CHAPITRE 8 : ALGORITHMES DE TRI

I. Le tri par sélection.

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison. Cet algorithme est simple, mais considéré comme inefficace car il s'exécute en temps quadratique en le nombre d'éléments à trier, et non en temps pseudo linéaire.

1.1 Description de l'algorithme.

Sur un tableau de n éléments (numérotés de 0 à n-1 , attention un tableau de 5 valeurs (5 cases) sera numéroté de 0 à 4 et non de 1 à 5), le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

1.2 Pseudo-code.

```
procédure tri_selection(tableau tab)
   n ← longueur(t)
   pour i de 0 à n - 2
      indice _mini ← i
      pour j de i + 1 à n - 1
         si tab[j] < tab[indice _mini], alors indice _mini ← j
      fin pour
         si indice _mini ≠ i, alors échanger tab[i] et tab[indice _mini]
   fin pour
fin procédure
```

Une variante consiste à procéder de façon symétrique, en plaçant d'abord le plus grand élément à la fin, puis le second plus grand élément en avant-dernière position, etc.

Le tri par sélection peut aussi être utilisé sur des listes. Le principe est identique, mais au lieu de déplacer les éléments par échanges, on réalise des suppressions et insertions dans la liste.

1.3 Complexité.

Dans tous les cas, pour trier n éléments, le tri par sélection effectue $\frac{n\times(n-1)}{2}$ comparaisons. Sa complexité est donc $\Theta(n^2)$. De ce point de vue, il est inefficace puisque les meilleurs algorithmes s'exécutent en temps $O(n \log n)$. Il est même moins bon que le tri par insertion ou le tri à bulles, qui sont aussi quadratiques dans le pire cas mais peuvent être plus rapides sur certaines entrées particulières.

1.4 Code Python.

```
# Tri par sélection

def tri_selection(tab):
   # Trouver l'indice et la valeur du minimum
```

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

# Programme principale pour tester le code ci-dessus
tab = [9, 6, 1, 4, 8]
print("Le tableau initial est :", tab)
print("Le tableau trié est :", tri_selection(tab))
```

1.5 Pas à pas. T = [9, 6, 1, 4, 8].

n	i	ind_mini	j	tab[ind_mini] > tab[i]		
T = [9, 6, 1, 4, 8]						
5						
	0					
		0				
			1			
				tab[0] > tab[1]	9 > 6	True
		1				
			2			
				tab[1] > tab[2]	6 > 1	True
		2				
			3			
				tab[2] > tab[3]	1 > 4	False
		2				
			4			
				tab[2] > tab[4]	1 > 8	False
T = [1, 6, 9, 4, 8] permutation de 9 avec 1						
	1					
		1				
			2			
				tab[1] > tab[2]	6 > 9	False
			3			
				tab[1] > tab[3]	6 > 4	True
		3				
			4			
				tab[3] > tab[4]	4 > 8	False
T = [1, 4, 9, 6, 8] permutation de 6 avec 4						
	2					
		2				
			3			
				tab[2] > tab[3]	9 > 6	True
		3				
			4			
				tab[3] > tab[4]	6 > 8	False

T = [1, 4, 6, 9, 8] permutation de 9 avec 6						
	3					
		3				
			4			
				tab[2] > tab[3]	9 > 8	True
		4				
T = [1, 4, 6, 8, 9] permutation de 9 avec 8						

II. Le tri par insertion.

Le **tri par insertion** est un algorithme de tri classique, que la plupart des personnes utilisent naturellement pour trier des cartes : prendre une à une les cartes mélangées sur la table, et former une main en insérant chaque carte à sa place.
En général, le tri par insertion est beaucoup plus lent que d'autres algorithmes comme le tri rapide et le tri fusion pour traiter de grandes séquences.
Le tri par insertion est cependant considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées.

2.1 Description de l'algorithme.

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le *i*-ème élément, les éléments qui le précèdent sont déjà triés. Pour faire l'analogie avec l'exemple du jeu de cartes, lorsqu'on est à la *i*-ème étape du parcours, le *i*-ème élément est la carte saisie, les éléments précédents sont la main triée et les éléments suivants correspondent aux cartes encore mélangées sur la table.
L'objectif d'une étape est d'insérer le *i*-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire "remonter" l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

2.2 Pseudo-code.

Les éléments du tableau *T* sont numérotés de 1 à *n*.

```
procédure tri_insertion(tableau T, entier n)
   pour i de 2 à n (du 2ème élément au dernier élément)
      j = i
      tant que j > 0 et T[j - 1] > T[j]
         échanger T[j] et T[j - 1]
         j = j - 1
      fin tant que
  fin pour
fin procédure
```

2.3 Complexité.

a) Approche.

Si N est la taille de l'entrée, le contenu de la première boucle est exécuté (l'ordinateur fait ce qui est dedans) environ N fois (plus précisément, N-1 fois).
À chaque fois qu'on exécute cette première boucle, on exécute la deuxième boucle entre 0 et i fois : on part de la case i, et on va jusqu'à la case 0, en s'arrêtant avant si on trouve une carte plus petite que l'élément à insérer. On a donc un nombre de comparaisons variant entre 0 et i, donc entre 0 et N.
Le nombre total d'exécutions de la deuxième est donc plus grand que N * 0, et plus petit que N * N, entre 0 et N^2.
On dit donc que le nombre de calculs de cet algorithme est de l'ordre de N^2 (dans le calcul détaillé, vous verrez que c'est plutôt $N^2 / 2$, mais que la différence n'a pas grand sens).

b) Calcul exact dans le pire cas.

Une bonne façon de mesurer la complexité de l'algorithme est de compter le nombre de calculs effectués "dans le pire cas", c'est à dire dans la configuration qui donnera le plus de travail possible à l'algorithme. Si on connaît bien ce cas, on a une idée du temps maximal que peut prendre l'algorithme (et donc on sait que "c'est toujours mieux que ...", ce qui est assez rassurant).
Dans le pire cas, il faut décaler à chaque fois toutes les cartes de la main gauche, et pas seulement une partie d'entre elles. À chaque tour de boucle on a donc **i** comparaisons (la taille de la main gauche).
Le nombre total de comparaisons est donc 1 + 2 + 3 + 4 + 5...+ *N*-1 (la dernière valeur de i est N-1).

Combien vaut ce nombre ? Appelons-le S. On a :
$S = 1 + 2 + 3 + 4 + 5 + \ldots + N\text{-}5 + N\text{-}4 + N\text{-}3 + N\text{-}2 + N\text{-}1$

Soit $S = \frac{N(N-1)}{2}$ (somme des n termes d'une suite arithmétique de premier terme $U_0 = 1$ et de raison $r = 1$)

Quand *N* est très grand, *N*(*N*-1) est approximativement égal à N^2, et le nombre de comparaisons de l'algorithme est donc d'environ $N^2/2$, ou $N^2 \times 0.5$ comparaisons.
Cependant, le facteur 0.5 n'a pas grand sens : sur un ordinateur deux fois plus rapide, on ira deux fois plus vite et sur un ordinateur 2 fois plus lent, deux fois moins vite. Pour estimer le temps mis par cet algorithme de manière indépendante de l'ordinateur, on dit donc que le nombre d'actions qu'il fait est "de l'ordre de N^2".

c) Conclusion.

En langage "scientifique", on dira que la complexité du tri par insertion est de $\Theta(n^2)$.
En pratique, cela signifie que si l'on double la taille du tableau, l'algorithme sera 4 fois plus lent, et si on la multiplie par 10, 100 fois plus lent.

1.4 Code Python.

```
# Tri par insertion

#tableau = [5, 1, 4, 2, 8]
#tableau = [17, 3, 0, 17, 16, 2, 20, 8, 6, 12, 8, 2, 17, 8, 13, 0, 15, 19, 12, 5]
#tableau = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
tableau = [9, 6, 1, 4, 8]

print("tab = ", tableau)
def tri_insertion(tab):
    nb_iter = 0
    for i in range(1, len(tab)):
        j = i
        while j > 0 and tab[j - 1] > tab[j]:
            tab[j - 1], tab[j] = tab[j], tab[j - 1]
            print("tab = ", tab)
            j -= 1
```

```
            nb_iter += 1
    print("Nombre d'étapes = ", nb_iter, "\ntab = ", tab)

tri_insertion(tableau)
```

```
>>>
tab =  [9, 6, 1, 4, 8]
tab =  [6, 9, 1, 4, 8]
tab =  [6, 1, 9, 4, 8]
tab =  [1, 6, 9, 4, 8]
tab =  [1, 6, 4, 9, 8]
tab =  [1, 4, 6, 9, 8]
tab =  [1, 4, 6, 8, 9]
Nombre d'étapes =  6
tab =  [1, 4, 6, 8, 9]
>>>
```

```
>>>
tab =  [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
tab =  [19, 20, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
…
tab =  [2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
tab =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Nombre d'étapes =  190
tab =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>>
```

1.5 Pas à pas. T = [9, 6, 1, 4, 8].

n	i	j	j > 0 et T[j - 1] > T[j]	T[j−1]	T[j]
T=[9, 6, 1, 4, 8]					
5					
	1				
		1			
			T[0] > T[1] | 9 > 6		
				T[0] = 9	T[1] = 6
		0		T[0] = 6	T[1] = 9
T=[6, 9, 1, 4, 8]					
	2				
		2			
			T[1] > T[2] | 9 > 1		
				T[1] = 9	T[2] = 1
				T[1] = 1	T[2] = 9
T=[6, 1, 9, 4, 8]					
		1			
			T[0] > T[1] | 6 > 1		
				T[0] = 6	T[1] = 1
				T[0] = 1	T[1] = 6
T=[1, 6, 9, 4, 8]					
		0			
	3				

		3			
			T[2] > T[3] | 9 > 4		
				T[2] = 9	T[3] = 4
				T[2] = 4	T[3] = 9
T=[1, 6, 4, 9, 8]					
		2			
			T[1] > T[2] | 6 > 4		
				T[1] = 6	T[2] = 4
				T[1] = 4	T[2] = 6
T=[1, 4, 6, 9, 8]					
		1			
			T[0] > T[1] | 1 > 4 (Faux)		
		0			
	4				
		4			
			T[3] > T[4] | 9 > 8		
				T[3] = 9	T[4] = 8
				T[3] = 8	T[4] = 9
T=[1, 4, 6, 8, 9]					
		3			
			T[2] > T[3] | 6 > 8 (Faux)		
		2			
			T[1] > T[2] | 4 > 6 (Faux)		
		1			
			T[0] > T[1] | 1 > 4 (Faux)		
		0			
T=[1, 4, 6, 8, 9]					

III. Le tri à bulles. (Hors programme)

Le tri à bulles ou tri par propagation est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.

3.1. Algorithme de base.

L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

3.2 Pseudo-code.

Voici la description en pseudo-code du tri à bulles, pour trier un tableau T de *n* éléments numérotés de 0 à n−1 :

```
procédure tri_bulles(tableau T, entier n)
   répéter
      permut = Faux
      pour i de 0 à n−2
         si T[i] > T[i + 1], alors
            échanger T[i] et T[i + 1]
            permut = Vrai
   tant que permut=Vrai
```

3.3 Complexité.

Pour un tableau de taille *n*, le nombre d'itérations de la boucle externe "répéter … tant que permut" est compris entre 1 et *n*. En effet, on peut démontrer qu'après la $i^{ème}$ étape, les *i* derniers éléments du tableau sont à leur place. À chaque itération, il y a exactement *n*−1 comparaisons et au plus *n*−1 échanges.

Le pire cas (*n* itérations) est atteint lorsque le plus petit élément est à la fin du tableau. Le meilleur cas (une seule itération) est atteint quand le tableau est déjà trié. Dans ce cas, la complexité est linéaire.

3.4 Exemple étape par étape.

Prenons la liste de chiffres "5 1 4 2 8" et trions-la de manière croissante en utilisant l'algorithme de tri à bulles.

Pour chaque passage, les éléments comparés sont écrits en gras.
Premier passage :

(**5** **1** 4 2 8) ▹ (**1** **5** 4 2 8) Les éléments 5 et 1 sont comparés, et comme 5 > 1, l'algorithme les intervertit.
(1 **5** **4** 2 8) ▹ (1 **4** **5** 2 8) Interversion car 5 > 4.
(1 4 **5** **2** 8) ▹ (1 4 **2** **5** 8) Interversion car 5 > 2.
(1 4 2 **5** **8**) ▹ (1 4 2 **5** **8**) Comme 5 < 8, les éléments ne sont pas échangés.

Deuxième passage :

(**1 4** 2 5 8) ▷ (**1 4** 2 5 8) Même principe qu'au premier passage.
(1 **4 2** 5 8) ▷ (1 **2 4** 5 8)
(1 2 **4 5** 8) ▷ (1 2 **4 5** 8)
(1 2 4 **5 8**) ▷ (1 2 4 **5 8**)

À ce stade, la liste est triée, mais pour le détecter, l'algorithme doit effectuer un dernier passage.

Troisième et dernier passage :

(**1 2** 4 5 8) ▷ (**1 2** 4 5 8)
(1 **2 4** 5 8) ▷ (1 **2 4** 5 8)
(1 2 **4 5** 8) ▷ (1 2 **4 5** 8)
(1 2 4 **5 8**) ▷ (1 2 4 **5 8**)

Comme la liste est triée, aucune interversion n'a lieu à ce dernier passage, ce qui provoque l'arrêt de l'algorithme.

3.5 Code Python.

Attention la boucle Do … While() (répéter … tant que) n'existe pas sur Python.

```
# Tri à bulles avec un seul passage

tableau = [17, 3, 0, 17, 16, 2, 20, 8, 6, 12, 8, 2, 17, 8, 13, 0, 15, 19, 12, 5]

def tri_bulles(tab):
    for i in range(len(tab) - 1):
        if tab[i] > tab[i + 1]:
            tab[i], tab[i + 1] = tab[i + 1], tab[i]
        print("i = ", i," tab = ", tab)

tri_bulles(tableau)
```

```
# Tri à bulles avec plusieurs passages

tableau=[5, 1, 4, 2, 8]
#tableau=[17, 3, 0, 17, 16, 2, 20, 8, 6, 12, 8, 2, 17, 8, 13, 0, 15, 19, 12, 5]
#tableau=[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]


def tri_bulles(tab):
    permut, nb_iter = True, 0
    while permut:
        permut = False
        for i in range(len(tab) - 1):
            nb_iter += 1
            if tab[i] > tab[i + 1]:
                tab[i], tab[i + 1] = tab[i + 1], tab[i]
                permut = True
```

```
            print("i = ", i," tab = ", tab)
    print("Nombre d'étapes = ", nb_iter, "\ntab = ", tab)

tri_bulles(tableau)
```

Résultats :

```
>>>
i =  0  tab =  [1, 5, 4, 2, 8]
i =  1  tab =  [1, 4, 5, 2, 8]
i =  2  tab =  [1, 4, 2, 5, 8]
i =  3  tab =  [1, 4, 2, 5, 8]
i =  0  tab =  [1, 4, 2, 5, 8]
i =  1  tab =  [1, 2, 4, 5, 8]
i =  2  tab =  [1, 2, 4, 5, 8]
i =  3  tab =  [1, 2, 4, 5, 8]
i =  0  tab =  [1, 2, 4, 5, 8]
i =  1  tab =  [1, 2, 4, 5, 8]
i =  2  tab =  [1, 2, 4, 5, 8]
i =  3  tab =  [1, 2, 4, 5, 8]
Nombre d'étapes =  12
tab =  [1, 2, 4, 5, 8]
>>>
```

```
>>>
i =  0  tab=[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Nombre d'étapes =  380
tab = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>>
```

3.6 Pas à pas.

```
T = [5, 1, 4, 2, 8]
```

```
procédure tri_bulles(tableau T, entier n)
    répéter
        permut = Faux
        pour i de 0 à n–2
            si T[i] > T[i + 1], alors
                échanger T[i] et T[i + 1]
                permut = Vrai
    tant que permut=Vrai
```

n	permut	i	T[i]	T[i+1]
T = [5, 1, 4, 2, 8]				
5				
	Faux			
		0		
			5	
				1

			1	5
	Vrai			
T = [1, 5, 4, 2, 8]				
	Faux			
		1		
			5	
				4
			4	5
	Vrai			
T = [1, 4, 5, 2, 8]				
	Faux			
		2		
			5	
				2
	Vrai			
T = [1, 4, 2, 5, 8]				
		…		
T = [1, 2, 4, 5, 8]				
	Faux			
T = [1, 2, 4, 5, 8]				
	Faux			