Algorithmes

1) Les algorithmes

Le mot **algorithme** vient d'un mathématicien Perse du IX ème siècle, **Al Khawarizmi**. Mais, des algorithmes, il en existait bien avant. En effet Euclide a donné une série d'instructions permettant de trouver le PGCD de deux nombres vers 300 avant JC !

Un algorithme est donc une suite **finie d'opérations ou d'instructions élémentaires** visant à résoudre un problème.

Exemples : Une recette de cuisine, une tactique sportive (tant que je gagne, je ne change pas de stratégie...) tout peut être vu sous forme d'algorithme.

Quand on écrit un algorithme, on utilise un langage dit **"langage naturel"** ("tant que", "si"...) ; ce langage naturel permet de passer facilement à un langage de programmation (Python, Java...). On dit alors que l'on **implémente** l'algorithme.

2) Complexité d'un algorithme

En informatique, la question de la **performance des algorithmes** est centrale. Si un problème est traité en un **temps raisonnable**, l'utilisateur est satisfait. Si un temps relativement long est nécessaire pour obtenir un résultat, se pose la question d'**une solution plus efficace**.

De manière générale, le traitement d'un certain volume de données requiert un **temps d'exécution lié à ce volume**. On s'attache rarement à la détermination exacte de ce temps, **une estimation** suffit.

C'est ce qu'on appelle l'étude du **coût** ou de la **complexité en temps**.

Exemple 1

```
s ← 0
pour i allant de 1 à n faire
    s ← s +i
```

Cet algorithme permet de calculer la somme des n premiers entiers positifs, non nuls.

Il y a n **passages dans la boucle** et à chaque passage, on fait **une addition et une affectation**, soit deux opérations.

Le temps d'exécution de cet algorithme est de $2n$, soit un ordre de grandeur de n. Il sera proportionnel au nombre n choisi. On dit que le coût est **linéaire**.

3) Preuve d'un algorithme

a. Terminaison

Un algorithme doit comporter un **nombre fini d'étapes**.

Or lorsque l'on utilise une **boucle conditionnelle** non bornée (Tant que), il faut être sûr que celle-ci s'arrête. C'est ce qu'on appelle l'étude de la **terminaison**.

Pour faire cette étude on utilise un **variant de boucle**. Il s'agit d'une quantité qui est entière et positive et qui décroît strictement à chaque tour de boucle. Cela provoquera alors l'arrêt de la boucle.

Exemple 2

```
q ← 0
tant que n >= 3 faire
    n ← n - 3
    q ← q + 1
```

Ici, le variant de boucle est n. La boucle s'arrêtera lorsque n sera inférieur à 3.

Or à chaque tour de boucle le nombre n diminue strictement. Donc, au bout d'**un nombre fini d'étapes**, n aura une valeur inférieure à 3, la boucle s'arrêtera. On est sûr, que cet algorithme se terminera.

b. Correction

Lorsqu'on écrit un algorithme, il est impératif de vérifier que celui-ci va produire un résultat **correct** dans le sens ou il sera conforme à ce qu'on attendait.

Pour cela, lors de l'utilisation d'une boucle, on va utiliser un **invariant de boucle**.

Il s'agit d'une propriété qui est vérifiée **avant l'entrée** dans la boucle, mais également **à chaque passage** de la boucle et **à sa sortie**.

Si l'on souhaite s'assurer de la correction de l'algorithme, on doit vérifier l'invariant.

Reprenons l'algorithme de l'exemple 1

Exemple 1

$s \leftarrow 0$
Pour i allant de 1 à n faire
    $s \leftarrow s + i$

L'invariant de boucle dans ce cas, est que *s est égal à la somme des entiers de 1 à i, où i est l'indice de la boucle ($s = 1 + 2 + ... + i$).*

A l'entrée de boucle, $s = 0$ ce qui correspond à une somme ne contenant aucune valeur.

Lors du passage dans la boucle pour l'indice i, on ajoute i à la somme précédente $s = 1 + 2 + ... + (i - 1)$ et on obtient $s = 1 + 2 + ... + i$.

L'invariant est vérifié.

À la sortie de la boucle, l'indice i est égal à n, et donc la somme s est égale à $1 + 2 + ... + n$: l'algorithme est correct, on a obtenu la somme des n premiers entiers positifs.

4) Parcours Séquentiel

Un **parcours séquentiel** signifie que le tableau est parcouru élément par élément, suivant l'ordre des éléments.

Exemple 3 :

On cherche la présence d'une valeur dans un tableau (méthode par balayage).

On compare la valeur recherchée successivement à toutes les valeurs du tableau.

L'algorithme s'arrête dès que l'élément est trouvé ou si la fin du tableau est atteinte.

Exemple 3

On note **tab** un tableau et **n** sa taille. On cherche si l'élément **e** est dans le tableau. On renvoie un booléen pour indiquer si l'élément est présent ou non :

$i \leftarrow 0$
Tant que i est inférieur à n et que tab[i] n'est pas égal à e
    $i \leftarrow i + 1$
Si i inférieur à n
    alors on retourne Vrai
    sinon on retourne Faux

Au pire des cas, l'élément e n'est pas dans le tableau est on le parcours en entier. Alors la **complexité en temps** de cet algorithme est égal à n, la taille du tableau à parcourir. Le **coût est linéaire**.

5) Algorithmes de Tri

Trier des données c'est les ranger suivant un ordre défini au préalable. (Ordre croissant pour des nombres, alphabétique pour des lettres...)

Lorsque les données d'un tableau sont triées, on peut déterminer plus rapidement si une valeur donnée est présente, ou bien qu'elle est la valeur la plus fréquente ; le tri d'un tableau est la base d'une grand nombres d'algorithmes.

Il existe plusieurs méthodes permettant de trier un tableau, nous allons étudier deux d'entre elles.

a. Tri par sélection

Le principe :

On parcours le tableau de la gauche vers la droite, en maintenant sur la gauche une partie déjà triée et à sa place définitive :

déjà trié	pas encore trié

À chaque étape, on cherche le plus petit élément dans la partie droite non triée, puis on l'échange avec l'élément le plus à gauche de la partie non triée.

Ainsi, la première étape va déterminer le plus petit élément et le placer à gauche du tableau. Puis la deuxième étape va déterminer le deuxième plus petit élément et le placer dans la deuxième case du tableau, et ainsi de suite.

Exemple :

15	7	11	2	25	8
2	7	11	15	25	8
2	7	11	15	25	8
2	7	8	15	25	11
2	7	8	11	25	15
2	7	8	11	15	25

Voir l'animation du site :

https ://professeurb.github.io/articles/tris/

Algorithme du tri par sélection :

On note n la **taille du tableau** à trier.

⋆ On va d'abord devoir rechercher le plus petit élément de la partie de droite (non triée) :

Minimum

```
min ← tab[1]
indice_min ← 1
Pour i allant de 2 à n faire
    si tab[i] < min
        alors min ← tab[i]
              indice_min ← i
```

⋆ Ensuite, on échange le minimum de la partie droite du tableau avec le premier élément de cette même partie :

Echange

```
tmp ← tab[1]
tab[1] ← min
tab[indice_min] ← tmp
```

Remarque : On se sert d'une variable temporaire **tmp** pour stocker la valeur **tab[1]**, car elle est remplacée par celle du **min**.

⋆ On obtient l'algorithme du tri par sélection :

Tri par sélection

```
Pour i allant de 1 à n-1 faire
    min ← tab[i]
    indice_min ← i
    Pour j allant de i+1 à n faire
        si tab[j] < min
            alors min ← tab[j]
                  indice_min ← j
    tmp ← tab[i]
    tab[i] ← min
    tab[indice_min] ← tmp
```

- Cet algorithme **se termine** car il s'agit de deux boucles finies imbriquées.

- Pour vérifier **la correction** de cet algorithme, on utilise l'**invariant de boucle** suivant :

"pour chaque indice i de la première boucle, les éléments de 1 à i du tableau sont triés et les éléments de i+1 à n sont tous supérieurs ou égaux aux éléments de 1 à i du tableau"

Pour i égal à 1, on cherche le plus petit élément du tableau, et on le place en première position ; tous les autres sont donc supérieurs ou égaux à celui-ci.

Au passage dans la boucle d'indice i, on a les éléments de 1 à i-1 du tableau qui sont triés et les éléments de i à n sont tous supérieurs ou égaux aux éléments de 1 à i-1 du tableau. On recherche le plus petit élément de la partie du tableau de i à n, et on le place en i, ainsi, les éléments de 1 à i du tableau sont triés et les éléments de i+1 à n sont tous supérieurs ou égaux aux éléments de 1 à i du tableau.

Enfin, lors du dernier passage dans la boucle i est égal à n-1 et les éléments de 1 à n-1 du tableau sont triés et l'élément n est supérieur ou égal aux autres, donc le tableau est trié.

On a prouvé la correction de l'algorithme.

- La **complexité** de l'algorithme :

Pour simplifier cette étude on va simplement regarder le nombre de comparaisons effectuées.

En effet, les opérations d'affectation se font en temps constant, ce temps ne dépend pas de la taille du tableau que l'on doit trier.

La boucle **pour** tourne pour i variant de 1 à n-1, donc (n-1) fois.

La seconde boucle fait varier j de i+1 à n, on fait donc (n-i) comparaisons.

Le temps d'exécution total est donc :

$$S = (n-1) + (n-2) + (n-3) + ... + 1$$

Une formule Mathématiques permet de montrer que $S = \dfrac{n(n-1)}{2}$.

L'ordre de grandeur de la complexité de l'algorithme est n^2, on dit que le coût du tri par sélection est **quadratique**.

Programme Python :

```
def tri_selection(tab):
    for i in range(len(tab)-1):
        min = tab[i]
        indice_min = i
        for j in range(i+1,len(tab)):
            if tab[j] < min:
                min = tab[j]
                indice_min = j
        tab[i], tab[indice_min] = min, tab[i]
    return tab


def tri_selection_bis(tab):
    for i in range(len(tab)-1):
        indice_min = i
        for j in range(i+1,len(tab)):
            if tab[j] < tab[indice_min]:
                indice_min = j
        tab[i], tab[indice_min] = tab[indice_min], tab[i]
    return tab
```

Visualiser l'exécution du programme sur le site **http ://pythontutor.com/visualize.html#mode=edit**

b. Tri par insertion

Le principe :

On parcours le tableau de la gauche vers la droite, en maintenant sur la gauche une partie déjà triée et à sa place définitive :

déjà trié	pas encore trié

A chaque étape on insère **la première valeur non encore triée**, dans la partie de gauche déjà triée. Pour cela on décale d'une case vers la droite tous les éléments déjà triés qui sont plus grands que la valeur à insérer, puis on place cette valeur dans la case ainsi libérée.

Exemple :

15	7	11	2	25	8
7	15	11	2	25	8
7	11	15	2	25	8
2	7	11	15	25	8
2	7	11	15	25	8
2	7	8	11	15	25

Voir l'animation du site :

https ://professeurb.github.io/articles/tris/

Algorithme du tri par insertion :

On note n la **taille du tableau** à trier.

⋆ On va d'abord écrire un algorithme d'insertion d'un élément dans un tableau déjà trié :

Insertion

Le tableau **tab**, a pour taille **i** et est trié dans l'ordre croissant. On veut insérer l'élément **e** dans ce tableau.

$k \leftarrow i$
Tant que $k \geq 1$ et $tab[k] > e$
    $tab[k+1] \leftarrow tab[k]$ *On décale les éléments plus grands que e vers la droite.*
    $k \leftarrow k-1$
$tab[k+1] \leftarrow e$ *On place l'élément e dans l'emplacement vide*

⋆ Puis on écrit l'algorithme du tri par insertion, en parcourant le tableau de gauche à droite :

Tri par Insertion

Pour i allant de 2 à n faire
    $e \leftarrow tab[i]$ *On mémorise la valeur à placer dans la variable e*
    $k \leftarrow i-1$
    Tant que $k \geq 1$ et $tab[k] > e$
        $tab[k+1] \leftarrow tab[k]$
        $k \leftarrow k-1$
    $tab[k+1] \leftarrow e$

Remarque : On commence à examiner le deuxième élément du tableau.

- **Terminaison** de l'algorithme : La boucle **pour** a un nombre de passage déterminé et fini.

Pour la boucle **tant que**, *le variant de boucle est k*. Les valeurs prises par k constituent une suite d'entiers strictement décroissante. Donc en un nombre fini d'étapes, k sera inférieur à 1 et la boucle se terminera.

- **Correction** de l'algorithme :

Pour chaque valeur i, à la fin de la boucle **pour**, l'invariant de boucle est *"les éléments du tableau de* 1 *à* i *sont triés"*.

Avant d'entrer dans la boucle **pour**, le tableau réduit à un élément est trié.

Pour une valeur i quelconque, supposons que les éléments du tableau de 1 à $i-1$ est sont triés, avant d'entrer dans la boucle.

Pendant le tour de la boucle **tant que**, le i ième élément sera inséré correctement parmi les éléments de 1 à $i-1$ ou restera à sa place, ainsi les éléments du tableau de 1 à i seront triés.

Ainsi, lorsque $i = n$ dans la boucle **pour**, tous les éléments du tableau sont triés et l'algorithme est correct.

- **Complexité** de l'algorithme :

Les différentes affectations de cet algorithme se font en temps constant, on les fait quelle que soit la taille n du tableau à trier.

Pour étudier la **complexité du tri par insertion**, on ne va regarder que les comparaisons faites dans la condition de la boucle tant que.

Dans la boucle **pour**, i prend les valeurs de 2 à n.

Au pire des cas, l'élément e à insérer est plus petit que tous ceux qui le précèdent, et k varie de $i-1$ à 1.

Donc pour chaque valeur de i, on fait $2 \times (i-1)$ comparaisons.

Ainsi, au total on fait $S = 2 \times 1 + 2 \times 2 + 2 \times 3 + ... + 2 \times (n-1)$ comparaisons.

Une formule Mathématiques permet de montrer que $S = 2 \times \dfrac{n(n-1)}{2} = n^2 + n$.

La complexité du tri par insertion au pire des cas est **d'ordre** n^2 ; on dit qu'elle est **quadratique**.

```
def tri_insertion(tab):
    for i in range(1,len(tab)): #on décale les indices car le premier indice est 0
        e = tab[i]
        k = i-1
        while k >= 0 and tab[k] > e:  #là aussi !
            tab[k+1] = tab[k]
            k = k-1
        tab[k+1] = e
    return tab
```

Visualiser l'exécution du programme sur le site **http ://pythontutor.com/visualize.html#mode=edit**

Remarque : Dans le pire des cas, si le tableau est trié dans l'ordre décroissant, la complexité du tri par insertion est la même que celle du tri par sélection, elle est quadratique.

Tableau de taille n	Nombre de comparaisons n^2
10	100
1 000	1 000 000
10 000	100 000 000

Mais le tri par insertion a un **meilleur comportement** que le tri par sélection lorsque le tableau est presque trié.

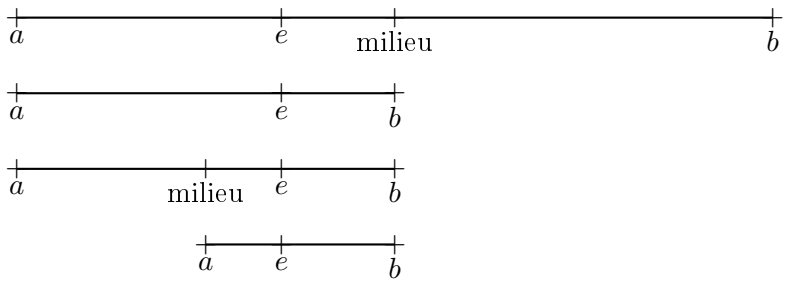
Maintenant ces deux algorithmes restent peu efficaces dès que les tableaux à trier contiennent plusieurs milliers d'éléments ; il existe de meilleurs algorithmes de tri, plus complexes, dont celui offert par Python avec les fonctions **sort** et **sorted**. On peut **observer le temps d'exécution** de ces algorithmes à l'aide du module **time** de Python.

6) Algorithme de Dichotomie

Le fait qu'un tableau soit trié, par exemple par ordre croissant, facilite de nombreuses opérations.

Par exemple, lors de la recherche d'un élément dans un tableau trié, on peut utiliser la **recherche Dichotomique**.

Principe : À chaque étape, on coupe le tableau en deux et on effectue un test pour savoir dans quelle partie se trouve l'élément recherché. On peut ainsi restreindre la zone de recherche par deux à chaque fois. C'est le principe **diviser pour régner**.

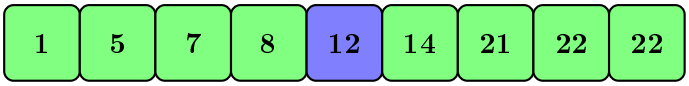


On répète ce procédé, jusqu'à obtenir la valeur recherchée, ou un intervalle devenu vide (l'élément n'est pas dans le tableau).

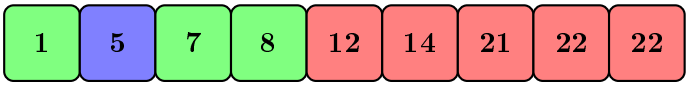
Un premier exemple :

Recherchons l'élément **5** dans le tableau trié **[1,5,7,8,12,14,21,22,22]**.

Tour 1 : La liste est non vide, l'élément central vaut 12, et $12 > 5$.



Tour 2 : La liste est non vide, l'élément central vaut 5, on a trouvé la valeur recherchée.



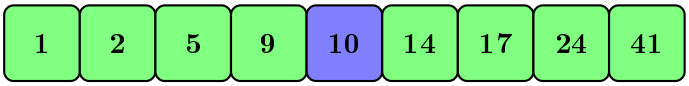
On a fait deux comparaisons pour trouver le nombre 5 dans ce tableau.

- -

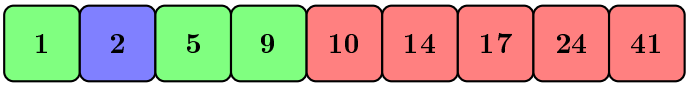
Un seconde exemple :

Recherchons l'élément **7** dans le tableau trié $[1, 2, 5, 7, 10, 14, 17, 24, 41]$.

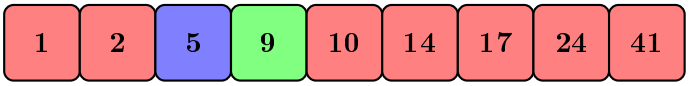
Tour 1 : La liste est non vide, l'élément central vaut 10, et $10 > 7$.



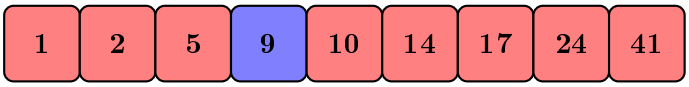
Tour 2 : La liste n'est pas vide, l'élément central vaut 2, et $7 > 2$.



Tour 3 : La liste n'est pas vide, l'élément central vaut 5 et $5 < 7$.



Tour 4 : Il ne reste plus qu'un seul élément dans la liste examinée, c'est 9 et comme $9 \neq 7$, la recherche se termine, et l'élément n'a pas été trouvé.



Il a donc fallu 4 comparaisons pour montrer qu'un élément n'est pas dans ce tableau.

Recherche Dichotomique

On note **tab** un tableau et **n** sa taille. On cherche si l'élément **e** est dans le tableau. On renvoie un booléen pour indiquer si l'élément est présent ou non :

$a \leftarrow 1$ #Indice du premier élément du tableau
$b \leftarrow n$ #Indice du dernier élément du tableau
tant que $b - a \geq 0$ #tant que le tableau n'est pas vide
  milieu $\leftarrow$ partie entière de $\left(\frac{a+b}{2}\right)$ *# Il faut un nombre entier car c'est un indice*
  si tab[milieu] = e #On a trouvé la valeur dans le tableau
    alors on retourne Vrai
  si tab[milieu] > e
    alors $b \leftarrow milieu - 1$ *# la valeur est trop grande, au mieux c'est celle d'avant*
    sinon $a \leftarrow milieu + 1$ *# la valeur est trop petite, au mieux c'est celle d'après*
on retourne Faux

- **Terminaison** de l'algorithme :

Pour la boucle **tant que** :

Premier cas, on trouve la valeur e et l'algorithme s'arrête ;

Second cas, on ne trouve pas e dans le tableau : alors *le variant de boucle est* $b - a$ car les valeurs prises par $b - a$ constituent une suite d'entiers strictement décroissante. En effet, soit a augmente de 1, soit b diminue de 1 et donc l'écart entre les deux ne fait que diminuer strictement. Donc en un nombre fini d'étapes, $b - a$ sera inférieur strictement à 0 et la boucle se terminera.

On a donc démontré la terminaison de cet algorithme.

- **Correction** de l'algorithme :

Plusieurs points sont à vérifier :

1) L'algorithme ne va pas accéder au tableau en dehors de ses bornes :

   En effet, au départ a est égal à l'indice du premier élément du tableau, et b à celui du dernier élément.

   La variable milieu, de par sa définition est un entier tel que $a \leq milieu \leq b$. Donc on a bien $1 \leq milieu \leq n$.

2) On a deux premiers invariants de boucle : $a \geq 1$ et $b \leq n$.

   Cela est vérifié avant l'entrée dans la boucle Tant que ;

   Ensuite a peut augmenter de 1, donc reste supérieur ou égal à 1 et b peut diminuer de 1, dons reste inférieur ou égal à n.

   Ces invariants sont bien vérifiés.

3) Il reste à montrer que si l'**on retourne "Vrai"**, c'est que l'élément e est dans le tableau :

   Or on retourne vrai si $tab[milieu] == e$, et $tab[milieu]$ est un élément du tableau.

   Donc on a bien trouvé e dans notre tableau.

4) Enfin, il faut montrer que si l'**on retourne "Faux"**, c'est que l'élément e n'est pas dans le tableau :

   pour cela on utilise l'invariant de boucle suivant : *"l'élément e ne peut se trouver qu'entre les éléments d'indices a et b du tableau, compris"*

   Avant d'entrer dans la boucle, ceci est vrai car $a = 1$ et $b = n$, on a tout le tableau.

   Ensuite, il y a deux cas :

   - soit $tab[milieu] > e$: on a tous les éléments de $milieu$ à b qui sont supérieurs ou égaux à celui d'indice $milieu$ car le tableau est trié dans l'ordre croissant, donc e ne peut-être qu'entre les termes d'indice a et $milieu - 1$, et b prenant la valeur $milieu - 1$, l'invariant est vérifié.

   - soit $tab[milieu] < e$: on a tous les éléments de a à $milieu$ qui sont inférieurs ou égaux à celui d'indice $milieu$ car le tableau est trié dans l'ordre croissant, donc e ne peut-être qu'entre les termes d'indice $milieu + 1$ et b, et a prenant la valeur $milieu + 1$, l'invariant est vérifié.

Tout ceci prouve la correction de notre algorithme.

- **Complexité** de l'algorithme :

On va s'intéresser au nombre de valeurs du tableau que l'on va examiner. Cela correspond au nombre d'itérations de la boucle Tant que.

On se place dans le pire des cas, lorsque l'élément e n'est pas dans le tableau.

Alors on répète la boucle tant que l'intervalle du tableau que nous gardons n'est pas vide. ($b - a \geq 0$)

Par exemple, pour un tableau de taille 100 :

On examine la valeur tab[milieu], puis on garde un tableau de taille 49 ou 50, selon le côté choisi.

On regarde à nouveau la valeur tab[milieu], puis on garde un tableau de taille au plus égale à 25.

Puis on recommence en gardant un tableau de taille au plus égale à 12, puis à 6, puis à 3, puis à 1.

Il aura fallu 7 tours de boucles. La complexité est d'ordre 7 pour un tableau de taille 100, au pire des cas.

De manière générale, pour un tableau de taille n, on cherche **le plus petit entier k tel que $2^k > n$**. (On cherche le nombre de divisions par 2 que l'on peut faire sur n.)

k est alors l'ordre de grandeur de la complexité de l'algorithme dans le pire des cas.

Tableau comparatif :

Taille du tableau n	nombre maximal d'itérations k
10	4
100	7
1 000	10
1 000 000	20
1 000 000 000	30

L'algorithme de recherche Dichotomique est **extrêmement efficace**, notamment par rapport à un algorithme de recherche séquentiel.

On dit que le coût de l'algorithme de recherche Dichotomique est d'ordre $log_2(n)$.

```
def recherche_dichotomique(tab,e):
    a = 0
    b = len(tab) - 1
    while b-a >= 0:
        milieu = (a+b)//2  #obtenir un entier
        if tab[milieu] == e:
            return True
        elif tab[milieu] > e:
            b = milieu - 1
        else:
            a = milieu + 1
    return False
```

7) Exercice

Écrire un algorithme de recherche du maximum dans une liste de nombres entiers. Puis l'implémenter en Python.