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 $\Theta(n)$

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

Properties:

- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} < +\infty \rightarrow g = O(f)$ → **To prove that an algorithm cost** (Ex: $f(n) = \Theta(\mathrm{n})$)
- 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 → $\Theta(\mathrm{n})$ Because the parameter is copied
- Pass a parameter by reference (&) → $\Theta(1)$

Analysis of iterative algorithms

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

`if` ***B*** `then` ***S1*** `else` ***S2***	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

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

Analysis of recursive algorithm

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

Where $c \ge 1$, $g(n)$ is the cost of base case, a is the number of recursive calls and $f(n) = \Theta(\mathrm{n}^{\mathrm{k}})$ 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 $f(n) = \Theta(\mathrm{n}) \rightarrow n^k = n \rightarrow k = 1$

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

Where $a \ge 1$, $b > 1, g(n)$ is the cost of base case, a is the number of recursive calls and $f(n) = \Theta(\mathrm{n}^{\mathrm{k}})$ 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 {
      (x₁, x₂, … , xₖ):= DIVIDE(x)
        for := 1 to k {
           yᵢ := DIVIDE_AND_CONQUER(xᵢ)
        }
        return COMBINE(y₁, y₂, … , yₖ)
   }
}
```

Cost of Divide and conquer algorithms

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

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
→ $M(n) = \Theta(n^{\log_2 3})$

$x =$ | 01… | 11… | $y =$ | 10… | 10… |
|---|---|---|---|---|
| A | B | | C | D |

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*×*n* → $M(n) = \Theta(n^{\log_2 7})$

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

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

```
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 → $\Theta(\mathrm{n}\log n)$

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 → $\mathrm{O}(n^2)$

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 $A[l..u]$,
$l \le j \le u$
  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 → $\Theta(n)$, worst case → $\Theta(n^2)$

Dictionaries

BST (Binary Search Tree) → 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 $k = KEY(x)$ → The search is successful.
- If $k < KEY(x)$ → If there exist an element in T with key k it must be stored in the left subtree of T → We must make a recursive call on the left subtree of T to continue the search.
- If $k > KEY(x)$ → 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 $k = KEY(x)$ → Overwrite value.
- If $k < KEY(x)$ → Insert in the left subtree.
- If $k > KEY(x)$ → 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) → Remove the node.
- If the node have only one non-empty subtree → Replace the non-empty subtree by father.
- If the node have both subtrees →
  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;
}
```

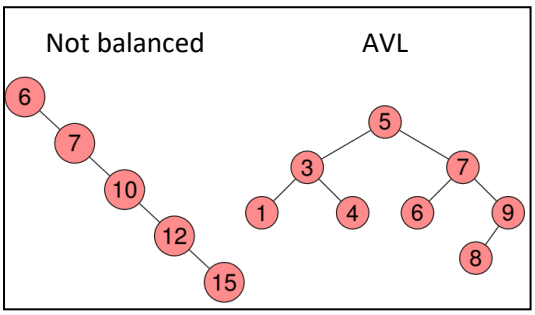
Cost

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

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

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



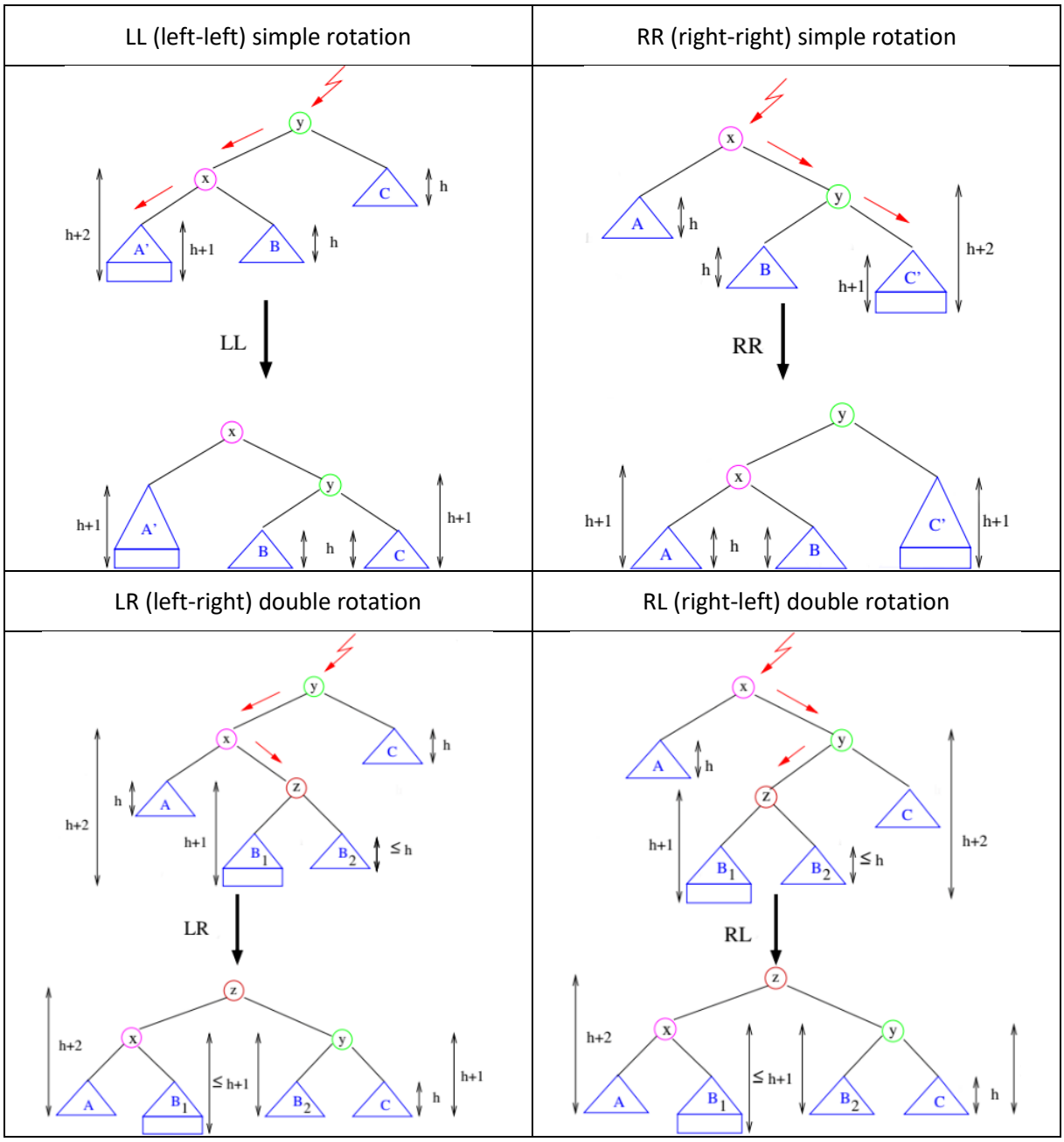
An AVL T is a binary search tree that is either empty or

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_alv* 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, nod_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 <key, value>) 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.

0		→ 13	→ 0
1			
2			
3			
4		→ 30	→ 17 → 4
5			
6		→ 19	→ 6
7			
8			
9			
10		→ 23	→ 10
11			
12		→ 25	→ 12

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

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

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 <key, value>.
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 $i_1, i_2, \ldots$
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:** $i_1 = i_0 + 1, i_2 = i_1 + 1, \ldots$

0	0		0	0	occupied		0	0	occupied
1			1	13	occupied		1	13	occupied
2			2		free		2	25	occupied
3			3		free		3		free
4	4		4	4	occupied		4	4	occupied
5			5	17	occupied		5	17	occupied
6	6	→	6	6	occupied	→	6	6	occupied
7			7	19	occupied		7	19	occupied
8			8		free		8	30	occupied
9			9		free		9		free
10	10		10	10	occupied		10	10	occupied
11			11	23	occupied		11	23	occupied
12	12		12	12	occupied		12	12	occupied

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

Priority queues

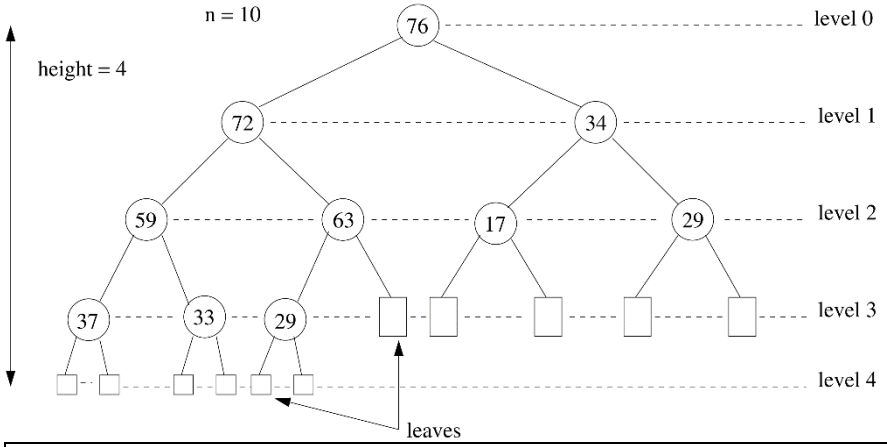
Stores a collection of elements, each one has a priority associated with it, and we order by this value. Priority queues support the insertions of new elements and the query and removal of an element of minimum (or maximum) priority.

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.



A heap of size n has height $h = \log(n+1)$

```
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::insert(cons Elem& x, cons Prio& p) {

   ++nelems;
   h.push_back(make_pair(x, p));
   siftup(nelems);
}

template <typename Elem, typename Prio>
void PriorityQueue<Elem,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 Elem, typename Prio>
void PriorityQueue<Elem,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 Elem, typename Prio>
void PriorityQueue<Elem,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 Elem's; Elem == priorities
// this is a.k.a. as heapify
template <typename Elem>
void make_heap(Elem v[], int n) {
   for (int i = n/2; i > 0; --i)
      sink(v, n, i);
}

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

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elem>
void heapsort(Elem 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 $u, v \in V$.

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

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

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

The number of arcs m: $\ 0 \leq m \leq n(n-1)$

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

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)→ $\deg(v)$.

- **In-degree:** number of successors of the vertex $u \rightarrow$ in-deg(u) .
- **Out-degree**: number of predeceddors of the vertex $u \rightarrow$ 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 form 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 witch each edge/arc bears a **label**, these are numeric (integers, rational, real numbers). When a graph is labelled then we called **weighted graphs**, and the label of the edge is called its weight.

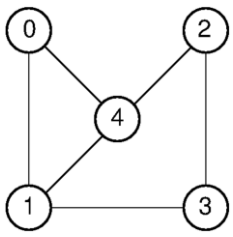
Implementation of Graphs

Adjancency 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.

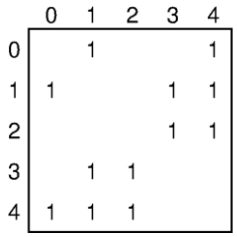
- Entry $A[i][j]$ is a Boolean indicating whether (i, j) is an edge/arc or not.
- For a weighted (di)graph $A[i][j]$ stores the label assigned to the edge/arc (i, j).

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

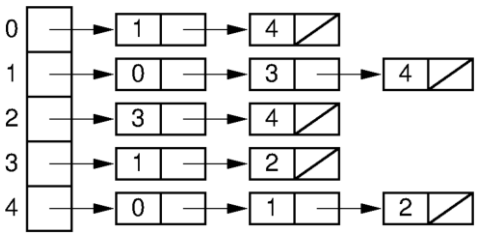
- We use an array or vector T such that for a vertex u, $T[u]$ points to a list of edges incident to u or a list of the vertices $v \in V$ which are adjacent to u.
- For digraphs, we will have the lists of successors of a vertex u, for all vertices u, and, possibly, the list of predecessors of a vertex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



Graph



Adjancency matrice



Adjancency list

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

class Graph {
// undirected graphs with V = {0, ..., n-1}
public:
// create en empty graph (no vertices and no edges)
   Graph();

// create en 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.

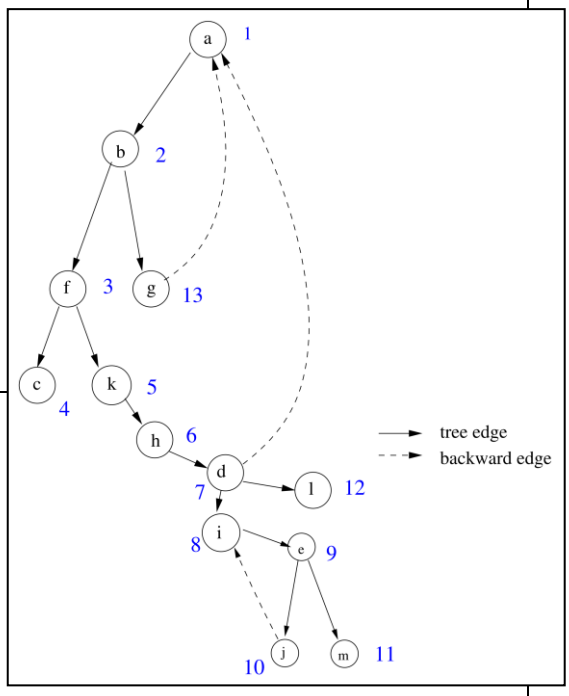
- Find the connected components of an undirected graph.
- Find the strongly connected components of a digraph.
- Find whether a graph contains cycles or not.
- Decide if a graph is bipartite or not (equivalently, if a graph is 2-coloreable or not)
- Decide whether a graph is biconnected or not (a graph 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 ∈ V (G) do
        visited[v] := false
        ndfs[v] := 0; ninv[v] := 0
    end for
    num_dfs := 0; num_inv := 0
    for v ∈ V (G) do
        if ¬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 ∈ G.ADJACENT(v) do
        if ¬visited[w] then
            PRE-VISIT-EDGE(v, w)
            DFS-REC(G, w, v)
            POST-VISIT-EDGE(v, w)
        else
// If w ≠ father there is a cycle (that conatins 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 DFS(v, …) 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, $ndfs[v] < ndfs[w]$ and $ndfs[w]$ is open.
- **Forward edges**: from currently visited vertex v to a previously visited descendant (from below) w; $ndfs[v] < ndfs[w]$.
- **Cross edges**: from currently visited vertex v to a previously visited vertex w in the same DFS tree or a different traversal tree; $ndfs[w] < ndfs[v]$, but w is already closed ($ninv[w] \neq 0$).

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 ∈ G do pred[v] := 0
   end for
   for v ∈ G do
      for w ∈ G.SUCCESSORS(v) do pred[w] := pred[w] + 1
      end for
   end for
   Q := ∅
   for v ∈ G do
      if pred[v] = 0 then
         Q.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 ∈ G do
      D[v] := 1  // D[v] = 1 indicates that v hasn't been visited
   end for
   Q := ∅; Q.PUSH(s); D[s] := 0
   while ¬Q.EMPTY() do
      v := Q.POP()
      VISIT(v)
      for w ∈ G.ADJACENT(v) do
         if D[w] = −1 then
            D[w] := D[v] + 1
            Q.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 $\Theta\big((m+n)\log n\big)$

```
template <class Elem, class Prio>
class DijkstraPrioQueue<Elem,Prio> {
public:
   DijkstraPrioQueue(int n = 0);
   void insert(const Elem& e, const Prio& prio);
   Elem min() const;
   void remove_min();
   Prio prio(const Elem& e) const; // returns the priority of element v
   bool contains(const Elem& e) const;
   void decrease_prio(const Elem& 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 $\boldsymbol{P}$ **is decidable in time** $\boldsymbol{t}$, 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 ∈ P
- REACHABILITY ∈ P
- PRIMALITY ∈ P
- SHORTEST-PATH ∈ P
- 2-COLORABILITY ∈ P
- 3-COLORABILITY ∈ EXP; also ∈ P? (not known to be P)
- LONGEST-PATH ∈ EXP; also ∈ P? (not known to be P)
- TSP ∈ EXP; also ∈ P? (not known to be P)
- GENERALIZED-CHESS ∈ EXP
- GENERALIZED-CHECKERS ∈ EXP
- GENERALIZED-GO ∈ 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.