex  $u$ : nombre d'arestes incidents a  $u$
- un graf  $H = (V', E')$  és **subgraf** del graf  $G = (V, E)$  si  $V' \subseteq V$  i  $E' \subseteq E$
- un graf  $H$  és **subgraf induït** d'un graf  $G$  si  $H$  és subgraf de  $G$  i conté totes les arestes que  $G$  té entre els vèrtexs de  $H$



Graf  $G$



Subgraf de  $G$



Subgraf induït de  $G$

Pel **príncipi de les caselles**,  $G$  té dos vèrtexs amb el mateix grau.

## Camins i cicles

Sigui  $G = (V, E)$  un graf.

- Un **camí** en  $G$  és una seqüència de vèrtexs  $(x_0, x_1, \dots, x_n)$  on  $x_i \in V$  per a tot  $i \leq n$ ,  $\{x_i, x_{i+1}\} \in E$  per a tot  $i < n$  i tots els  $x_i$  són diferents
- Un **cicle** en  $G$  és una seqüència  $(x_0, x_1, \dots, x_n, x_0)$  tal que  $(x_0, x_1, \dots, x_n)$  és un camí en  $G$  i  $\{x_0, x_n\} \in E$

## Connectivitat

- Un graf és **connex** si existeix un camí entre tot parell de vèrtexs
- Un **component connex** d'un graf és un subgraf induït connex maximal (no hi ha cap vèrtex extern adjacent)

## Distància

- La **distància** entre dos vèrtexs és el nombre mínim d'arestes d'un camí que els uneix
- El **diàmetre** d'un graf és la màxima distància entre qualsevol parell de vèrtexs del graf

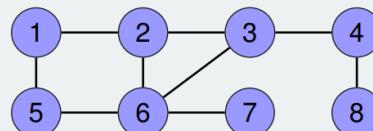


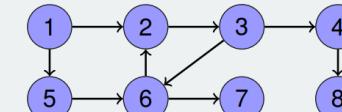
Figura: La distància entre 4 i 5 és 3. El diàmetre del graf és 4.

## Definició

- Un **graf dirigit** o **digraf** és un parell  $(V, E)$ , on
  - $V$  és un conjunt finit (**vèrtexs**)
  - $E$  és un conjunt de parells ordenats de vèrtexs (**arcs**)

## Digrafs: graus i distàncies

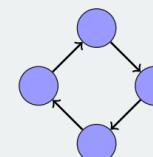
- Es distingeix entre **grau d'entrada** i **grau de sortida**
- En un **camí** (o **camí dirigit**), tots els arcs van en la mateixa direcció
- La **distància** entre dos vèrtexs es refereix als camins dirigits



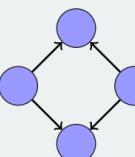
El vèrtex 2 té grau d'entrada 2 i grau de sortida 1.

La distància de 5 a 4 és 4. La de 4 a 5,  $\infty$ .

- Un digraf és **feblement connex** (o **connex**) si el graf obtingut substituint els arcs per arestes no dirigides és connex
- Un digraf és **fortament connex** si existeix un camí dirigit entre qualsevol parell de vèrtexs
- Si un digraf és fortament connex llavors també és feblement connex
- A la inversa no és cert:



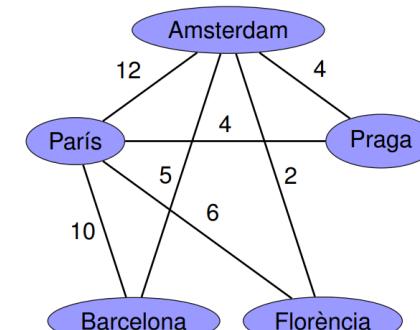
fortament connex



feblement connex

## Definició

Un **graf etiquetat** (dirigit o no dirigit) és un graf en el qual les arestes tenen etiquetes associades. També se'n diu **ponderat** o **graf amb pesos**.



Es fan servir les 4 combinacions:

- Grafs no dirigits no etiquetats
- Grafs no dirigits etiquetats
- Grafs dirigits no etiquetats
- Grafs dirigits etiquetats

## Densitat

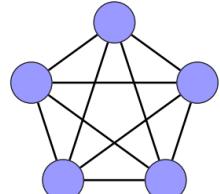
La **densitat** d'un graf  $G$  de  $n$  vèrtexs i  $m$  arestes és

$$D(G) = \frac{2m}{n(n-1)}$$

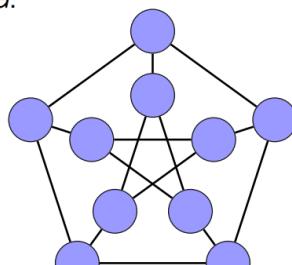
## Densitat

Un graf  $G$  de  $n$  vèrtexs i  $m$  arestes és **dens** si  $m \approx n^2/2$  (si  $D(G)$  és proper a 1). Altrament, se'n diu **espars**.

Hem vist que el nombre màxim d'arestes d'un graf  $G$  de  $n$  vèrtexs és  $\frac{n(n-1)}{2}$ . Per tant,  $0 \leq D(G) \leq 1$  per a tot graf  $G$ .



$$D(K_5) = 1$$



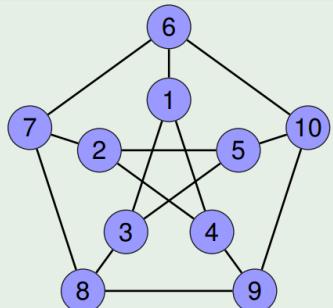
$$D(GP(5,2)) = 1/3$$

## Matriu d'adjacència / grafs no dirigits

La **matriu d'adjacència** d'un graf no dirigit  $G = (V, E)$  és una matriu  $M$  de  $n \times n$  valors booleans tal que

$$M_{ij} = \begin{cases} 1, & \text{si } \{i, j\} \in E \\ 0, & \text{si } \{i, j\} \notin E \end{cases}$$

## Exemple



$$1 \quad \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

## Matriu d'adjacència: estructura de dades

Les matrius d'adjacència es poden implementar amb un vector de vectors.

```
typedef vector< vector<bool> > graph;
```

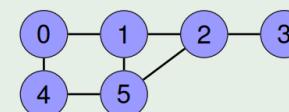
- La matriu d'adjacència és adequada per a **grafs densos**

## Llistes d'adjacència

Cada vèrtex apunta a la llista dels adjacents (grafs) o successors (digrafs).

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

## Exemple



0	1 4
1	0 5 2
2	3 1 5
3	2
4	5 0
5	1 2 4

- Les llistes d'adjacència són adequades per a **grafs esparsos**

## Cost de les operacions

$n$ : nombre de vèrtexs /  $m$ : nombre d'arestes

operacions	matriu d'adjacència	llistes d'adjacència
espai	$\Theta(n^2)$	$\Theta(n + m)$
crear	$\Theta(n^2)$	$\Theta(n)$
afegir vèrtex	$\Theta(n)$	$\Theta(1)$
afegir aresta	$\Theta(1)$	$O(n)$
esborrar aresta	$\Theta(1)$	$O(n)$
consultar vèrtex	$\Theta(1)$	$\Theta(1)$
consultar aresta	$\Theta(1)$	$O(n)$
és $v$ aïllat?	$\Theta(n)$	$\Theta(1)$
successors*	$\Theta(n)$	$O(n)$
predecessors*	$\Theta(n)$	$\Theta(n + m)$
adjacents <sup>+</sup>	$\Theta(n)$	$O(n)$

\* només en grafs dirigits

+ només en grafs no dirigits

## Cerca en profunditat

La **cerca** (o recorregut) **en profunditat** (en anglès: **DFS**, de *Depth-First Search*) resol la pregunta:

Per explorar un laberint, cal tenir guix i corda:

- El **guix** evita anar en cercles (saber què hem visitat) **vector de booleans**
- La **corda** permet anar enrere i veure passadissos encara no visitats **pila**

```
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);
    }
}
```

Cost total:  $O(n + m)$

## Cerca en profunditat recursiva

Recorregut en profunditat de tot el graf, encara que no sigui connex.

```
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;
} \Theta(n + m).
```

## Cerca en profunditat iterativa

```
list<int> DFS_ite (const graph& G) {
    int n = G.size();
    list<int> L;
    stack<int> S;
    vector<bool> 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;
}
```

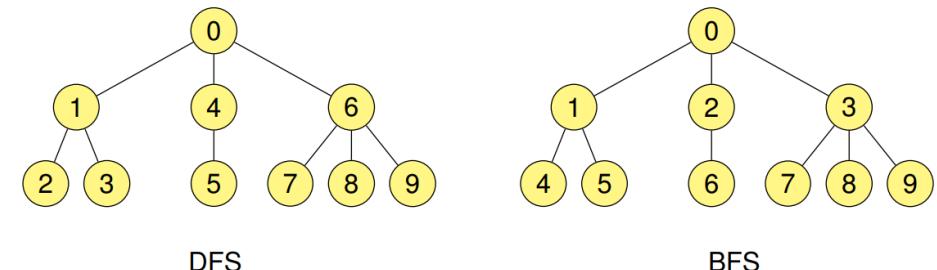
## Cerca en amplada

La **cerca** (o recorregut) **en amplada** (en anglès: **BFS**, de *Breadth-First Search*) **avanza localment** des d'un vèrtex inicial  $s$  visitant els vèrtexs a distància  $k + 1$  de  $s$  després d'haver visitat els vèrtexs a distància  $k$  de  $s$ .

- L'algorisme de cerca en amplada, a partir d'un vèrtex  $s$ , calcula
  - un **recorregut** en amplada a partir de  $s$
  - les **distàncies** mínimes de  $s$  a tots els vèrtexs
  - els **camins** mínims de  $s$  fins tots els vèrtexs

En arbres,

- DFS correspon al recorregut preordre i és la base del *backtracking*
- BFS recorre els vèrtexs per nivells



Els codis de DFS i BFS són molt semblants.

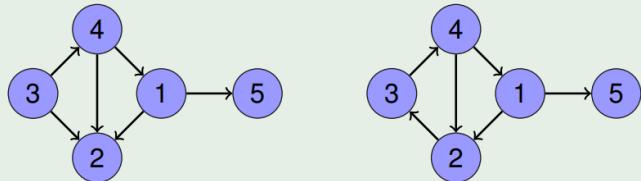
- La **diferència** fonamental és l'ús
  - d'una **pila** en DFS
  - d'una **caua** en BFS

```
list<int> BFS (const graph& G) {
    int n = G.size();
    list<int> L;
    queue<int> Q;
    vector<bool> 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 (w : G[v])
                    if (not enc[w])
                        Q.push(w); enc[w] = true;
            }
        }
    }
    return L;
} \Theta(n + m).
```

## Definició

Un **dag** és un graf dirigit acíclic.

## Exemple



El digraf de l'esquerra és un dag; el de la dreta, no.

## Definició

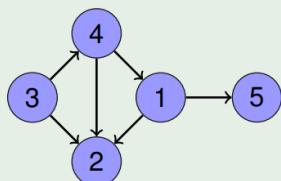
Una **ordenació topològica** d'un dag  $G = (V, E)$  és una seqüència

$$v_1, v_2, v_3, \dots, v_n$$

tal que  $V = \{v_1, \dots, v_n\}$  i si  $(v_i, v_j) \in E$ , llavors  $i < j$ .

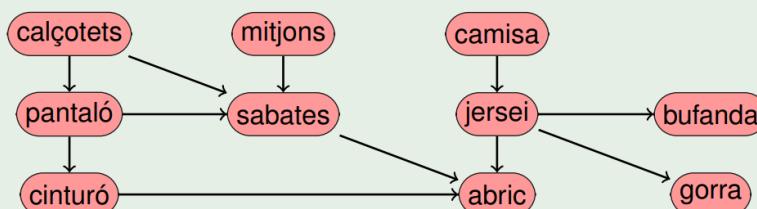
## Exemple

Un dag pot tenir més d'una ordenació topològica.



Tant 3,4,1,2,5 com 3,4,1,5,2 en són ordenacions topològiques.

## Exemple: roba masculina d'hivern



Possibles ordenacions:

- calcotets, mitjons, pantaló, camisa, cinturó, jersei, sabates, abric, bufanda, gorra
- mitjons, camisa, calcotets, jersei, pantaló, cinturó, bufanda, gorra, sabates, abric

## Ordenació topològica

Preparació del vector  $i$  de la pila d'un graf de  $n$  vèrtexs i  $m$  arestes.

Representació amb llistes d'adjacència. Cost  $\Theta(n + m)$ .

```
list<int> ordenacio_topologica (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);
}
```

Bucle principal.

```
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;
}
```

Es visita

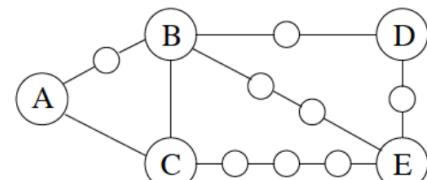
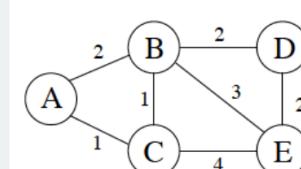
- un cop cada vèrtex
- un cop cada aresta

Per tant, el cost és  $\Theta(n + m)$ .

## Algorisme de Dijkstra

Truc per utilitzar BFS en grafs amb distàncies

Trencar les arestes del graf d'entrada en arestes de "mida unitària" introduint vèrtexs de farciment.

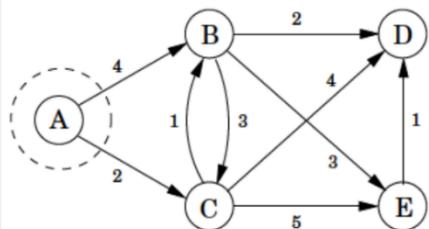


Cada aresta  $e = (u, v)$  amb etiqueta  $d(e)$  se substitueix per  $d(e)$  arestes amb etiqueta 1 afegint-hi  $d(e) - 1$  vèrtexs nous entre  $u$  i  $v$ .

## Esquema voraç

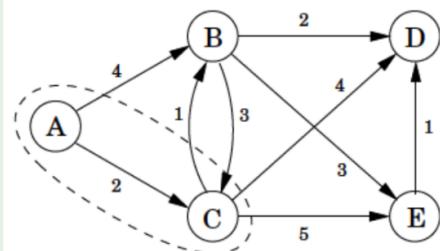
Un **algorisme voraç** resol un problema per etapes fent, a cada etapa, allò que sembla millor. Habitualment, consisteixen en una estratègia simple que es va repetint.

## Algorisme de Dijkstra



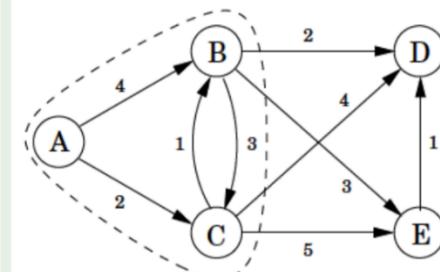
A: 0	D: $\infty$
B: 4	E: $\infty$
C: 2	

Exemple (*Algorithms*, Dasgupta et al.)



A: 0	D: 6
B: 3	E: 7
C: 2	

Exemple (*Algorithms*, Dasgupta et al.)



A: 0	D: 5
B: 3	E: 6
C: 2	

Quina cua amb prioritat és millor?

Implementació	esborrar-min	afegir/ decrementar-clau	$n \times$ esborrar-min + $(n + m) \times$ afegir
vector	$O(n)$	$O(1)$	$O(n^2)$
heap binari	$O(\log n)$	$O(\log n)$	$O((n + m) \log n)$
heap d-ari*	$O\left(\frac{d \log n}{\log d}\right)$	$O\left(\frac{\log n}{\log d}\right)$	$O((nd + m)\frac{\log n}{\log d})$
heap Fibonacci	$O(\log n)$	$O(1)^+$	$O(n \log n + m)$

\* Tria òptima per a  $d$ :  $d = m/n$ .

+ Cost amortitzat ( $O(1)$  en mitjana al llarg de tot l'algorisme).

```

typedef pair<double, int> ArcP;           // arc amb pes
typedef vector< vector<ArcP> > GrafP;   // graf amb pesos

void dijkstra(const GrafP& 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<bool> S(n, false);
    priority_queue<ArcP, vector<ArcP>, greater<ArcP> > Q;
    Q.push(ArcP(0, s));
    
```

$\Theta((n + m) \log n)$ .

## Definition

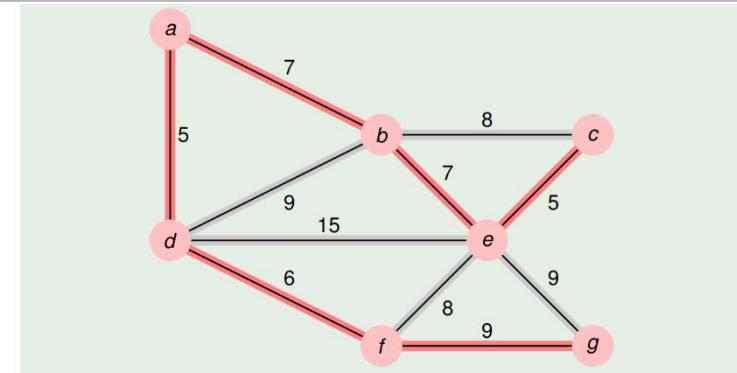
Un **arbre d'expansió** (*spanning tree*, en anglès) de  $G$  és un subgraf  $T = (V, A)$  de  $G$ , amb  $A \subseteq E$ , que és connex i acíclic.

Observem que  $T$  és un arbre i conté tots els vèrtexos de  $G$

Un **arbre d'expansió mínim** (*MST*, de l'anglès *minimum spanning tree*) de  $G$  és un arbre d'expansió  $T = (V, A)$  de  $G$  el pes total del qual

$$\omega(T) = \sum_{e \in A} \omega(e)$$

és mínim entre tots els arbres d'expansió de  $G$ .



## Definition

Un subconjunt d'arestes  $A \subseteq E$  és **prometedor** si  $A$  és un subconjunt de les arestes d'un MST de  $G$ .

## Definition

Un **tall** d'un graf  $G = (V, E)$  és una partició de  $V$ , és a dir, un parell  $(C, C')$  tal que

- $C \cup C' = V$
- $C \cap C' = \emptyset$

## Definition

Una aresta  $e$  **respecta** un tall  $(C, C')$  si ambdós extrems de  $e$  pertanyen a  $C$  o ambdós pertanyen a  $C'$ ; altrament, diem que  $e$  **travessa** el tall.

Un conjunt d'arestes  $A$  **respecta** un tall si totes les arestes de  $A$  el respecten.

## Arbres d'expansió mínims

### Teorema

Sigui  $A$  un conjunt prometedor d'arestes que respecta el tall  $(C, C')$  de  $G$ .

Sigui  $e$  una aresta amb pes mínim entre les que travessen el tall  $(C, C')$ .

Aleshores,  $A \cup \{e\}$  és prometedor.

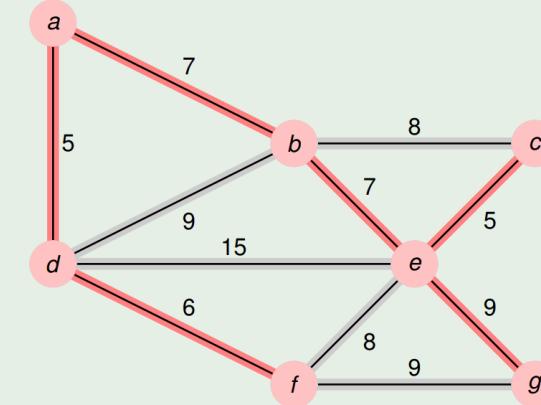
## Algorisme de Prim

En l'**algorisme de Prim** (coneugut també com **algorisme Prim-Jarník**), mantenim un subconjunt de vèrtexs visitats.

- El conjunt de vèrtexs es divideix entre visitats i no visitats
- Cada iteració de l'algorisme tria l'aresta de pes mínim entre les que uneixen vèrtexs visitats amb no visitats

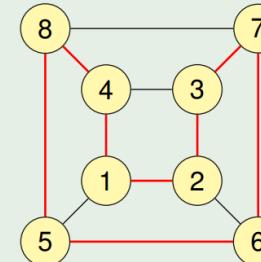
```
typedef pair<int,int> P;
int mst(const vector<vector<P>>& g) { // Ret. el cost d'un MST
    vector<bool> vis(n, false);
    vis[0] = true;
    priority_queue<P, vector<P>, greater<P> > pq;
    for (P x : g[0]) pq.push(x);
    int sz = 1;
    int sum = 0;
    while (sz < n) {
        int c = pq.top().first;
        int x = pq.top().second;
        pq.pop();
        if (not vis[x]) {
            vis[x] = true;
            for (P y : g[x]) pq.push(y);
            sum += c;
            ++sz; } }
    return sum; }
```

$O(m \log n)$ .



## Algorismes de força bruta

Exemple: Cicle hamiltonià



Les possibilitats són totes les permutacions de vèrtexs:

(12345678), (12345687), ..., (12376584), ...

BINARI( $n$ )

(processar totes les cadenes binàries de mida  $n$ )

si  $n = 0$  llavors

PROCESSAR( $C$ )

si no

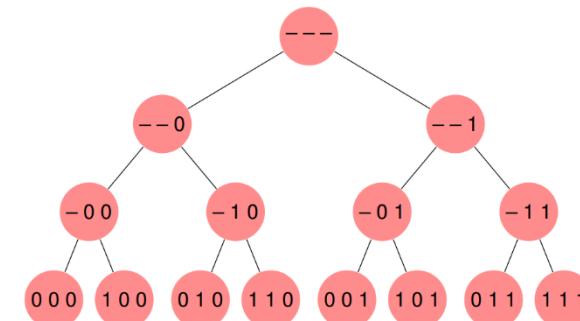
$C[n] \leftarrow 0$ ; BINARI( $n - 1$ )

$C[n] \leftarrow 1$ ; BINARI( $n - 1$ )

$$T(n) = 2T(n - 1) + \Theta(1).$$

$$T(n) \in \Theta(2^n).$$

Per a  $n = 3$ , s'obté l'arbre de recursió:



Podem considerar un algorisme genèric iteratiu de cerca exhaustiva que fa servir els procediments:

- PRIMER( $x$ ): genera un primer candidat
- SEGÜENT( $x, c$ ): genera el candidat següent a  $c$
- VÀLID( $x, c$ ): comprova si el candidat  $c$  és solució
- NUL( $c$ ): diu si  $c$  és un candidat “nul”

### Algorisme genèric de força bruta

```
FORÇA BRUTA( $x$ )
 $c \leftarrow \text{PRIMER}(x)$ 
mentre no ( $\text{NUL}(c)$ )
    si  $\text{VALID}(x, c)$  llavors
        PROCESSAR( $c$ )
     $c \leftarrow \text{SEGÜENT}(x, c)$ 
```

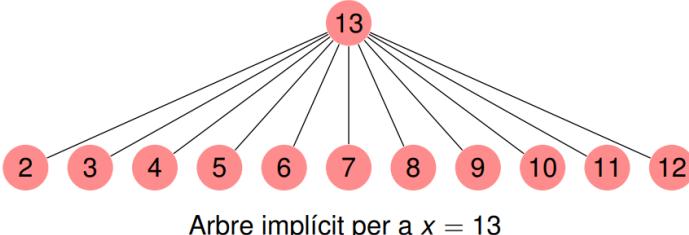
### Exemple: nombres primers

```
bool es_primer (int x) {
    if (x <= 1) return false;
    for (int i = 2; i < x; ++i)
        if (x % i == 0) return false;
    return true; }
```

Nombre màxim d'iteracions:  $(x - 1) - 2 + 1 = x - 2$ .

Cost en funció de  $x$ :  $\Theta(x)$ .

Cost en funció de  $n = |x|$ :  $\Theta(2^n)$ .



### Cerca amb retrocés

Un algorisme de **cerca amb retrocés** (o **tornada enrere**) funciona com una cerca exhaustiva, però s'atura quan troba una solució parcial que no es pot estendre a una solució.

L'esquema de cerca amb retrocés:

- es pot veure com una implementació intel·ligent de la cerca exhaustiva amb un cost millorat, però sovint encara exponencial
- en anglès, s'anomena *backtracking*

### Exemple: moblar un pis

- **Estratègia de força bruta:** provar totes les configuracions dels mobles en tots els espais
- **L'estratègia de cerca amb retrocés** aprofita que:
  - cada moble acostuma a anar a un espai concret (*no posarem el sofà a la cuina*)
  - hi ha mobles que van junts (*cadires i taula, llit i tauletes*)
  - si una subdistribució no és satisfactòria, no considerarem la distribució que la conté (*si no ens agrada posar un moble davant d'una finestra, ja no explorarem a partir d'aquí*)

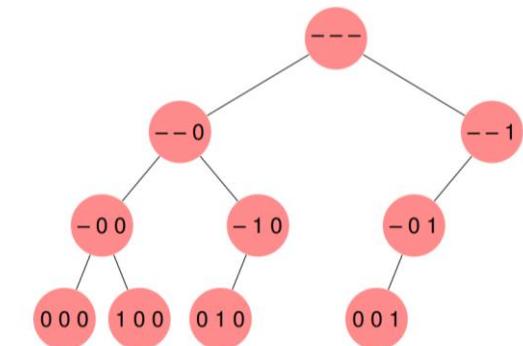
Eliminar un gran grup de possibilitats en un pas es coneix com a **poda**.

### BINARI( $n, i$ )

(processar totes les cadenes binàries de mida  $n$  amb  $\leq k - i$  uns)

```
si  $n = 0$  llavors
    PROCESSAR( $C$ )
si no
     $C[n] \leftarrow 0$ ; BINARI( $n - 1, i$ )
    si  $i < k$  llavors
         $C[n] \leftarrow 1$ ; BINARI( $n - 1, i + 1$ )
```

Per a  $n = 3$  i  $k = 1$ , s'obté l'arbre de recursió:



### Algorisme genèric

Es pot definir un algorisme genèric de tornada enrere:

- L'espai de solucions d'un problema s'acostuma a organitzar en forma d'**arbre de configuracions**
- Cada node o configuració de l'arbre es representa amb un vector

$$A = (a_1, a_2, \dots, a_k)$$

que conté les tries ja fetes

- El vector  $A$  s'amplia en la fase *avançar* triant un  $a_{k+1}$  d'un **conjunt de candidats**  $S_{k+1}$  (*explorar en profunditat*)
- $A$  es redueix en la fase *retrocedir* (*backtrack*)

TORNADA ENRERE( $x$ )

calcular  $S_1$  // conjunt de candidats per a  $a_1$

$k \leftarrow 1$

mentre  $k > 0$

mentre  $S_k \neq \emptyset$

$a_k \leftarrow$  un element de  $S_k$

$S_k \leftarrow S_k - \{a_k\}$

$A \leftarrow (a_1, a_2, \dots, a_k)$

si SOLUCIÓ( $A$ ) llavors

PROCESSAR( $A$ )

$k \leftarrow k + 1$  // avançar

calcular  $S_k$  // candidats per a  $a_k$

$k \leftarrow k - 1$  // retrocedir

Exemple: permutacions de  $n$  elements

Quines són les permutacions dels naturals  $\{1, \dots, n\}$ ?

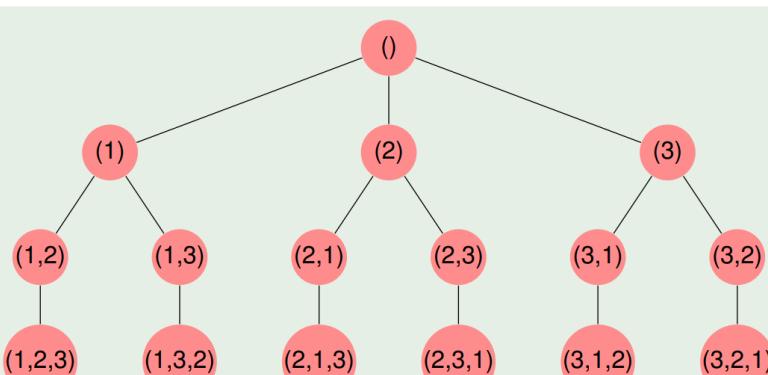
- Hi ha  $n$  possibilitats per al primer
- Fixat el primer, hi ha  $n - 1$  possibilitats per al segon
- Repetint el raonament, obtenim

$$\prod_{k=1}^n k = n!$$

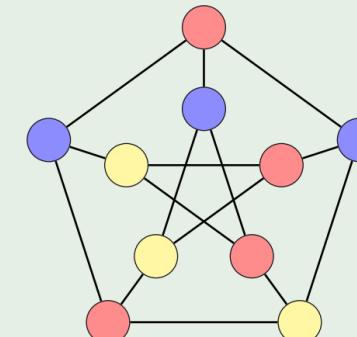
Adaptem l'algorisme genèric amb

- $A = (a_1, \dots, a_k)$ , tal que  $a_i$  és l' $i$ -èsim element triat
- $S_1 = \{1, \dots, n\}$  i  $S_{k+1} = \{1, \dots, n\} - A$  per a  $k \geq 1$

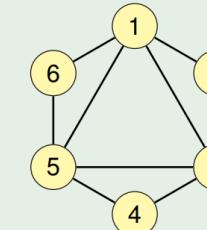
Amb  $n = 3$ , s'obté l'arbre de configuracions:



El problema de la **3-colorabilitat** consisteix a decidir si es pot assignar un color a cada vèrtex d'un total de 3 de manera que els vèrtexs adjacents tinguin colors diferents.



Una 3-coloració del graf de Petersen



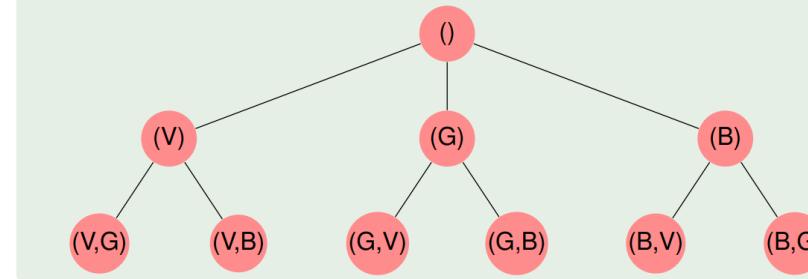
- Les **configuracions** seran assignacions parcials de colors, és a dir,

$$A = (a_1, a_2, \dots, a_k)$$

representarà el fet que el vèrtex  $i$  s'acoloreix amb el color  $a_i \in \{B, G, V\}$

- El conjunt de **candidats**  $S_{k+1}$  per a  $a_{k+1}$  contindrà els colors compatibles amb els veïns que ja han estat acolorits

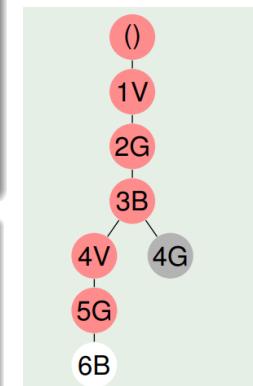
Els 3 primers nivells de l'arbre de configuracions serien:



Fent la tria

- $S_1 = \{V\}$
- $S_2 = \{G\}$

i definint  $S_{k+1} = \{c \in \{V, G, B\} \mid \forall i \leq k \quad (\{i, k+1\} \in A(G) \Rightarrow c \neq a_i)\}$ , s'obté l'arbre de configuracions



# La reconstrucció Turnpike

## Observació

És fàcil construir el conjunt de distàncies a partir del conjunt de punts en temps  $\Theta(n^2)$  (ordenades, en temps  $\Theta(n^2 \log n)$ ).

## Problema de la reconstrucció Turnpike

Donat el conjunt de distàncies, reconstruir el conjunt de punts.

Exemple amb  $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$

Com que  $|D| = 15 = n(n - 1)/2$ , obtenim  $n = 6$ .

- Comencem fent  $x_1 = 0$ .
- Clarament,  $x_6 = 10$ .
- Eliminem 10 de  $D$ . Ara,

$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}.$$

$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}$ , eix x: 0 \_ \_ \_ 10

- La distància més gran que queda és 8. Per tant,

$$x_2 = 2 \text{ o } x_5 = 8.$$

Si tenen solució, seran simètriques. Triem  $x_5 = 8$ .

- Eliminem de  $D$  les distàncies  $x_6 - x_5 = 2$  i  $x_5 - x_1 = 8$ . Ara,

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}.$$

$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}$ , eix x: 0 \_ \_ \_ 8 10

- Com que 7 és el valor més alt a  $D$ ,  $x_4 = 7$  o  $x_2 = 3$ .

- Si  $x_4 = 7$ , les distàncies

$$x_6 - 7 = 3 \text{ i } x_5 - 7 = 1$$

han de ser a  $D$ , i hi són.

- Si  $x_2 = 3$ , les distàncies

$$3 - x_1 = 3 \text{ i } x_5 - 3 = 5$$

han de ser a  $D$ , i també hi són.

No tenim cap guia per triar. Provarem una opció ( $x_4 = 7$ ) i veurem si porta a una solució. Si no, tornarem enrere.

- Eliminem les distàncies  $x_4 - x_1 = 7$ ,  $x_5 - x_4 = 1$  i  $x_6 - x_4 = 3$ . Ara,

$$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}.$$

$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}$ , eix x: 0 \_ \_ 7 8 10

- La distància més gran és 6. Per tant,  $x_3 = 6$  o  $x_2 = 4$ .

- Si  $x_3 = 6$ , llavors  $x_4 - x_3 = 1$ , que no pertany a  $D$ .
- Si  $x_2 = 4$ , llavors

$$x_2 - x_1 = 4 \text{ i } x_5 - x_2 = 4$$

i això és impossible perquè 4 només apareix un cop a  $D$ .

Aquesta línia de raonament no porta a una solució. Tornem enrere i triem  $x_2 = 3$ .

- En el conjunt de distàncies anteriors

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\},$$

eliminem  $x_2 - x_1 = 3$ ,  $x_5 - x_2 = 5$  i  $x_6 - x_2 = 7$ . Ara,

$$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}.$$

$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}$ , eix x: 0 3 \_ \_ 8 10

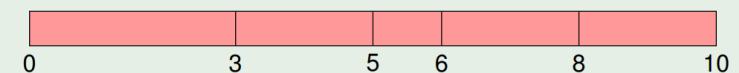
- Com que 6 és el valor més alt a  $D$ ,  $x_4 = 6$  o  $x_3 = 4$ .

- Si  $x_3 = 4$ , tant  $x_3 - x_1$  com  $x_5 - x_3$  valdrien 4, però no és possible perquè  $D$  només conté un 4.

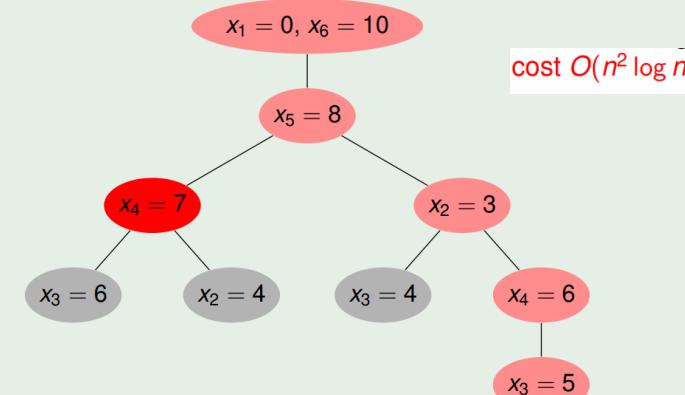
Per tant,  $x_4 = 6$  i obtenim

$$D = \{1, 2, 3, 5, 5\}.$$

- Només queda triar  $x_3 = 5$ . Com que ens queda  $D = \emptyset$ , tenim una solució:



Arbre de decisió

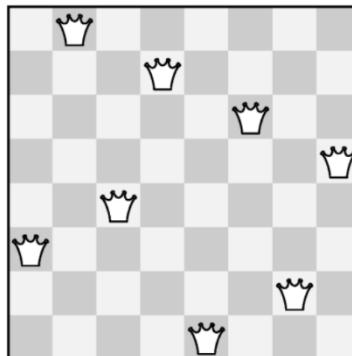


Els nodes grisos indiquen que els punts triats són inconsistents amb les dades. El node vermell no té nodes vàlids com a fills.

# Les $n$ reines

Problema de les vuit reines

Col·locar vuit reines en un tauler d'escacs sense que cap amenaci cap altra.



Nombre de solucions no isomorfes (per rotació o reflexió) de les  $n$  reines per a  $n \in \{1, \dots, 10\}$ :

$n$	solucions
1	1
2	0
3	0
4	1
5	2
6	1
7	6
8	12
9	46
10	92

Primera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui “legal” (que es pugui estendre a una solució completa)
- cost en cas pitjor:  $\Theta(n^n)$

Implementarem la posició de les reines amb un vector

vector<int> T;

que indicarà que la reina de la fila  $i$  és a la columna  $T[i]$ .

Per saber si les reines de les files  $i$  i  $k$  comparteixen

- columna, comprovem si  $T[i] = T[k]$
- diagonal principal ( $\searrow$ ), comprovem si  $T[i] - i = T[k] - k$
- diagonal secundària ( $\nearrow$ ), comprovem si  $T[i] + i = T[k] + k$

```
class NReines {
```

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

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

```
    bool legal(int i) {
        // Indica si la config. amb les reines 0..i es legal
        // sabent que la config. amb les reines 0..i-1 ho es
```

```
        for (int k = 0; k < i; ++k) {
            if (T[k] == T[i] || T[i]-i == T[k]-k || T[i]+i == T[k]+k) {
                return false;
            }
        }
        return true;
    }
```

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

```
public:
```

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

## Programa principal

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

## Segona implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- amb marcatges
- cost en cas pitjor:  $\Theta(n^n)$

```
class NReines {
    int n; // nombre de reines
    vector<int> T; // configuració actual
    vector<boolean> mc; // marca de les columnes
    vector<boolean> md1; // marca de les diagonals 1
    vector<boolean> md2; // marca de les diagonals 2

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

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

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

public:
    NReines(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);
        recursiu(0);
    }
};
```

## Tercera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- amb marcatges
- amb un booleà per finalitzar la cerca
- cost en cas pitjor:  $\Theta(n^n)$

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

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

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

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

public:
    NReines(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);
        trobat = false;
        recursiu(0);
    }
};
```

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

# La motxilla

## Problema de la motxilla (entera)

Donada una motxilla que pot carregar un pes  $C$  i  $n$  objectes amb

- pesos  $p_1, p_2, \dots, p_n$
- i valors  $v_1, v_2, \dots, v_n$

trobar una selecció  $S \subseteq \{1, \dots, n\}$  dels objectes

- amb el màxim valor  $\sum_{i \in S} v_i$
- que no superi la capacitat de la motxilla:

$$\sum_{i \in S} p_i \leq C$$

**Solució amb fita superior.** Aquest cop es té en compte la contribució màxima que podrien arribar a tenir tots els objectes a partir de l' $i+1$  (encara que no càpiguen a la motxilla).

Si fins i tot agafant-los tots no es pogués superar el millor cost trobat fins ara, no cal seguir per aquell camí.

```
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; // solució activa
    vector<boolean> sol; // millor solució provisional
    double millor; // cost millor solució 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 {
            // la 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);
            }
        }
    }
}
```

Són com abans: solució, cost, main.

```
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);
    }

    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;
    }
}
```

- L' **anàlisi d'algorismes** estudia la quantitat de recursos que necessita un algorisme per resoldre un problema.
- La **teoria de la complexitat** considera els algorismes possibles que resolen un mateix problema.
- Mentre l'anàlisi d'algorismes se centra en els **algorismes**, la teoria de la complexitat s'interessa pels **problemes**.

Per classificar els problemes, considerarem les seves versions decisionals.

## Definició

Un **problema decisional** és un problema en el qual la sortida és **sí** o **no**

Equivalentment, un problema és decisional quan s'ha de determinar si l'**entrada** (també anomenada **instància**) satisfà o no una certa propietat.

Molts problemes vistos fins ara són decisionals:

- **connectivitat**: donat un graf, decidir si és connex.
- **3-colorabilitat**: donat un graf, decidir si és 3-colorable.
- **accessibilitat**: donat un graf  $G = (V, E)$  i dos vèrtexs  $i, j \in V$ , decidir si hi ha un camí a  $G$  entre  $i$  i  $j$ .

o es poden transformar en decisionals:

- **camí curt**: donat un graf  $G = (V, E)$ , dos vèrtexs  $i, j \in V$  i un natural  $k$ , decidir si hi ha un camí a  $G$  entre  $i$  i  $j$  de longitud màxima  $k$ .

Certes versions decisionals d'alguns problemes no tenen gaire sentit.

#### Problema de les $n$ -reines decisional (1a versió)

Donat un natural  $n$ , decidir si es poden col·locar  $n$  reines en un tauler  $n \times n$  sense que cap n'amenaci cap altra.

Se sap que hi ha solucions per a tot  $n \neq 2, 3$ .

Per tant, l'algorisme següent decideix el problema en temps  $\Theta(1)$ .

REINES( $n$ )

  si  $n = 2$  o  $n = 3$  llavors

    retornar FALS

  si no

    retornar CERT

El que ens interessa és trobar una solució, no saber si existeix.

#### Problema de les $n$ -reines decisional (2a versió)

Donat un natural  $n$  i  $k$  valors  $r_1, \dots, r_k$ , amb  $k \leq n$ , decidir si es poden col·locar  $n$  reines en un tauler  $n \times n$  sense que cap n'amenaci cap altra i de manera que per a tot  $i$  tal que  $1 \leq i \leq k$ , la reina de la fila  $i$  ocupa la columna  $r_i$ .

Aquesta versió, tot i ser decisional, permet trobar una solució amb

$$n + (n - 1) + (n - 2) \cdots + 1 = \sum_{i=1}^n i = \frac{n(n + 1)}{2}$$

execucions de l'algorisme que el resol.

Donat un problema  $A$  definit sobre un conjunt d'entrades  $E$ , distingirem entre:

- les **entrades positives**: les que pertanyen a  $A$
- les **entrades negatives**: les que pertanyen a  $E - A$

#### Primalitat

El problema de la primalitat el podem descriure informalment així:

##### Primalitat

Donat un natural  $x$ , determinar si  $x$  és primer.

O bé formalment com el conjunt de les entrades positives:

$$P = \{x \in \mathbb{N} \mid x \text{ és primer}\}.$$

Un exemple de funció de mida per als naturals és la que compta el nombre de dígits de la representació binària:

$$|x| = \text{nombre de dígits de } x \text{ en binari} = \lfloor \log_2 x \rfloor + 1.$$

#### Problemes decisionals

- Cal distingir entre tres nivells d'abstracció:
  - Les **entrades**  
Per exemple, les seqüències d'enters
  - Els **problems**: conjunts d'entrades  
Per exemple, les seqüències d'enters ordenades
  - Les **classes**: conjunts de problemes  
Per exemple, els que podem resoldre en temps lineal

#### Temps polinòmic i exponencial

Suposem que  $t : \mathbb{N} \rightarrow \mathbb{N}$  és una funció.

##### Algorismes de cost $t$

Diem que un algorisme  $\mathcal{A}$  té cost  $t$  si el seu cost en cas pitjor pertany a  $\mathcal{O}(t)$ .

#### Problemes decidibles en temps $t$

Si un algorisme  $\mathcal{A}$  rep entrades d'un conjunt  $E$  i té una sortida binària, escriurem:

$$\mathcal{A} : E \rightarrow \{0, 1\}.$$

Diem que un problema decisional  $A$  és decidable en temps  $t$  si existeix un algorisme de cost  $t$  que el decideix (el resol); és a dir, si existeix  $\mathcal{A} : E \rightarrow \{0, 1\}$  de cost  $t$  tal que, per a tot  $x \in E$ :

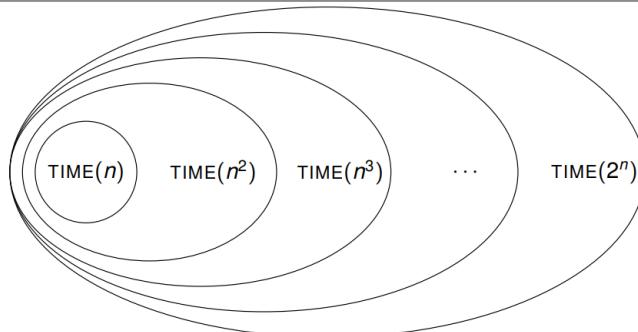
$$x \in A \Rightarrow \mathcal{A}(x) = 1$$

$$x \notin A \Rightarrow \mathcal{A}(x) = 0$$

## Classe TIME( $t$ )

Donada una funció  $t : \mathbb{N} \rightarrow \mathbb{N}$ , agrupem els problemes decidibles en temps  $t$ :

$$\text{TIME}(t) = \{A \mid A \text{ és decidible en temps } t\}.$$



## Classe P

Definim la classe P com la unió de les classes de temps polinòmiques:

$$P = \bigcup_{k>0} \text{TIME}(n^k).$$

És a dir, un problema pertany a P si és decidible en temps  $n^k$  per a algun  $k$ .

P són els problemes que podem resoldre amb un algorisme polinòmic

## Classe EXP

Definim la classe EXP com la unió de les classes de temps exponencials:

$$\text{EXP} = \bigcup_{k>0} \text{TIME}(2^{n^k}).$$

És a dir, un problema és a EXP si és decidible en temps  $2^{n^k}$  per a algun  $k$ .

EXP són els problemes que podem resoldre amb un algorisme exponencial

Es considera que els problemes de la classe P són **tractables**, mentre que els de la classe EXP que no estan a P són **intractables**

## Exemples

- CONNECTIVITAT  $\in P$
- ACCESSIBILITAT  $\in P$
- PRIMALITAT  $\in P$
- CAMÍ CURT  $\in P$
- 2-COLORABILITAT  $\in P$
- 3-COLORABILITAT  $\in \text{EXP}$  (no se sap si és a P)
- VIATJANT  $\in \text{EXP}$  (no se sap si és a P)
- ATURADA FITADA  $\in \text{EXP}$  (i se sap que no és a P)

## Theorema

$$P \subsetneq \text{EXP}.$$

La inclusió estricta del teorema es pot dividir en dues parts:

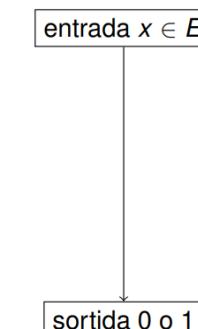
- ①  $P \subseteq \text{EXP}$ . Evident a partir de les definicions:

$$P = \bigcup_{k>0} \text{TIME}(n^k) \subseteq \bigcup_{k>0} \text{TIME}(2^{n^k}) = \text{EXP}$$

- ②  $P \neq \text{EXP}$ . La demostració està fora de l'abast de l'assignatura.

## Indeterminisme

- Els algorismes vistos fins ara són **deterministes**: segueixen un únic **camí de càlcul** des de l'entrada fins al resultat.
- L'execució d'un algorisme  $\mathcal{A} : E \rightarrow \{0, 1\}$  per a un conjunt de dades  $E$  es pot veure com un camí:



Un algorisme **indeterminista** pot arribar a un resultat a través de diferents camins. El seu funcionament s'assembla més a un **arbre**.

Un algorisme  $\mathcal{A} : E \rightarrow \{0, 1\}$  és **indeterminista** si pot fer ús d'una nova funció

$$\text{TRIAR}(x)$$

que retorna un nombre  $y$  entre 0 i  $x$ .

Aleshores:

- $\mathcal{A}$  comença el càlcul de manera determinista fins la primera instrucció TRIAR.
- Per a cada valor retornat per TRIAR, el càlcul es divideix en diferents branques amb el valor corresponent.
- Diem que  $\mathcal{A}$  retorna 1 si ho fa en **alguna** de les branques de l'arbre de càlcul.

## El problema

$$\text{COMPOSTOS} = \{x \mid \exists y \quad 1 < y < x \text{ i } y \text{ divideix } x\}$$

té un algorisme determinista trivial de temps exponencial

```
entrada x
per a y = 2 fins x - 1
    si y divideix x llavors
        retornar 1
    retornar 0
```

i un algorisme indeterminista de temps polinòmic

```
entrada x
y ← TRIAR(x)
si 1 < y < x i y divideix x llavors
    retornar 1
retornar 0
```

- En l'exemple anterior, diem que el 3 és un **testimoni** del fet que el nombre 27 és compost.

- És a dir, en el problema COMPOSTOS existeixen:

- Possibles testimonis ( $y < x$ ) del fet que un nombre  $x$  és compost.  
La mida dels testimonis no és més gran que la de l'entrada:  $|y| \leq |x|$
- Un algorisme polinòmic que, donat un  $y$ , **verifica** si  $y$  divideix  $x$ .

$$\text{3-COLORABILITAT} = \{G \mid G \text{ és 3-colorable}\}$$

també té un algorisme exhaustiu de temps exponencial

```
entrada G = (V, E)
n ← |V|
per a cada tupla  $(c_1, \dots, c_n)$  on  $\forall i \leq n \quad c_i \in \{0, 1, 2\}$ 
    si  $(c_1, \dots, c_n)$  és una 3-coloració de G llavors
        retornar 1
    retornar 0
```

i un algorisme indeterminista de temps polinòmic

```
entrada G = (V, E)
n ← |V|
per a i = 1 fins n
     $c_i \leftarrow \text{TRIAR}(2)$ 
    si  $(c_1, \dots, c_n)$  és una 3-coloració de G llavors
        retornar 1
    si no
        retornar 0
```

La **definició formal** dels algorismes polinòmics indeterministes separa:

- el càlcul del testimoni i
- el càlcul determinista.

### Decidibilitat en temps polinòmic indeterminista

Un problema decisional  $A$  definit sobre un conjunt d'entrades  $E$  es diu que és **decidable en temps polinòmic indeterminista** si existeix

- un conjunt  $E'$  de possibles testimonis
- un algorisme polinòmic  $\mathcal{V} : E \times E' \rightarrow \{0, 1\}$  (anomenat **verificador**) i
- un polinomi  $p(n)$

tals que per a tot  $x \in E$ , tenim

$$x \in A \Rightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \in E' \text{ tal que } |y| \leq p(|x|)$$

$$x \notin A \Rightarrow \mathcal{V}(x, y) = 0 \text{ per a tot } y \in E' \text{ tal que } |y| \leq p(|x|)$$

Si  $x \in A$ , els  $y$  tals que  $\mathcal{V}(x, y) = 1$  se'n diuen **testimonis** o **certificats**.

Per veure que un problema  $A$  és decidible en temps polinòmic indeterminista caldrà comprovar:

- que les entrades positives de  $A$  tenen testimonis de mida polinòmica i que les entrades negatives de  $A$  no tenen testimonis de mida polinòmica  
(cal indicar quins són els testimonis)
- que els testimonis es poden verificar en temps polinòmic  
(cal trobar un verificador)

$$\text{COMPOSTOS} = \{x \mid \exists y \quad 1 < y < x \text{ i } y \text{ divideix } x\}$$

- Els **testimonis** per a  $x$  són tots els  $y \neq 1, x$  que divideixen  $x$ .
- El **polinomi** és  $p(n) = n$
- El **verificador** és

```
 $\mathcal{V}(x, y)$ 
    si  $1 < y < x$  i  $y$  divideix  $x$  llavors
        retornar 1
    si no
        retornar 0
```

COMPOSTOS és decidible en temps polinòmic indeterminista perquè

$$x \in \text{COMPOSTOS} \Leftrightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \text{ t.q. } |y| \leq p(|x|).$$

## 3-colorabilitat

Considerem el problema

$$3\text{-COLOR} = \{ G \mid G \text{ és 3-colorable} \}$$

- 1 Els **testimonis** per a  $G = (V, E)$  són totes les 3-coloracions  $C$  de  $G$  de la forma  $C = (c_1, c_2, \dots, c_n)$ , on  $n = |V|$  i  $c_i \in \{0, 1, 2\}$  per a tot  $i \leq n$ .
- 2 El **polinomi** (amb representacions raonables de  $G$  i  $C$ ) pot ser  $p(n) = n$
- 3 El **verificador** és

```

 $\mathcal{V}(G, C)$ 
 $n \leftarrow |V|$ 
si  $C$  és una 3-coloració de  $G$  llavors
    retornar 1
si no
    retornar 0

```

Tots els problemes decidibles en temps polinòmic indeterminista els agrupem en una classe.

## Classe NP

Definim la classe NP (de *nondeterministic polynomial time*) com:

$$\text{NP} = \{ A \mid A \text{ és decidible en temps polinòmic indeterminista} \}.$$

Diferència fonamental entre P i NP:

- els testimonis dels problemes de P es poden **trobar** en temps polinòmic
- els testimonis dels problemes de NP es poden **verificar** en temps polinòmic

## Quadrats perfectes i compostos

- 1 QUADRATS =  $\{ x \in \mathbb{N} \mid \exists y \quad 1 \leq y < x \quad \text{i} \quad x = y^2 \}$
- 2 COMPOSTOS =  $\{ x \in \mathbb{N} \mid \exists y \quad 1 < y < x \quad \text{i} \quad y \text{ divideix } x \}$

## 2 i 3-colorabilitat

- 1 2-COLORABILITAT =  $\{ G \mid G \text{ és 2-colorable} \}$
- 2 3-COLORABILITAT =  $\{ G \mid G \text{ és 3-colorable} \}$

## Teorema

$$P \subseteq NP.$$

## Demostració

Tot algorisme determinista també és indeterminista (però no fa ús de la instrucció TRIAR).

Vist d'una altra manera, per a tot  $A \in P$ , podem crear verificadors  $\mathcal{V}$  tals que

$$\mathcal{V}(x, y) = 1 \Leftrightarrow x \in A$$

independentment de  $y$ . Per tant,  $A \in NP$ .

Diferència entre NP i EXP:

- els problemes de NP tenen testimonis verificables en temps polinòmic
- els problemes d'EXP poden tenir testimonis exponencialment llargs

## Teorema

$$NP \subseteq EXP.$$

## Demostració

Sigui  $A \in NP$ . Llavors, existeix un polinomi  $p(n)$  i un verificador  $\mathcal{V}$  tals que

$$x \in A \Rightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \in E \text{ tal que } |y| \leq p(|x|)$$

$$x \notin A \Rightarrow \mathcal{V}(x, y) = 0 \text{ per a tot } y \in E \text{ tal que } |y| \leq p(|x|)$$

Podem considerar un algorisme exponencial per a  $A$  que cerca un testimoni:

```

    entrada  $x$ 
    per a tot  $y$  tal que  $|y| \leq p(|x|)$ 
        si  $\mathcal{V}(x, y) = 1$  llavors
            retornar 1
        retornar 0

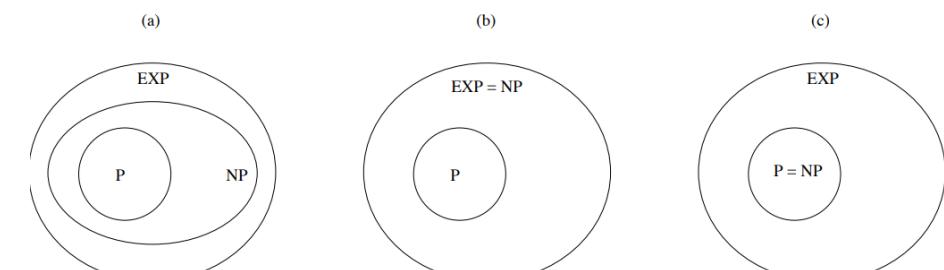
```

És fàcil veure que l'algorisme anterior és exponencial i decideix  $A$ .

Per tant,  $A \in EXP$ .

- No se sap si  $P = NP$ .

- Podem assegurar que  $P \neq NP$  o  $NP \neq EXP$   
(perquè se sap que  $P \neq EXP$ ). Prendrem (a) com a hipòtesi de treball.



# Concepte de reducció

## Reduccions

Siguin  $A$  i  $B$  dos problemes decisionals. Diem que  $A$  es redueix a  $B$  en temps polinòmic si existeix un algorisme polinòmic  $\mathcal{F}$  tal que

$$x \in A \Rightarrow \mathcal{F}(x) \in B$$

$$x \notin A \Rightarrow \mathcal{F}(x) \notin B$$

En aquest cas, escrivim  $A \leq^p B$  (o  $A \leq^p B$  via  $\mathcal{F}$ ) i diem que  $\mathcal{F}$  és una reducció polinòmica de  $A$  a  $B$ .

## Paritat

Considerem el llenguatge dels nombres parells

$$\text{PARELLS} = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \quad x = 2y\}$$

i el dels senars

$$\text{SENARS} = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \quad x = 2y + 1\}$$

Veiem que podem reduir PARELLS a SENARS ( $\text{PARELLS} \leq^p \text{SENARS}$ ) amb un algorisme  $\mathcal{F}$  tal que  $\mathcal{F}(x) = x + 1$ . És evident que per a tot  $x$ :

$$x \in \text{PARELLS} \Leftrightarrow \mathcal{F}(x) \in \text{SENARS}.$$

## Particions

Considereu els dos problemes següents.

### Partició

Donats els naturals  $x_1, x_2, \dots, x_n$ , determinar si es poden dividir en dos grups que sumin el mateix.

### Motxilla

Donats els naturals  $x_1, x_2, \dots, x_n$  i una capacitat  $C \in \mathbb{N}$ , determinar si es pot trobar una selecció dels  $x_i$ 's que sumi exactament  $C$ .

$$\text{PARTICIÓ} = \{(x_1, \dots, x_n) \mid \exists I \subseteq \{1, \dots, n\} \quad \sum_{i \in I} x_i = \sum_{i \notin I} x_i\}$$

$$\text{MOTXILLA} = \{(x_1, \dots, x_n, C) \mid \exists I \subseteq \{1, \dots, n\} \quad \sum_{i \in I} x_i = C\}$$

L'algorisme

```
 $\mathcal{F}(x_1, \dots, x_n)$ 
 $S \leftarrow \sum_{i=1}^n x_i$ 
si  $S$  és senar llavors
    retornar  $(x_1, \dots, x_n, S+1)$ 
si no
    retornar  $(x_1, \dots, x_n, S/2)$ 
```

és una reducció polinòmica de PARTICIÓ a MOTXILLA:

$$(x_1, \dots, x_n) \in \text{PARTICIÓ} \Leftrightarrow \mathcal{F}(x_1, \dots, x_n) \in \text{MOTXILLA}.$$

## Definició

Un **camí hamiltonià** d'un graf  $G$  és un camí que passa per tots els vèrtexs sense repetir-ne cap.

## Propietats: reflexivitat

Per a tot  $A$ ,  $A \leq^p A$ .

N'hi ha prou a considerar l'algorisme que calcula la funció identitat:

$\mathcal{F}(x)$   
retornar  $x$

## Equivalència polinòmica

Donats dos problemes decisionals  $A, B$ , escrivim  $A \equiv^p B$  si  $A \leq^p B$  i  $B \leq^p A$ .

És evident que, per a tot  $x$

$$x \in A \Leftrightarrow \mathcal{F}(x) = x \in A.$$

## Propietats: transitivitat

Per a tot  $A, B, C$ , si  $A \leq^p B$  i  $B \leq^p C$ , llavors  $A \leq^p C$ .

Si

- $A \leq^p B$  via  $\mathcal{F}$  i
- $B \leq^p C$  via  $\mathcal{G}$ ,

llavors la composició  $\mathcal{G} \circ \mathcal{F}$  ( $\mathcal{F}| \mathcal{G}$  en notació pipe de UNIX) demostra que  $A \leq^p C$ .

Considerem que  $\mathcal{G} \circ \mathcal{F}(x) = \mathcal{G}(\mathcal{F}(x))$ .

## Corol·lari

Les reduccions formen un preordre.

## Qüestió

Observeu que, si bé les reduccions formen un preordre, en canvi no formen un ordre parcial perquè no compleixen la propietat antisimètrica:

- $\forall A, B \quad A \leq^p B \wedge B \leq^p A \Rightarrow A = B$

## Tancament de P per reduccions

Per a tot  $A, B$ , si  $A \leq^p B$  i  $B \in P$ , llavors  $A \in P$ .

Si

- $\mathcal{B}$  és un algorisme polinòmic per a  $B$  i
- $\mathcal{F}$  és un algorisme polinòmic que demostra  $A \leq^p B$ ,

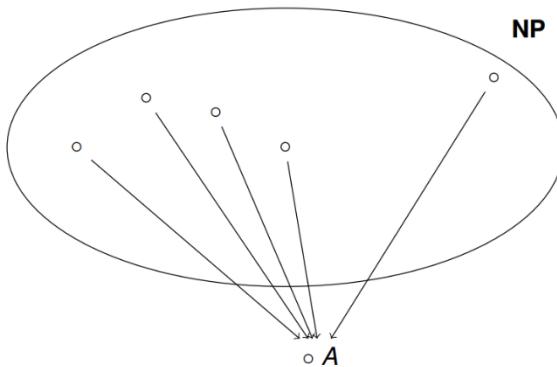
llavors la composició  $\mathcal{B} \circ \mathcal{F}$  és un algorisme polinòmic per a  $A$ :

- 1  $\mathcal{B} \circ \mathcal{F}$  és polinòmic perquè és composició d'algorismes polinòmics
- 2  $\mathcal{B} \circ \mathcal{F}(x)$  accepta  $\Leftrightarrow \mathcal{B}$  accepta  $\mathcal{F}(x) \Leftrightarrow \mathcal{F}(x) \in B \Leftrightarrow x \in A$

## Teoria de la NP-completesa

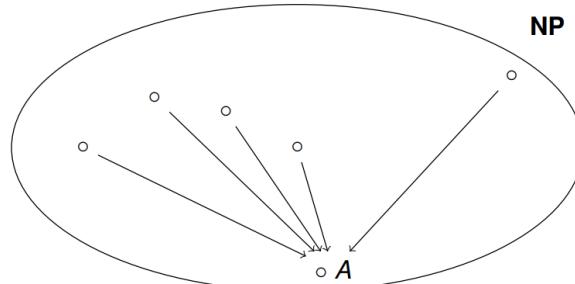
### Definició

Un problema  $A$  és **NP-difícil** si per a tot problema  $B \in \text{NP}$  tenim que  $B \leq^P A$ .



### Definició

Un problema  $A$  és **NP-complet** si és NP-difícil i  $A \in \text{NP}$ .



Qualsevol problema NP-complet “representa” tota la classe NP respecte de P.

Més formalment...

### Proposició

Sigui  $A$  un problema NP-complet. Llavors,  $P = \text{NP}$  si i només si  $A \in P$ .

⇒ Com que  $A$  és NP-complet,  $A \in \text{NP}$  i, per tant,  $A \in P$ .

⇐ Sigui  $A \in P$ .

- ① Com que  $A$  és NP-complet, sabem que per a tot  $B \in \text{NP}$ ,  $B \leq^P A$
- ② Pel tancament de P per reduccions, sabem que si  $B \leq^P A$ , llavors  $B \in P$

Per 1 i 2,  $\text{NP} \subseteq P$  i, per tant,  $P = \text{NP}$ .

Qualsevol parell de problemes NP-complets són equivalents.

Més formalment...

### Proposició

Si  $A$  i  $B$  són NP-complets, llavors  $A \equiv^P B$ .

Com que  $A$  i  $B$  són NP-complets, tenim

- ①  $A \in \text{NP}$  i
- ②  $B$  és NP-difícil

i llavors,  $A \leq^P B$ .

Simètricament, podem deduir que  $B \leq^P A$ . Aleshores,  $A \equiv^P B$ .

### Fòrmules booleanes

- Una **fórmula booleana** (f.b.) és un predicat sobre variables booleanes sense quantificadors.
- Farem servir les connectives  $\vee$  (disjunció),  $\wedge$  (conjunció) i  $\neg$  (negació).

Per exemple,

$$F(x, y, z) = (x \vee y \vee \neg z) \wedge \neg(x \wedge y \wedge z)$$

és una fórmula booleana.

### Forma normal conjuntiva (CNF)

- Un **literal** és una variable afirmada o negada:  $x$ ,  $\neg x$
- Una **clàusula** és una disjunció de literals:  $(x \vee \neg y \vee z)$
- Una fórmula booleana està en **forma normal conjuntiva** (CNF) si és una conjunció de clàusules:  $F(x, y, z) = (x \vee \neg y \vee z) \wedge (\neg x \vee \neg z)$

### Satisfactibilitat

Una fórmula booleana és **satisfactible** si existeix una assignació de valors de veritat a les variables per a la qual la fórmula és certa. Per exemple,

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (\neg x \vee \neg z)$$

és satisfactible amb  $x = 1$ ,  $y = 0$ ,  $z = 0$ . Escrivim  $F(100) = 1$ .

Definim

$$\text{SAT} = \{ F \mid F \text{ és una fórmula booleana satisfactible} \}$$

$$\text{CNF-SAT} = \{ F \mid F \text{ és una f.b. en CNF satisfactible} \}$$

### Teorema de Cook-Levin (1971)

CNF-SAT és NP-complet.

# Teoria de la NP-completesa

## (1) CNF-SAT ∈ NP

- Els **testimonis** són les assignacions de les variables booleanes a  $\{0, 1\}$
- En qualsevol codificació raonable d'una fórmula  $F$  de  $n$  variables,  $n \leq |F|$ . Com que un testimoni  $\alpha$  consta de  $n$  bits,  $|\alpha| = n \leq |F|$
- Per tant, triant  $p(n) = n$ , tenim que  $|\alpha| \leq p(|F|)$
- Podem **verificar** si una assignació  $\alpha$  satisfà  $F$  en temps polinòmic:
  - substituem les variables pels valors donats per  $\alpha$
  - avaluem les connectives de dins cap a fora

### Exemple

En el cas de la fórmula booleana en CNF

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (x \vee \neg z)$$

i l'assignació  $\alpha = 100$  (és a dir,  $x = 1, y = 0, z = 0$ ), el verificador avaluaria:

- $F(\alpha) = (1 \vee \neg 0 \vee 0) \wedge (1 \vee \neg 0)$   
(substituir valors)
- $F(\alpha) = (1 \vee 1 \vee 0) \wedge (1 \vee 1)$   
(calcular negacions)
- $F(\alpha) = (1) \wedge (1)$   
(calcular disjuncions)
- $F(\alpha) = 1$   
(calcular conjuncions)

### Lema

Donat un algorisme  $\mathcal{A} : E \rightarrow \{0, 1\}$  amb cost d'espai polinòmic en cas pitjor, podem trobar en temps polinòmic una f.b. en CNF  $F_{\mathcal{A}}$  tal que per a tot  $y \in E$ :

$$F_{\mathcal{A}}(y) = 1 \Leftrightarrow \mathcal{A}(y) = 1.$$

## (2) CNF-SAT és NP-difícil.

Sigui  $A \in \text{NP}$ . Llavors, hi ha un polinomi  $q$  i un verificador  $\mathcal{V}$  t.q. per a tot  $x$ :

$$x \in A \Leftrightarrow \exists y \quad |y| = q(|x|) \wedge \mathcal{V}(x, y) = 1.$$

Sigui  $\mathcal{V}_x(y)$  un nou verificador, per a  $x$  fixat, tal que

$$\mathcal{V}_x(y) = 1 \Leftrightarrow |y| = q(|x|) \wedge \mathcal{V}(x, y) = 1.$$

Llavors,

$$x \in A \Leftrightarrow \exists y \quad F_{\mathcal{V}_x}(y) \Leftrightarrow F_{\mathcal{V}_x} \in \text{CNF-SAT}.$$

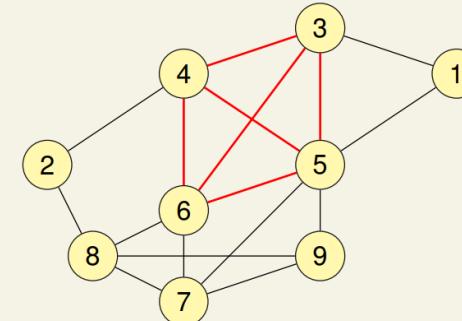
Per tant,  $A \leq^p \text{CNF-SAT}$ .

## Problema de la clica

Diem que  $H$  és un **subgraf complet** de  $G$  si conté totes les arestes possibles entre els seus vèrtexs, és a dir, si  $H$  és isomorf a  $K_i$  per a algun  $i$ . Definim

$$\text{CLICA} = \{ (G, k) \mid G \text{ té un subgraf complet de } k \text{ vèrtexs} \}.$$

Donat el graf  $G$



observem que  $(G, 4) \in \text{CLICA}$  però  $(G, 5) \notin \text{CLICA}$ .

### Teorema

CLICA és NP-complet

Per demostrar la NP-completesa de CLICA cal veure que:

- CLICA ∈ NP
- CLICA és NP-difícil

## (1) CLICA ∈ NP

Sigui  $(G, k)$  una entrada de CLICA.

- Els **testimonis** són els vèrtexs dels subgrafs complets de  $G$  de  $k$  vèrtexs (en l'exemple anterior, el conjunt  $C = \{3, 4, 5, 6\}$ )
- El **polinomi**  $p(n) = n$  és suficient perquè un testimoni  $C$  compleix  $|C| \leq |(G, k)| = p(|(G, k)|)$
- Podem **verificar** en temps polinòmic si un conjunt  $C$  és un testimoni: tot parell de vèrtexs de  $C$  ha de formar una aresta en  $G$  ( $\binom{|C|}{2} \leq n^2$  comprovacions)

## (2) CLICA és NP-difícil

Demostrarem que CNF-SAT  $\leq^p$  CLICA. Aleshores,

- Com que CNF-SAT és NP-difícil, tot  $S \in \text{NP}$  compleix  $S \leq^p \text{CNF-SAT}$
- Per transitivitat, tot  $S \in \text{NP}$  complirà  $S \leq^p \text{CLICA}$
- Per tant, CLICA és NP-difícil

# Problemes NP-complets

## Proposició

Sigui  $A$  un problema NP-complet i  $B$  un problema tal que  $B \in \text{NP}$  i  $A \leq^P B$ . Llavors,  $B$  també és NP-complet.

En general, tenim que  $F \in \text{CNF-SAT} \Leftrightarrow (G, m) \in \text{CLICA}$ :

- Com que  $A$  és NP-difícil, qualsevol  $S \in \text{NP}$  satisfà  $S \leq^P A$
- Per transitivitat, qualsevol  $S \in \text{NP}$  satisfà  $S \leq^P B$
- Per tant,  $B$  és NP-difícil

## CNF-SAT $\leq^P$ CLICA

Sigui  $F$  una fórmula booleana en CNF amb:

- clàusules  $C_1, \dots, C_m$
- literals  $l_1, \dots, l_r$

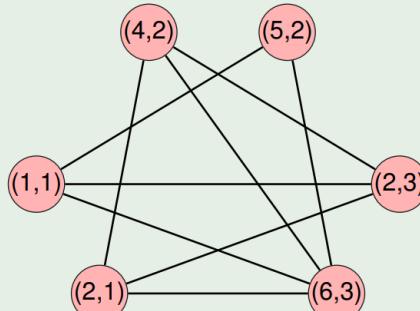
L'algorisme de reducció és  $\mathcal{R}(F) = (G, m)$ , on  $G = (V, E)$  és:

- $V = \{(i, j) \mid l_i$  apareix a  $C_j\}$   
(Els vèrtexs representen ocurrències de literals en clàusules)
- $E = \{ \{(i, j), (k, l)\} \mid j \neq l \wedge \neg l_i \neq l_k\}$   
(Les arestes representen parells de literals que poden ser certs alhora)

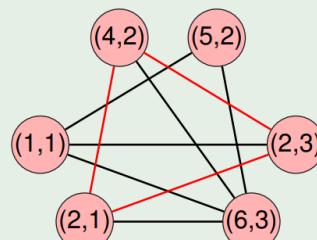
$F(x_1, x_2, x_3) = C_1 \wedge C_2 \wedge C_3$ , on

- $C_1 = (x_1 \vee x_2)$ ,  $C_2 = (\neg x_1 \vee \neg x_2)$ ,  $C_3 = (x_2 \vee \neg x_3)$
- $l_1 = x_1$ ,  $l_2 = x_2$ ,  $l_3 = x_3$ ,  $l_4 = \neg x_1$ ,  $l_5 = \neg x_2$ ,  $l_6 = \neg x_3$

La reducció és  $\mathcal{R}(F) = (G, 3)$ , on  $G$  és el graf



Exemple anterior amb  $l_2 = 1$ ,  $l_4 = 1$



## Definicions

- $H$  és un **subconjunt independent** de  $G$  si consisteix en vèrtexs aïllats
- $H$  és un **recobriment de vèrtexs** de  $G$  si té un extrem de tota aresta de  $G$

Molts NP-complets tenen “casos particulars” que són a P.

### Satisfactibilitat $k$ -fitada ( $k$ -SAT)

Donada un fórmula booleana en CNF de  $n$  variables amb  $\leq k$  literals per clàusula, determinar si és satisfactible.

### Satisfactibilitat 1-fitada (1-SAT)

Donada un fórmula booleana en CNF  $F$  de  $n$  variables amb 1 literal per clàusula, determinar si és satisfactible.

Per exemple,

$$F(x, y, z, t) = (x) \wedge (\neg y) \wedge (z) \wedge (\neg t).$$

1-SAT és **decidable en temps polinòmic** amb l'algorisme següent:

```
entrada F
si F conté dos literals contradictoris llavors
    retornar 0
si no
    retornar 1
```

### Satisfactibilitat 2-fitada (2-SAT)

Donada un fórmula booleana en CNF  $F$  de  $n$  variables amb  $\leq 2$  literals per clàusula, determinar si és satisfactible.

Per exemple,

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z).$$

2-SAT és **decidable en temps polinòmic**

- transformant la fórmula en un graf dirigit
- aplicant al graf un algorisme de camins

## Esbós de l'algorisme

Donada una fórmula booleana en 2-CNF

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z)$$

es reescriu fent servir implicacions

$$F(x, y, z) = (\neg x \Rightarrow y) \wedge (z \Rightarrow x) \wedge (x \Rightarrow y) \wedge (y \Rightarrow \neg z)$$

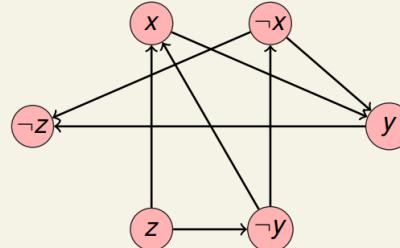
que es basen en les equivalències

- $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$
- $(a) \equiv (a \vee a) \equiv (\neg a \Rightarrow a) \equiv (a \Rightarrow \neg a)$

La fórmula booleana amb implicacions

$$F(x, y, z) = (\neg x \Rightarrow y) \wedge (z \Rightarrow x) \wedge (x \Rightarrow y) \wedge (y \Rightarrow \neg z)$$

es transforma en un digraf  $D_F$  i s'aplica el lema següent.



## Lema

$F$  és insatisfactible si i només per a alguna variable  $x$ ,  $D_F$  té camins de  $x$  a  $\neg x$  i de  $\neg x$  a  $x$ .

## Satisfactibilitat 3-fitada (3-SAT)

Donada un fórmula booleana en CNF  $F$  de  $n$  variables amb  $\leq 3$  literals per clàusula, determinar si és satisfactible.

## Teorema

3-SAT és NP-complet.

Per demostrar-ho, cal provar:

- ① 3-SAT  $\in$  NP  
(semblant a CNF-SAT)
- ② 3-SAT és NP-difícil  
(demostrem  $\text{CNF-SAT} \leq^P \text{3-SAT}$ )

## CNF-SAT $\leq^P$ 3-SAT

El mètode següent transforma un fórmula booleana en CNF en una altra d'equivalent en 3-CNF.

Donada una f.b.  $F$  en CNF,

- ① Sigui  $F'$  una f.b. buida
- ② Per a cada clàusula  $C = (a_1 \vee \dots \vee a_k)$  de  $F$ :
  - si  $k \leq 3$ , afegir  $C$  a  $F'$
  - si  $k > 3$ , afegir a  $F'$  la clàusula  $(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \dots (\neg z_{k-3} \vee a_{k-1} \vee a_k)$  on  $z_1, \dots, z_{k-3}$  són variables noves.
- ③ Retornar  $F'$

Donada una clàusula de cinc literals  $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$ , la reducció retorna

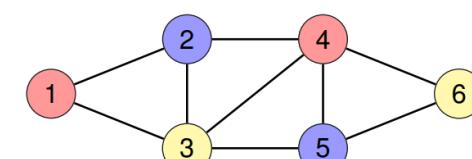
$$C' = (a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee a_5).$$

## Definició

Un graf  $G = (V, E)$  de  $n$  vèrtexs és ***k*-colorable** si existeix una funció total

$$\chi : V \rightarrow \{1, \dots, k\}$$

t.q.  $\chi(u) \neq \chi(v)$  per a cada aresta  $\{u, v\} \in E$ . La funció  $\chi$  és una ***k*-coloració**.



3-coloració

## ***k*-Colorabilitat (*k*-COLOR)**

Donat un graf  $G$ , determinar si és *k*-colorable.

Per als casos següents se'n coneixen algoritmes polinòmics:

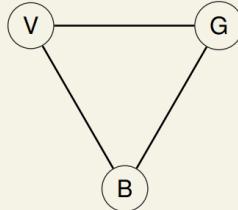
- 1-COLOR
- 2-COLOR

Per a 3-COLOR, demostrem la NP-completesa:

- Ja hem vist que 3-COLOR  $\in$  NP
- Ara veurem que és NP-difícil amb una reducció des de 3-CNF-SAT

## CNF-SAT $\leq^p$ 3-COLOR

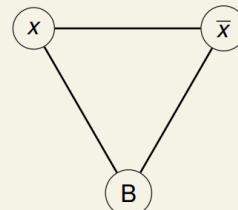
- Hi haurà 3 vèrtexos especials anomenats V, G, B.



Podem suposar que, en qualsevol coloració, tenen els colors:

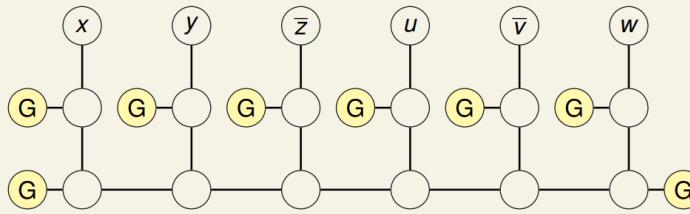
$$V \rightarrow \text{vermell}, G \rightarrow \text{groc}, B \rightarrow \text{blau}$$

- Afegim un vèrtex per cada literal i connectem cada literal i el seu complementari al vèrtex B.



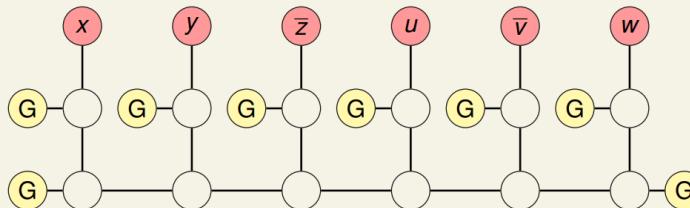
- Per cada clàusula, afegim un subgraf com el següent. En aquest cas

$$(x \vee y \vee \bar{z} \vee u \vee \bar{v} \vee w).$$



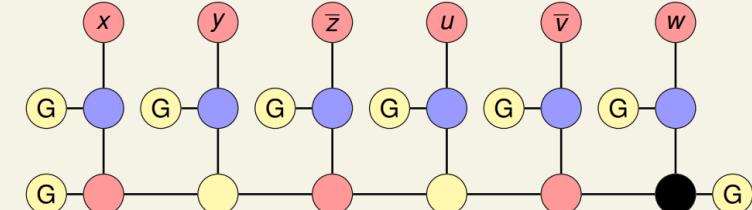
**Propietat:** Una coloració dels vèrtexos superiors amb vermell o groc es pot estendre a una 3-coloració global si i només si almenys un és groc.

Si tots els de dalt són vermellos...



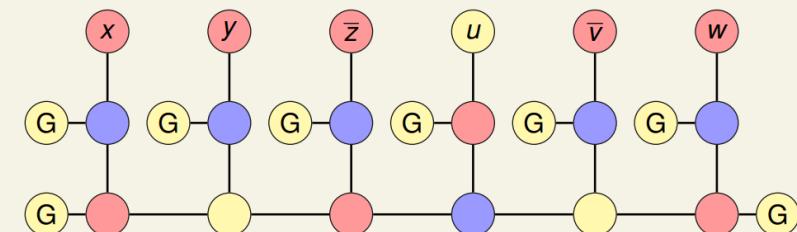
...no podem completar la 3-coloració.

Si tots els de dalt són vermellos...



...no podem completar la 3-coloració.

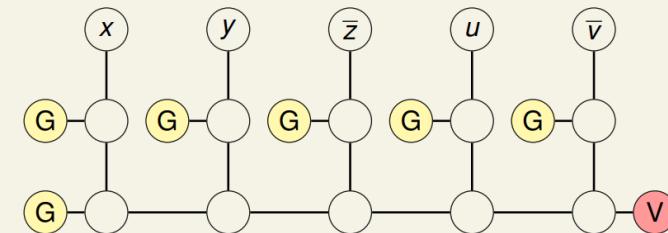
Si almenys un de dalt és groc...



...podem obtenir una 3-coloració global.

En cas que el nombre de literals sigui senar, el vèrtex de la dreta serà V. Per exemple,

$$(x \vee y \vee \bar{z} \vee u \vee \bar{v})$$



Si G és el graf amb tots els vèrtexos i arestes definitos abans, llavors

$$F \text{ és satisfactible} \Leftrightarrow G \text{ és 3-colorable.}$$

Com que G es pot construir en temps polinòmic, tenim que

$$\text{CNF-SAT} \leq^p \text{3-COLOR}.$$

### Teorema

3-COLOR és NP-complet.

Per la resta de problemes  $k$ -COLOR, podem observar el següent.

### Proposició

Per a tot  $k > 3$ ,  $3\text{-COLOR} \leq^P k\text{-COLOR}$ .

La reducció consisteix, donat un graf  $G$ , a afegir-li un subgraf complet de  $k - 3$  vèrtexs connectats a tots els de  $G$ .

### Corol·lari

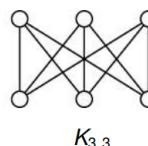
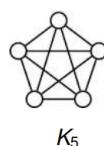
Per a tot  $k > 3$ ,  $k\text{-COLOR}$  és NP-complet.

Per tant, tenim:

- $k\text{-COLOR} \in P$  per a  $k \leq 2$
- $k\text{-COLOR}$  és NP-complet per a  $k \geq 3$

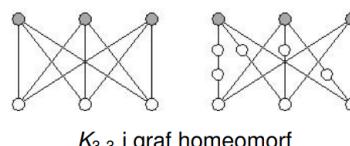
### Definició de planaritat

Un graf és planar si es pot dibuixar en el pla sense creuaments d'arestes.



### Teorema de Kuratowski

Un graf és planar si i només si no conté cap subgraf homeomorf a  $K_5$  o  $K_{3,3}$ .



$K_{3,3}$  i graf homeomorf

### Teorema de Kuratowski

Un graf és planar si i només si no conté cap subgraf homeomorf a  $K_5$  o  $K_{3,3}$ .

### Test de planaritat

#### • Força bruta: $O(n^6)$

- Contreure arestes de grau 2
- Comprovar si cada subconjunt de 5 vèrtexs és un  $K_5$
- Comprovar si cada subconjunt de 6 vèrtexs és un  $K_{3,3}$

#### • Eficient: $O(n)$

- Aplicar DFS

### k-Colorabilitat planar ( $k$ -COLOR-PL)

Donat un graf planar  $G$ , determinar si és  $k$ -colorable.

La planaritat es pot comprovar en temps polinòmic

### Corol·lari

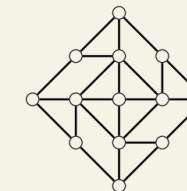
$3\text{-COLOR-PL}$  és NP-complet.

Per tant, tenim:

- $k\text{-COLOR-PL} \in P$  per a  $k \leq 2$
- $3\text{-COLOR-PL}$  és NP-complet
- $k\text{-COLOR-PL} \in P$  per a  $k \geq 4$  (pel teorema dels 4 colors)

### $3\text{-COLOR} \leq^P 3\text{-COLOR-PL}$

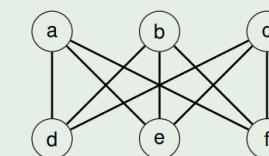
Donat un graf  $G$ , considerem un dibuix de  $G$ , possiblement amb creuaments d'arestes. Cada creuament el substituïm pel giny  $W$ :



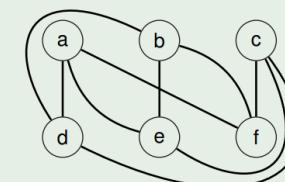
$W$  té propietats interessants:

- 1 en tota 3-coloració de  $W$ , els extrems opositius tenen el mateix color
- 2 tota coloració dels extrems on els opositius tenen el mateix color es pot estendre a una 3-coloració de  $W$

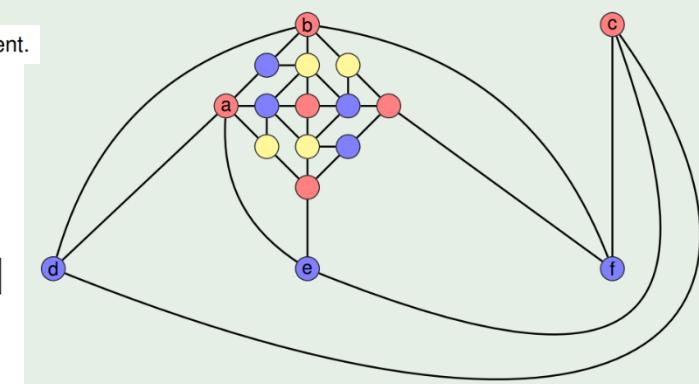
Suposem que tenim el graf  $K_{3,3}$  com a entrada de  $3\text{-COLOR}$ :



Però considerem el dibuix següent que redueix els creuaments a un:



Una 3-coloració de  $K_{3,3}$  induceix una 3-coloració de (i a l'inrevés):



Fins ara hem vist l'arbre de reduccions següent.

