# AGR. Algorithmes. Tri

## I/ Historique

Les algorithmes de tri sont fondamentaux dans la gestion des données et permettent l'accès à des informations dans des délais très brefs. Ainsi, l'ingéniosité des informaticiens (informaticiennes en particulier d'ailleurs) a été mise en œuvre pour élaborer ceux qui nécessitent le moins d'opérations.

Durant la Seconde Guerre mondiale, la *Moore School of Engineering* à Philadelphie embauche des femmes pour calculer les trajectoires balistiques. Parmi elles, Betty Holberton est remarquée pour ses compétences et fait partie de l'équipe de conception de l'ENIAC, l'un des tous premiers ordinateurs créé en 1946. C'est dans ce cadre qu'elle développe le premier algorithme de tri. Toute sa carrière est consacrée au développement des langages Fortran et Cobol ce qui lui vaut en 1997 le prix Ada Lovelace.

Le mathématicien américain Donald Knuth né en 1938 est l'un des pionniers de l'algorithmique et l'auteur d'une série de livres « *Art of programming* » qui sont toujours des références actuellement. Sa célébrité lui vient également de la création d'un traitement de texte scientifique, du nom de TeX.

Pour contrer l'avance de l'URSS dans la conquête de l'espace (le premier satellite du monde, Spoutnik lancé avec succès en 1957 pendant que les tentatives américaines échouaient les unes après les autres), les Etats-Unis créent la **NASA** l'année suivante. Les femmes (afro-américaines notamment) y joueront un rôle crucial -dans une période encore très ancrée dans le racisme- rarement évoquées dans les médias de l'époque.

En savoir plus :
Betty Holberton : https://wikimonde.com/article/Betty_Holberton
Ada Lovelace : https://www.futura-sciences.com/sciences/personnalites/mathematiques-augusta-ada-lovelace-8
Donald Knuth : https://www.babelio.com/auteur/Donald-Knuth/208907
Les femmes afro-américaines de la guerre froide : https://www.20minutes.fr/monde/1927475-20160920-nasa-mathematiciennes-noires-geniales-coeur-conquete-spatiale-americaine
Spoutnik de l'ex-URSS : https://www.dailymotion.com/video/x63atf2

## II/ Tri par sélection

### 1/ Principe

On dispose de *n* données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi celles restantes et on la place en deuxième position et ainsi de suite.

Exemple :

Un exemple de programme de tri par sélection en Python :

```
from random import randint

# Compte le nombre de comparaisons
counter=0

## Tri en ordre croissant
def triTableauCroissant(Tab) :
    global counter

    for i in range(0,len(Tab) - 1) :  # Le dernier élément sera forcément le plus grand
        min_value = Tab[i]
        min_index = i
        for j in range(i+1,len(Tab)) : # Les i premiers éléments sont ordonnés
            counter += 1

            if Tab[j] < Tab[min_index] :
                min_value = Tab[j]
                min_index = j

        Tab[i],Tab[min_index] = Tab[min_index],Tab[i]

    return Tab

# Génération d'un tableau de valeurs aléatoires (ici 30 valeurs)
T = [randint(0,100) for i in range(30)]
print("Tableau initial :",T,"\n")

# Tableau trié
T = triTableauCroissant(T)
print("Tableau final :", T,"en ", counter,  "opérations")
```

```
Tableau initial : [40, 63, 71, 22, 2, 23, 26, 28, 36, 32, 90, 11, 45, 58, 96, 72, 87, 43, 21, 65, 62, 76, 49, 38, 16, 44, 28, 8
8, 41, 61]

Tableau final : [2, 11, 16, 21, 22, 23, 26, 28, 28, 32, 36, 38, 40, 41, 43, 44, 45, 49, 58, 61, 62, 63, 65, 71, 72, 76, 87, 88,
90, 96] en  435 opérations
```

La variable *counter* compte le nombre de permutations effectuées.

## 2/ Terminaison et coût de l'algorithme

**Rappel** : un algorithme est une suite d'instructions ordonnée et finie aboutissant à un résultat.

Terminaison : L'algorithme se termine puisqu'il est composé d'une boucle double *for* finie.

Coût : Pour un tableau de n valeurs, il y a deux boucles *for* imbriquées.

- Lors du premier tour, on a i = 0 donc la seconde boucle varie entre *1* et *n – 1* : n – 1 tours.
- Lors du second tour, on a i = 1, donc la seconde boucle varie entre *2* et *n - 1* : n – 2 tours.
- etc.
- Lors du dernier tour, on a i = n – 2, donc la seconde varie entre *n – 1* et *n – 1* : 1 tour.

On en conclut qu'il y a 1 + 2 + 3 + …….. + n – 3 + n – 2 + n – 1 tours au total.

Soit :
S = 1 + 2 + 3 + …. + n – 3 + n – 2 + n – 1. Il y a *n – 1* termes.
On peut aussi écrire que :
S = n – 1 + n – 2 + n – 3 + … + 3 + 2 + 1. Il y a *n – 1* termes.

En additionnant les deux égalités :
2S = (1 + n – 1) + (2 + n – 2) + (3 + n – 3) + … + (n – 3 + 3) + (n – 2 + 2) + (n – 1 + 1). Il y a *n – 1* couples.
2S = n + n + n + … + n + n + n. Il y a *n – 1* couples.

Donc 2S = n x (n – 1) soit **S = n(n – 1)/2**

On a donc, en développant, $S = n^2/2 – n/2$. Le nombre de données n étant très grand, on peut donc assimiler S à $n^2/2$ soit un **coût quadratique**.

**A savoir : Lors de l'exécution d'un algorithme de tri par sélection, le coût est toujours de n(n – 1)/2 permutations.**
**A savoir : L'algorithme tri par sélection a un coût quadratique quelque soit la configuration de la liste à trier.**

Cet algorithme est donc peu performant, réservé à des listes contenant peu d'éléments. Même si la liste est déjà triée, le coût sera quadratique !

**Rappels sur les coûts :**
Pour une liste contenant *n* éléments :

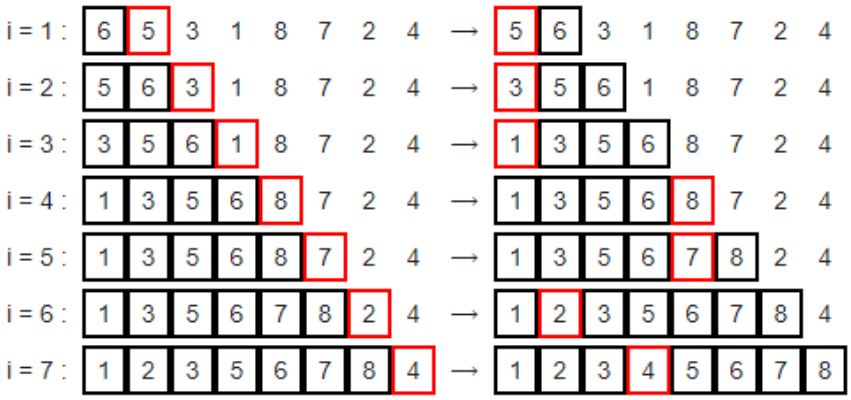
- **Un algorithme ayant un coût constant ne dépend pas du nombre d'éléments d'une liste. On peut citer l'accès au premier élément d'une liste.**
- **Un algorithme ayant un coût linéaire (en fonction du nombre *n* d'éléments) parcourt la liste une seule fois. On peut citer le calcul d'une moyenne, la recherche d'un extremum etc.**
- **Un algorithme ayant un coût quadratique (en fonction du nombre *n* d'éléments) est composé -pour simplifier_ d'une boucle double sur les éléments de la liste (d'autres cas existent mais dépassent le cadre de NSI de première).**
- **Un algorithme ayant un coût logarithmique (en fonction du nombre *n* d'éléments) divise le nombre d'éléments à traiter par tour (souvent par deux). On peut citer la recherche dichotomique.**

# III/ Tri par insertion

## 1/ Principe

Il est très proche d'un tri d'un jeu de cartes. On place les cartes en fonction des autres en tenant compte d'éventuelles cartes déjà triées au préalable, ce que ne considère pas l'algorithme par sélection.

Exemple :



Un exemple programme en Python de tri par insertion :

```python
def tri_insertion(L) :
    n = len(L)  # Nombre d'éléments de la liste L

    if n == 0 :  # Cas de la liste vide.
        return L

    for j in range(1,n) :
        # A l'étape j, le sous-tableau L[0,j-1] est trié.
        i = j

        # On insère L[j] dans ce sous-tableau en déterminant
        # le plus petit indice `i` tel que 0 <= i <= j et L[i-1] > L[j].
        # Cela permet de ne pas retrier ce qui est déjà partiellement trié : gain de temps
        # par rapport à l'insertion par substitution.
        while i > 0 and L[i-1] > L[j] :
            i = i - 1  # L'indice `i` est décroissant

        # Si i != j, on décale le sous tableau L[i,j-1] d'un cran
        # vers la droite et on place L[j] en position i.
        # On peut remarquer que l'instruction `if i < j` fonctionne également.
        if  i != j :
            for k in range(j,i,-1) :
                L[k] = L[k-1]  # Décalage à droite, attention, k est décroissant !!

            L[i] = L[j]

    return L
```

```python
# Jeu de tests
print(tri_insertion([2,5,-1,7,0,28]))          # Attendu : [-1, 0, 2, 5, 7, 28]
print(tri_insertion([10,9,8,7,6,5,4,3,2,1,0])) # Attendu : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[-1, 0, 2, 5, 7, 28]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## 2/ Terminaison et coût de l'algorithme

Terminaison : Il  y a deux boucles *for* et une boucle *while* imbriquées. Les boucle *for* se terminent bien sûr ; la boucle *while* ayant pour condition i > 0 avec i entier naturel aussi puisque i est strictement décroissant, Il y a au plus i + 1 tours.

Coût : Dans le pire des cas (liste ordonnée dans le sens décroissant), on retrouve le cas de l'algorithme précédent, soit une **complexité quadratique**. En revanche, si la liste est déjà ordonnée, on retrouve un coût linéaire.

Cet algorithme est donc efficace sur une liste « presque ordonnée ».