# Eléments d'algorithmique

8 novembre 2019

1. Algorithmes, programmes et fonctions

2. Algorithmes à accumulateurs
   - Recherche de minimum, de maximum
   - Somme, produit sur les éléments d'une liste
   - Compter des occurences

3. Recherches
   - Recherche séquentielle dans une liste.
   - Recherche d'un mot dans une chaîne de caractères
   - Recherche par dichotomie dans une liste triée.

# Extrait du programme officiel

L'introduction à l'algorithmique contribue à apprendre à l'étudiant à analyser, à spécifier et à modéliser de manière rigoureuse une situation ou un problème. Cette démarche algorithmique procède par décomposition en sous-problèmes et par affinements successifs. L'accent étant porté sur le développement raisonné d'algorithmes, leur implantation dans un langage de programmation n'intervient qu'après une présentation organisée de la solution algorithmique, indépendante du langage choisi.

# Extrait du programme officiel

Pour faire mieux comprendre la notion d'algorithme et sa portée universelle, on s'appuie sur un petit nombre d'algorithmes simples, classiques et d'usage universel, que les étudiants doivent savoir expliquer et programmer, voire modifier selon les besoins et contraintes des problèmes étudiés :

- recherche dans une liste, recherche du maximum dans une liste de nombres, calcul de la moyenne et de la variance ;
- recherche par dichotomie dans un tableau trié ;
- recherche d'un mot dans une chaîne de caractères (On se limite ici à l'algorithme « naïf »).

# Extrait du programme officiel

Les principales capacités développées dans cette partie de la formation sont :

- comprendre un algorithme et expliquer ce qu'il fait ;
- modifier un algorithme existant pour obtenir un résultat différent ;
- concevoir un algorithme répondant à un problème précisément posé ;
- expliquer le fonctionnement d'un algorithme ;
- traduire un algorithme dans un langage de programmation.

# Notion d'algorithme

- Un algorithme est une succession d'opérations élémentaires (calculs d'expressions, affectations, instructions conditionnelles, boucles...) effectuées méthodiquement afin de réaliser une tâche donnée en un nombre fini d'opérations.
  C'est l'aspect « méthodique » qui est important : étant donnés une situation et un but, l'algorithme doit décrire les opérations à effectuer, possiblement en les répétant pour que, connaissant la situation, on arrive au but.

# Notion d'algorithme

- Un algorithme est une succession d'opérations élémentaires (calculs d'expressions, affectations, instructions conditionnelles, boucles...) effectuées méthodiquement afin de réaliser une tâche donnée en un nombre fini d'opérations.
- Un algorithme répond à un certain *cahier des charges*, *i.e.* à certaines *spécifications*, notamment en termes de données d'entrée (la situation initiale) et de sortie (le but recherché).

# Notion d'algorithme

- Un algorithme est une succession d'opérations élémentaires (calculs d'expressions, affectations, instructions conditionnelles, boucles...) effectuées méthodiquement afin de réaliser une tâche donnée en un nombre fini d'opérations.
- Un algorithme répond à un certain *cahier des charges*, *i.e.* à certaines *spécifications*, notamment en termes de données d'entrée (la situation initiale) et de sortie (le but recherché).
- Plusieurs algorithmes, différents en essence peuvent répondre à un même cahier des charges. Par exemple, les algorithmes de tri, vus en 2eme année, partant d'une situation, une liste de nombres ont tous pour but de trier cette liste. Le choix de tel ou tel algorithme dépend alors d'autres paramètres tels que la quantité de mémoire utilisée (complexité spatiale), la vitesse d'exécution (complexité temporelle), compléxités qui s'évaluent le plus souvent en termes de fonction de la taille des données en entrée.

# Notion d'algorithme

Comme exemples classiques d'algorithmes mathématiques, on peut citer :

- l'algorithme (classes primaires) qui étant données les écritures décimales de deux nombres (situation initiale) calcule l'écriture décimale de leur somme (but) ; l'algorithme donnant l'écriture décimale de leur produit. Ce sont des algorithmes dont les données d'entrée et de sortie sont en fait des mots (formés de chiffres) ;
- l'algorithme de Gauss en algèbre linéaire ; la donnée d'entrée est un système linéaire avec second membre (décrits par exemple sous forme matricielle), la sortie est une description de l'ensemble des solutions du système (qui peut là encore se décrire à l'aide de matrices, en spécifiant une solution particulière et une base de l'espace des solutions du système linéaire associé ;
- l'algorithme de Gauss–Jordan, variante du précédent, permettant de calculer l'inverse, si elle existe, d'une matrice carrée ;
- l'algorithme d'Euclide en arithmétique ; la donnée d'entrée est un couple de nombres entiers (non nuls), la sortie est leur PGCD.

# Notion d'algorithme

Un algorithme (en tout cas ceux que nous verrons) utilise les mêmes éléments fondamentaux de contrôle des opération qu'un langage de programmation standard.
Un algorithme est par ailleurs indépendant du langage de programmation utilisé : il est théoriquement possible de l'exprimer en langage « courant ».
On va essayer de donner au maximum des descriptions d'algorithme en *pseudo-code*. , avant de l'implémenter en Python ou dans n'importe quel autre langage de programmation.
Traduire un algorithme (décrit en pseudo-code) en un programme (ou une fonction dans un programme) Python s'appelle *implémenter cet algorithme en Python*.
Cela implique que vous devez être à l'aise avec les éléments fondamentaux de ces langages : l'heure n'est plus à l'écriture de code correct du point de vue de la syntaxe, mais à la mise en pratique du langage à une fin donnée.

# Notion d'algorithme

Une remarque importante : tous les algorithmes présentés ici sont déjà implémentés par des fonctions Python de la bilbiothèque standard. Le but est de comprendre comment ils sont implémentés !
L'implémentation dans une fonction Python standard est souvent faite en un langage de plus bas niveau (C ou FORTRAN), compilé. En termes de rapidité, nos nouvelles fonctions seront donc particulièrement inefficaces.

Un algorithme à accumulateur(s) fonctionne sur le principe suivant :

1. Le(s) accumulateur(s) sont initialisés grâce aux données d'entrée ;
2. une boucle fait évoluer la valeur de ce(s) accumulateurs ;
3. la/les données de sorties sont présentes dans l'un des accumulateurs en sortie de boucle

# Maximum, minimum

On recherche le maximum d'une liste de $N$ nombres $L = (\ell_i, i \in \{1, \ldots, N\})$. L'algorithme en langage courant est le suivant

1. Un accumulateur $M$ est initialisé à la première valeur de la liste : $M = \ell_1$ ;
2. On parcourt en une boucle tous les éléments $\ell$ de la liste $L$. Pour chacun d'entre eux, si $M < \ell$, on met à jour $M$ en lui donnant la valeur $\ell$
3. En fin de boucle $M$ est le maximum des éléments de $L$.

NB : La fonction Python native est `max`.

# Maximum, minimum

Ce qui donne en pseudo-code

---

**Données:** $L = (L[k])_{1 \leq k \leq n}$ est un tableau de réels, $n \geq 1$.
**Résultat:** $M$, le maximum des éléments de $L$.
**début**
    $M \leftarrow L[1]$;
    **pour** $i = 2$ *à* $n$ **faire**
        **si** $L[i] > M$ **alors**
            $M \leftarrow L[i]$;

---

# Maximum, minimum

En Python, il faut veiller par exemple à avoir le bon indiçage des listes, récupérer le nombre d'éléments de la liste, l'algorithme peut aussi être mis en fonction, ce qui clarifie les données d'entrée et de sortie, etc...., Une première version Python [1] :

```
def maximum(L):
    """
    Entrée: une liste L (de nombres)
    Sortie :le maximum de la liste L
    """
    M=L[0]
    for i in range(1,len(L)):
        if L[i] > M :
            M = L[i]
    return M
```

# Maximum, minimum

On peut aussi jouer sur le fait qu'en Python, on peut boucler sur les éléments d'une liste. (Ce n'est pas le cas de beaucoup de langages !)

```
def maximum(L):
    """
    Entrée: une liste L (de nombres)
    Sortie :le maximum de la liste L
    """
    M=L[0]
    for ell in L :
        if ell > M :
            M = ell
    return M
```

# Maximum, minimum

La complexité (temporelle) de cet algorithme est en $O(n)$ où $n$ est la taille des données ( *i.e.* la longueur de la liste), ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $n$ opérations élémentaires.
La complexité spatiale de cet algorithme est en $O(1)$, ce qui signifie que, hormis la taille des données, la quantité de mémoire supplémentaire utilisée ne dépend pas de la taille de ces données.
En exercices :

1. Ecrire un algorithme calculant le minimum d'une liste, écrire (sur feuille) une fonction Python l'implémentant.
2. Ecrire un algorithme calculant le maximum d'une liste ainsi que le premier indice pour lequel ce maximum est atteint, écrire une fonction Python l'implémentant.
3. Modifier l'algorithme précédent pour calculer plutôt le dernier indice où le maximum est atteint.

# Sommes

On recherche la somme des termes d'une liste de $N$ nombres $L = (\ell_i, i \in \{1, \ldots, N\})$. L'algorithme en langage courant est le suivant

1. Un accumulateur $S$ est initialisé à 0, ce qui est la valeur conventionnelle de la somme sur une famille vide ;
2. On parcourt en une boucle tous les éléments $\ell$ de la liste $L$. Pour chacun d'entre eux, on met à jour $S$ en lui ajoutant $\ell$.
3. En fin de boucle $S$ est la somme de tous les éléments de $L$.

NB : La fonction Python native est `sum`.

# Sommes

Ce qui donne en pseudo-code

---

**Données:** $L = (L[k])_{1 \leq k \leq n}$ est un tableau de réels, $n \geq 1$.
**Résultat:** $S$, la somme des éléments de $L$.
**début**
    $S \leftarrow 0$;
    **pour** $i = 1$ *à* $n$ **faire**
        $S \leftarrow S + L[i]$;

---

# Sommes

Une première version Python[1] :

```
def somme(L):
    """
    Entrée: une liste L (de nombres)
    Sortie :la somme des termes de la liste L
    """
    S = 0
    for i in range(len(L)):
        S = S +L[i]
    return S
```

# Sommes

On peut aussi jouer sur le fait qu'en Python, on peut boucler sur les éléments d'une liste et utiliser la syntaxe +=. (Ce n'est pas le cas de beaucoup de langages !)

```
def somme(L):
    """
    Entrée: une liste L (de nombres)
    Sortie :la somme des termes de la liste L
    """
    S = 0
    for ell in L :
        S += ell
    return S
```

# Sommes

La complexité (temporelle) de cet algorithme est en $O(n)$ où $n$ est la taille des données ( *i.e.* la longueur de la liste), ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $n$ opérations élémentaires.
La complexité spatiale de cet algorithme est en $O(1)$, ce qui signifie que, hormis la taille des données, la quantité de mémoire supplémentaire utilisée ne dépend pas de la taille de ces données.

# Sommes

En exercices :

1. Ecrire un algorithme calculant le produit des éléments d'une liste de nombres, écrire (sur feuille) une fonction Python l'implémentant.
2. Ecrire une fonction Python calculant la chaine de caractères obtenue en concatenant les éléments d'une liste de chaines de caractères.
3. Ecrire une fonction `variance_definition(L)` retournant la variance de la liste numérique `L`. On rappelle que moyenne et variance d'une famile de nombres réels $(x_i)_{i\in\{1,\dots,n\}}$ sont données par les formules

$$\overline{x} := \frac{1}{n}.\sum_{k=1}^{n} x_i \text{ et } \sigma_x^2 := \frac{1}{n}.\sum_{k=1}^{n}(x_i-\overline{x})^2.$$

4. Une permutation de l'ensemble $\{0,\dots,n\}$ est une bijection de cet ensemble sur lui même. En supposant que l'on dispose d'une fonction de composition de deux telles bijections, écrire un algorithme calculant la permutation obtenue en composant les éléments d'une liste de telle permutations. Ecrire un ensemble[1] de fonctions Python implémentant ce type de calculs.

# Occurrences

Etant donnée une liste de $N$ nombres $L = (\ell_i, i \in \{1, \ldots, N\})$ où les $\ell_i$ sont des entiers entre 0 et $\ell^*$ ($\ell^*$ pas trop grand), on cherche à compter le nombre de fois où chaque entier entre 0 et $\ell^*$ apparait dans la liste $L$, *i.e.* chaque *occurrence* de $\ell$. L'algorithme en langage courant est le suivant

1. Une liste d'accumulateurs $S[\ell], 0 \leq \ell \leq \ell^*$ sont initialisés à 0 ;
2. On parcourt en une boucle tous les éléments $\ell$ de la liste $L$. Pour chacun d'entre eux, on ajoute 1 à l'accumulateur $S[\ell]$.
3. En fin de boucle $S[\ell]$ contient le nombre d'occurrences de $\ell$ dans la liste $L$.

# Occurrences

Ce qui donne en pseudo-code

---

**Données:** $L = (L[k])_{1 \leq k \leq n}$ est un tableau d'entiers entre 0 et $\ell^*$, $n \geq 1$.
**Résultat:** $S$, la liste des occurrences de chaque élément de la liste $L$.
**début**
    **pour** $\ell = 0$ *à* $\ell^*$ **faire**
        $S[\ell] \leftarrow 0$;
    **pour** $i = 1$ *à* $N$ **faire**
        $S[L[i]] \leftarrow S[L[i]] + 1$;

---

# Occurrences

Une première version Python[2] :

```
def occurences(L):
    S=[0]*(lstar+1)
    for ell in L:
        S[ell] += 1
    return S
```

# Occurrences

On peut aussi jouer sur le fait qu'en Python, il existe une structure plus souple que celle de liste pour faire ce travail, la structure de dictionnaire, qui est malheureusement hors-programme officiel. Un *dictionnaire*, ou *tableau associatif*, est une liste indexée par une famille de valeurs (des entiers, des chaînes de caractères, des couples ....), les clés, et qui à chaque clé, associe une valeur.

```
def occurences_dico(L):
    S=dict() #Un dictionnaire vide
    for ell in L:
        if ell in S:
            S[ell] +=1
        else:
            S[ell] = 0
    return S
```

# Occurrences

La complexité (temporelle) de cet algorithme est en $O(n)$ où $n$ est la taille des données ( *i.e.* la longueur de la liste), ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $n$ opérations élémentaires.
La complexité spatiale de cet algorithme est en $O(\ell^*)$, ce qui signifie que, hormis la taille des données, la quantité de mémoire supplémentaire utilisée est essentiellement proportionnelle au nombre de valeurs distinctes.

# Occurrences

En exercices :

1. Ecrire un algorithme calculant le nombre d'occurrences de chaque lettre dans une chaîne de caractères. (On pourra utiliser le fait qu'une lettre correspond à un nombre entier entre 0 et 255 via son code ASCII). Ecrire une fonction Python [2] l'implémentant.
2. Ecrire un algorithme calculant la(une) lettre(s) la(des) plus fréquente(s) dans une chaine de caractères. Ecrire une fonction Python l'implémentant.
3. Ecrire un algorithme calculant le nombre d'occurrences de chaque paire de lettres consécutives dans une chaine de caractères. Ecrire une fonction Python l'implémentant.
4. Une permutation de l'ensemble $\{0, \ldots, n\}$ est une bijection de cet ensemble sur lui même. Ecrire un algorithme vérifiant si oui ou non, une liste $L$ de $n+1$ nombres entre 0 et $n$ représente effectivement une telle permutation via la convention $i \mapsto L[i]$.
   Indication:Chaque nombre entre 0 et $n$ doit apparaitre exactement une fois dans a liste $L$

# Recherche séquentielle

Etant donnée une liste $L$ d'objets et un objet $x$, déterminer si oui ou non $x$ est dans $L$. L'algorithme en langage courant est le suivant

1. On parcourt en une boucle tous les éléments $\ell$ de la liste $L$ tant que $\ell$ est différent de $x$. Il y a deux possibilités :
   - Dès que $\ell$ vaut $x$, on arrête la boucle et on a calculé que oui, la liste $L$ contient $x$ ;
   - Tous les éléments de $L$ ont été passés en revue et on a calculé que non, la liste $L$ ne contient pas $x$ ;

NB : En Python, l'expression `x in L` fait ce travail. Cette expression n'est donc PAS une opération élémentaire.

# Recherche séquentielle

Ce qui donne en pseudo-code

---

**Données:** $L = (L[k])_{1 \leq k \leq n}$ est un tableau d'objets, $x$ un objet
**Résultat:** Oui ou Non, suivant que $L$ contient ou pas l'élément $x$.
**début**
    $i \leftarrow 1$;
    **tant que** $i \leq n$ **faire**
        **si** $L[i] = x$ **alors**
            **retourne** *Oui*
        $i \leftarrow i + 1$;
    **retourne** *Non*

---

# Recherche séquentielle

Une première version Python [3], fidèle au pseudo-code :

```
def recherche_seq(L,x):
    """
    Entree: L une liste, x un objet
    Sortie: True si x est dans L, False sinon
    """
    i = 0
    while i< len(L):
        if L[i] == x:
            return True
        i +=1
    return False
```

# Recherche séquentielle

Une boucle `for` est peut-être plus judicieuse...

```
def recherche_seq(L,x):
    """
    Entree: L une liste, x un objet
    Sortie: True si x est dans L, False sinon
    """
    for i in range(len(L)):
        if L[i] == x:
            return True
    return False
```

# Recherche séquentielle

On peut aussi jouer sur le fait qu'en Python, on peut boucler directement sur les éléments d'une liste

```
def recherche_seq(L,x):
    """
    Entree: L une liste, x un objet
    Sortie: True si x est dans L, False sinon
    """
    for ell in L :
        if ell == x:
            return True
    return False
```

# Recherche séquentielle

Ou, dernière variante, jouer sur le caractère paresseux des évaluations booléennes et tester qu'on est arrivé en fin de liste grâce à un compteur.

```
def recherche_seq(L,x):
    """
    Entree: L une liste, x un objet
    Sortie: True si x est dans L, False sinon
    """
    i = 0
    while (i < len(L)) and (L[i] != x) :
        i += 1
    return not(i == len(L))
```

# Recherche séquentielle

La complexité (temporelle) de cet algorithme est en $O(n)$ où $n$ est la taille des données ( *i.e.* la longueur de la liste), ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $n$ opérations élémentaires.
En exercice :

1. Écrire une fonction `recherche_indices` qui prend en argument une liste `L` et un objet quelconque `x`, et qui retourne la liste (éventuellement vide) des indices des occurrences de `x` dans `L`.

# Sous-chaine

Etant donnés une chaîne de caractères de longueur $N$,
$S = (s_n, n \in \{1, \ldots, N\})$ où les $s_n$ sont des caractères, un mot
$M = (m_k, k \in \{1, \ldots, K\})$ de longueur $K$, calculer si oui ou non le mot $M$ apparait dans la chaine $S$. L'algorithme en langage courant est le suivant

1. On parcourt en une boucle toutes les positions possibles dans $S$ et on teste si la sous-chaine de $S$ débutant à cette position vaut le mot $M$. Il y a deux possibilités :
   - Dès qu'on rencontre $M$, on arrête la boucle et on a calculé que oui, le mot $M$ est dans la chaîne $S$ ;
   - Si toutes les positions possibles dans $S$ ont été passées en revue et on a calculé que non, le mot $M$ n'est pas dans la chaîne $S$ ;

# Sous-chaine

Ce qui donne en pseudo-code

---

**Données:** $S = (S[n])_{1 \leq n \leq N}$ est une chaîne de caractères,
$M = (M[k])_{1 \leq k \leq K}$ un mot

**Résultat:** Oui ou Non, suivant que la chaîne $S$ contient ou pas le mot $M$.

**début**

- **pour** $n = 1$ *à* $N - K$ **faire**
  - $k \leftarrow 1$;
  - **tant que** $k \leq K$ *et* $S[n+k-1] = M[k]$ **faire**
    - $k \leftarrow k+1$;
  - **si** $k = K+1$ **alors**
    - **retourne** *Oui*
- **retourne** *Non*

---

# Sous-chaine

Une première version Python [3], fidèle au pseudo-code (la question de la gamme d'indices est cruciale !) :

```
def sous_chaine(S,M):
    """
    Entree: S une chaine, M un mot (chaine)
    Sortie: True M est dans S, False sinon
    """
    for n in range(len(S)-len(M)+1):
        k = 0
        while k<len(M) and S[n+k]==M[k]:
            k +=1
        if k == len(M) :
            return True
    return False
```

# Sous-chaine

On peut utiliser la possibilité de slicing et de comparaison directe des tranches (slices) pour éviter la boucle interne. Attention, cette comparaison n'est pas une opération élémentaire, elle cache une boucle...

```
def sous_chaine(S,M):
    """
    Entree: S une chaine, M un mot (chaine)
    Sortie: True M est dans S, False sinon
    """
    for n in range(len(S)-len(M)+1):
        if S[n:n+len(M)] == M :
            return True
    return False
```

# Sous-chaine

La complexité (temporelle) de cet algorithme est en $O(N.K)$ où $N$ est la taille de la chaine $S$, $K$ la taille du mot $M$, ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $N.K$ opérations élémentaires.
En exercice :

1. Écrire une fonction `recherche_indices` qui prend en argument une chaine `S` et un mot `M`, et qui retourne la liste (éventuellement vide) des indices de début des occurrences de `M` dans `S`.

# Dichotomie

Etant donnée une liste $L$ d'objets et un objet $x$, déterminer si oui ou non $x$ est dans $L$ et, variante, déterminer un indice $i$ tel que $L[i] = x$.
On a déjà rencontré ce problème, résolu par la méthode de recherche séquentielle en complexité $O(len(L))$.
Ce qu'on suppose de plus ici c'est que la liste $L$ est *triée par ordre croissant* et on cherche à obtenir un algorithme plus rapide. L'ordre est noté $\leq$. On suppose que l'ordre est total [3], cela implique que les éléments de $L$ se comparent les uns les autres mais surtout que $x$ peut-être comparé à chacun d'entre eux.
On imite la recherche usuelle dans un dictionnaire en la systématisant un peu etl'algorithme en langage courant est le suivant

# Dichotomie

1. On maintient tout le long de l'algorithme deux « curseurs », $a$ et $b$, de sorte que l'on cherche $x$ parmi les éléments de $L$ ayant un indice entre $a$ et $b$. A l'initialisation $a = 0$, $b =$indice maximal pour $L$. Si $x$ est élément de $L$, on a, tout le long de l'algorithme $L[a] \leq x \leq L[b]$.
2. Tant que $a \leq b$, on coupe la plage $\{a, \ldots, b\}$ en son « milieu » $c$ tel que $a \leq c \leq b$. Il y a trois possibilités :
   - Si $L[c] = x$, la recherche est finie et on retourne au choix, l'indice $i$ ou le booléen True ;
   - Si $L[c] < x$, comme $L$ est ordonnée par ordre croissant, cela signifie que $x$, s'il est dans la liste $L$, y est dans la plage d'indices $\{c+1, \ldots, b\}$ ; on déplace donc le curseur de gauche $a$ en posant $a = c+1$ ;
   - Si $L[c] > x$, comme $L$ est ordonnée par ordre croissant, cela signifie que $x$, s'il est dans la liste $L$, y est dans la plage d'indices $\{a, \ldots, c-1\}$ ; on déplace donc le curseur de droite $b$ en posant $b = c-1$ ;
3. Si on arrive à $a > b$ (sortie de la boucle), ce qui doit arriver par stricte décroissance des valeurs successives prises par $b-a$, et si $x$ est élément de $L$, c'est qu'il est impossible que $x$ soit élément de $L$ :

# Dichotomie

1. On maintient tout le long de l'algorithme deux « curseurs », $a$ et $b$, de sorte que l'on cherche $x$ parmi les éléments de $L$ ayant un indice entre $a$ et $b$. A l'initialisation $a = 0$, $b =$indice maximal pour $L$. Si $x$ est élément de $L$, on a, tout le long de l'algorithme $L[a] \leq x \leq L[b]$.
2. Tant que $a \leq b$, on coupe la plage $\{a, \ldots, b\}$ en son « milieu » $c$ tel que $a \leq c \leq b$. Il y a trois possibilités :
3. Si on arrive à $a > b$ (sortie de la boucle), ce qui doit arriver par stricte décroissance des valeurs successives prises par $b - a$, et si $x$ est élément de $L$, c'est qu'il est impossible que $x$ soit élément de $L$ :
   - Soit $a = c + 1$, $L[c] < x$, $c \leq b$, c'est à dire $a = b + 1 = c + 1$ et donc $L[b] < x$ : impossible
   - Soit $b = c - 1$, $L[c] > x$, $c \geq a$, c'est à dire $a = b + 1 = c$ et donc $L[a] > x$ : impossible

Ce qui donne en pseudo-code

# Dichotomie

**Données:** $L = (L[k])_{0 \leq k < n}$ est une liste croissante d'objets, $x$ un objet
**Résultat:** Oui ou Non :la liste $L$ contient ou pas l'élément $x$.

```
début
    a ← 0;
    b ← n − 1;
    si L[a] = x ou L[b] = x alors
        retourne Oui
    tant que a ≤ b faire
        c ← (a + b)//2;
        si L[c] = x alors
            retourne Oui
        si L[c] < x alors
            a ← c + 1;
        sinon
            b ← c − 1;
    retourne Non
```

# Dichotomie

Une seule version Python [3], fidèle au pseudo-code, hormis le fait que l'on écrit la version retournant un indice :

```
def recherche_dicho(L,x):
    """
    Entrée : L une liste en ordre croissant x un objet
             comparable à ceux de L
    Sortie : si existe, un indice i tel que L[i]=x,
             None sinon
    """
```

# Dichotomie

```
a = 0
b = len(L)-1
if L[a] == x:
    return a
if L[b] == x:
    return b
while a <= b :
    c= (a+b)//2
    #Pour voir les valeurs de a,c,b,...
    #print(a,c,b,L[a],L[c],L[b])
    if L[c] == x :
        return c
    if L[c] < x:
        a = c + 1
    else:
        b = c - 1
return None
```

# Dichotomie

La complexité (temporelle) de cet algorithme est en $O(\ln n)$ où $n$ est la taille des données ( *i.e.* la longueur de la liste), ce qui signifie que l'on a fait, à une constante multiplicative près, moins de $\ln n$ opérations élémentaires.