Algorithmes de tri sur les tableaux

I : Tri par insertion

On consultera www.visualgo.net pour voir l'algorithme en œuvre.

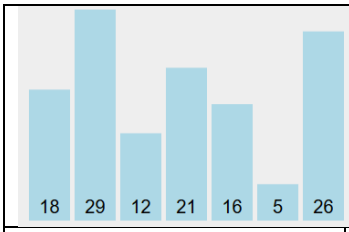


Tableau initial

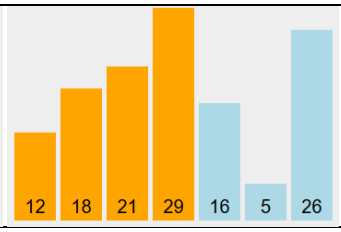
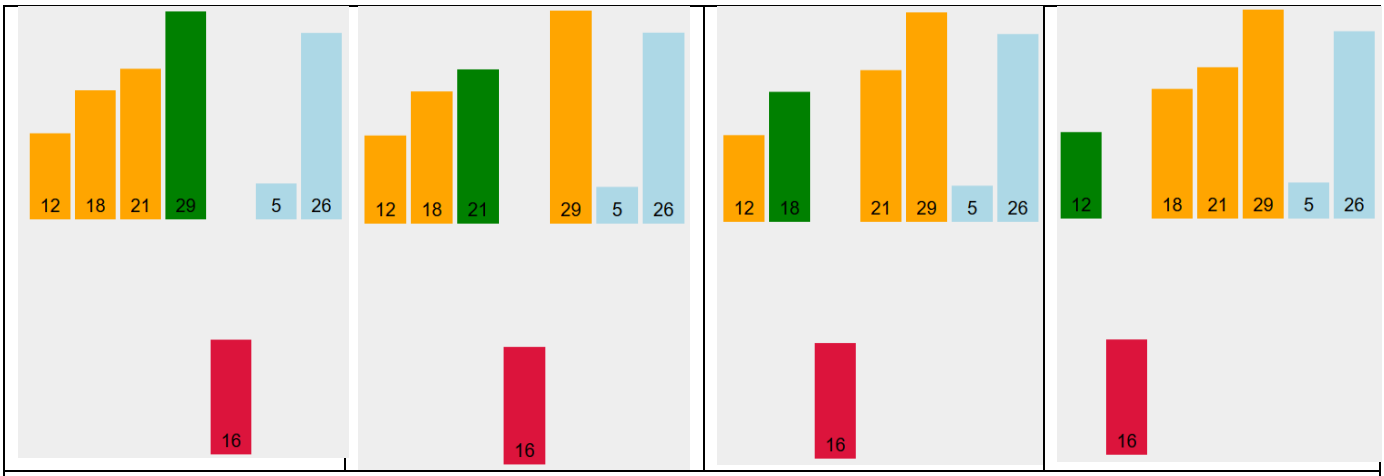


Tableau avant le 4e passage dans la boucle principale



4e passage dans la boucle : insertion de l'élément d'indice 4

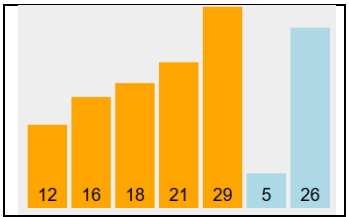


Tableau à la fin du 4e passage dans la boucle principale

I.1 : Caractérisation

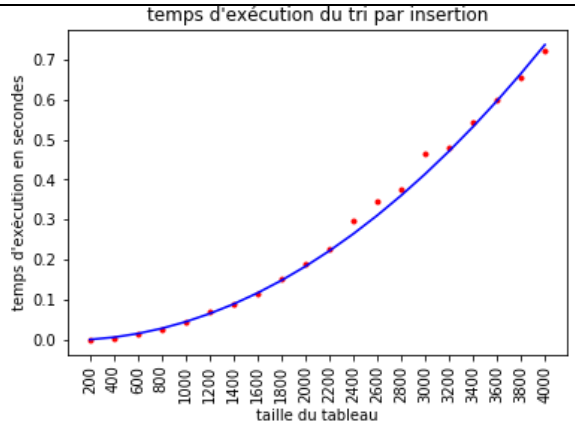
- Avant le 4ème passage dans la boucle principale, les éléments du tableau d'indices 0 à 3 sont les éléments d'indices 0 à 3 du tableau initial triés par ordre croissant,

- Le 4ème passage dans la boucle principale insère *au bon endroit* l'élément d'indice 4.

```
index = 1
while index < len(tableau):

    valeur_a_inserer = tableau[index]

    j = index
    while j > 0 and valeur_a_inserer < tableau[j-1] :
        tableau[j] = tableau[j-1]
        j = j - 1

    tableau[j] = valeur_a_inserer

    index = index + 1
```

I.2 : Étude expérimentale de la complexité sur des tableaux de nombres aléatoires

Taille n du tableau	Temps d'exécution (ms)
200	2
400	5
1 000	46
2 000	188
4 000	722



Le temps d'exécution sur machine de cet algorithme semble proportionnel au carré de la longueur n du tableau. On dit que la complexité dans le pire des cas est quadratique soit $O(n^2)$. Il faut alors se méfier des tableaux trop grands car le coût augmente beaucoup plus vite que la taille du tableau.

I.3 Démonstration de la complexité quadratique dans le pire des cas pour le tri par insertion

```
index = 1
while index < len(tableau):                                          \
                                                                     |
    valeur_a_inserer = tableau[index]                 # O(1)         \      |
                                                      #              |      |
    j = index                                         # O(1)         |      |
                                                      #              |      |
    while j > 0 and valeur_a_inserer < tableau[j-1] :  #      \       |      } O(n*n)
        tableau[j] = tableau[j-1]                     # O(1)  }O(index) }O(index)|
        j = j - 1                                     # O(1) /       |      |
                                                      #              |      |
    tableau[j] = valeur_a_inserer                     #              |      |
                                                      #              |      |
    index = index + 1                                 # O(1)         /      /

Ici la difficulté est de comprendre comment on passe de O(index) à O(n*n).
En fait index varie de 1 à n-1, et pour chacune de ces valeurs on a une complexité en O(index).
On a donc une complexité en O(1)+O(2)+O(3)+O(4)+ ... + O(n-2)+O(n-1).
Soit une complexité en O(1+2+3+4+...+(n-2)+(n-1)). Or en mathématiques on montre que cette somme vaut
O(n*(n-1)/2)=O(n*n/2 - n/2) qui équivaut à une complexité en O(n*n)
```

I.4 : Correction de l'algorithme

```
index = 1
while index < len(T):
    (♦)
    valeur_a_inserer = T[index]

    j = index
    while j > 0 and valeur_a_inserer < T[j-1] :
        T[j] = T[j-1]
        j = j-1

    T[j] = valeur_a_inserer

    index = index + 1
    (♦♦)
```

Nous appellerons boucle extérieure la boucle `"while index < len(tableau):"`
Nous appellerons boucle intérieure la boucle `"while j > 0 and valeur_a_inserer < tableau[j-1] :"`

L'algorithme se termine

- pour la boucle extérieure : `index` augmente de 1 à chaque passage dans la boucle alors que la taille de `tableau` ne change pas. Donc l'inégalité `index < taille(tableau)` finira par être fausse et on sortira bien de la boucle.

- pour la boucle intérieure : `j` décroît de 1 à chaque passage dans la boucle donc l'inégalité `j > 0` finira par être fausse et on sort bien de la boucle.

On dispose d'un invariant de boucle

Un invariant de boucle est une propriété qui, si elle est vraie au début du corps de la boucle (♦) est vraie à la fin du corps de la boucle (♦♦).
Or pour cet algorithme la propriété suivante est un invariant de boucle :
"Les éléments d'indice `0, 1, 2, 3 … ,index-1` du tableau sont triés par ordre croissant"

Conclusion

- Si le tableau fourni est de taille 0 ou 1, on ne rentre pas dans la boucle extérieure mais le tableau est trié dès le départ donc l'algorithme est correct.

- Sinon , la taille du tableau est supérieure ou égale à 2 :
  - on rentre bien dans la boucle extérieure
  - on commence avec `index` = 1 donc l'invariant "L'élément d'indice 0 du tableau est trié par ordre croissant" est vrai avant de rentrer dans la boucle extérieure.
  - on est certain que la boucle extérieure se termine donc on est certain que l'invariant sera vrai en sortie de boucle (lorsque `index = taille(tableau)`)
  - Donc en fin d'algorithme "Les éléments d'indice `0, 1, 2, 3 … ,taille(tableau)-1` du tableau sont triés par ordre croissant". CQFD.

II : Tri par sélection

On consultera www.visualgo.net pour voir l'algorithme en œuvre.

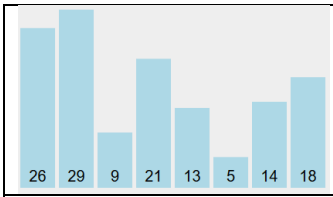


Tableau initial

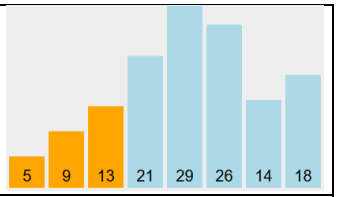
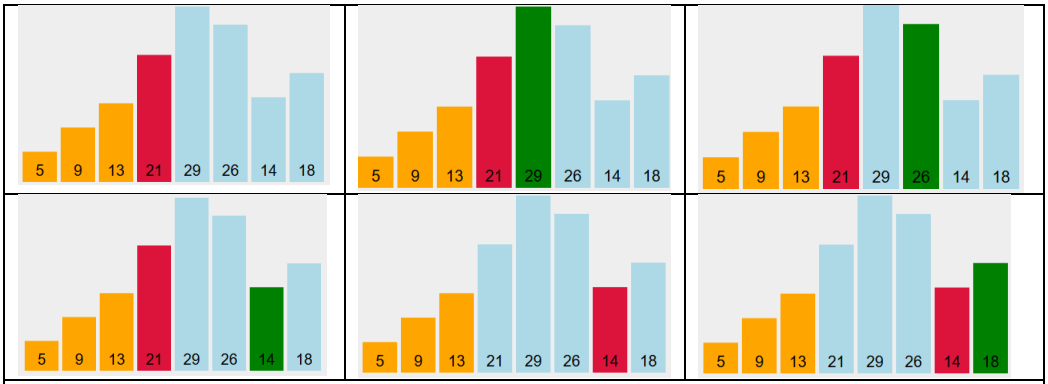


Tableau avant le 4e passage dans la boucle principale



4e passage dans la boucle : sélection du plus petit élément de la partie non triée du tableau

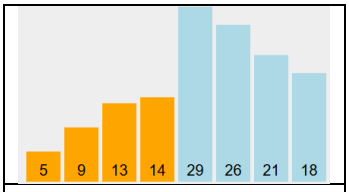


Tableau à la fin du 4e passage dans la boucle principale

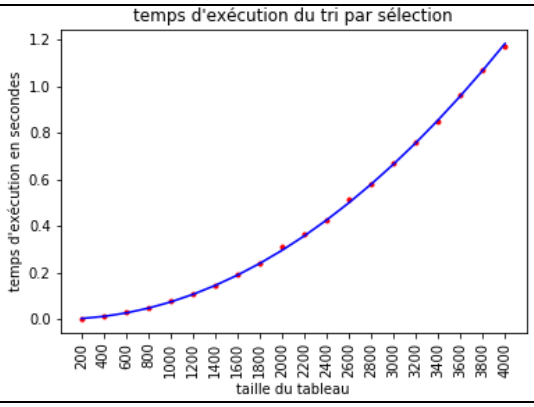
II.1 : Caractérisation

- Avant le 4ème passage dans la boucle principale, les éléments d'indices 0 à 2 du tableau sont les 3 plus petits éléments du tableau initial triées par ordre croissant,
- Le 4ème passage dans la boucle principale sélectionne le 4ème plus petit élément du tableau puis le positionne à l'indice 3

```
index = 0
while index < len(tableau):

    j_min = index
    j = j_min + 1

    while j < len(tableau) :
        if tableau[j] < tableau[j_min] :
            j_min = j
        j = j + 1

    tableau[index], tableau[j_min] = tableau[j_min], tableau[index]
    index = index + 1
```

II.2 : Étude expérimentale du coût sur des tableaux de nombres aléatoires

Taille n du tableau	Temps d'exécution (ms)
200	3
400	11
1 000	79
2 000	310
4 000	1 170



Le temps d'exécution sur machine de cet algorithme semble proportionnel au carré de la longueur n du tableau. On dit que la complexité dans le pire des cas est quadratique soit $O(n^2)$. Il faut alors se méfier des tableaux trop grands car le coût augmente beaucoup plus vite que la taille du tableau.

II.3 : Démonstration de la complexité quadratique dans le pire des cas pour le tri par sélection

```
index = 0
while index < len(tab):                                                    \
                                                                           |
    j_min = index                                   # O(1)            \    |
                                                    #                 |    |
    j = j_min + 1                                   # O(1)            |    |
                                                    #                 |    |
    while j < len(tab) :                            #      \          |    | O(n*n)
        if tab[j] < tab[j_min]:                     # O(1)  |          |    |
            j_min = j                               # O(1)  }O(val)    }O(val)  |
        j = j + 1                                   # O(1) /           |    |
                                                    #                 |    |
    tab[index], tab[j_min] = tab[j_min], tab[index] #0(1)             |    |
                                                    #                 |    |
    index = index + 1                               # O(1)            /    /

Avec val = len(tab) – index = n - index

Ici la difficulté est de comprendre comment on passe de O(val) à O(n*n).
En fait index varie de 0 à n-1, et pour chacune de ces valeurs la complexité est en O(val)=O(n-index).
On a donc une complexité en O(n)+O(n-1)+O(n-2)+O(n-3)+ ... + O(2)+O(1).
Soit une complexité en O(n+(n-1)+(n-2)+...+2+1). Or en mathématiques on montre que cette somme vaut
O(n*(n+1)/2)=O(n*n/2 + n/2) qui équivaut à une complexité en O(n*n).
```

II.4 : Correction de l'algorithme

Nous appellerons boucle extérieure la boucle `"while index < len(tab) :"`
Nous appellerons boucle intérieure la boucle `"while j < len(tab) :"`

```
index = 0
while index < len(tab):
    (♦)
    j_min = index
    j = j_min + 1

    while j < len(tab) :
        if tab[j] < tab[j_min] :
            j_min = j
        j = j + 1

    tab[index], tab[j_min] = tab[j_min], tab[index]
    index = index + 1
    (♦♦)
```

L'algorithme se termine

- pour la boucle extérieure : `index` augmente de 1 à chaque passage dans la boucle alors que la taille de `tab` ne change pas. Donc l'inégalité `index < taille(tab)` finira par être fausse et on sortira bien de la boucle.

- pour la boucle intérieure : `j` augmente de 1 à chaque passage dans la boucle donc l'inégalité `j < taille(tab)` finira

par être fausse et on sort bien de la boucle.

On dispose d'un invariant de boucle

Un invariant de boucle est une propriété qui, si elle est vraie au début du corps de la boucle (♦) est vraie à la fin du corps de la boucle (♦♦).
Or la propriété suivante est un invariant de boucle :
"Les `index` plus petits éléments du tableau sont positionnés aux indices `0, 1, 2 … index-1` et sont triés par ordre croissant"

Conclusion

- Si le tableau fourni est de taille 0, on ne rentre pas dans la boucle extérieure mais le tableau est trié dès le départ donc l'algorithme est correct.

- Sinon , la taille du tableau est supérieure ou égale à 1 :
  - on rentre bien dans la boucle extérieure
  - on commence avec `index` = 0 donc l'invariant "Les `0` plus petits éléments du tableau sont …" est vrai avant de rentrer dans la boucle extérieure.
  - on est certain que la boucle extérieure se termine donc on est certain que l'invariant sera vrai en sortie de boucle (lorsque `index = taille(tableau)`)
  - Donc en fin d'algorithme "Les `taille(tab)` plus petits éléments du tableau sont positionnés aux indices `0, 1, 2 … taille(tab)-1` et sont triés par ordre croissant"- CQFD