

## Introduction

Si on dispose d'un tableau d'éléments comparables, on a vu qu'une recherche dichotomique est efficace : l'ordre de grandeur en nombre de comparaisons est le nombre de chiffres en base 2 de la taille du tableau, on parle de coût logarithmique. Néanmoins une condition nécessaire est que le tableau soit trié.

Dans ce cours on présente quelques algorithmes de tri et on examine certaines de leurs propriétés. Les fonctions de tri sont testées sur des tableaux d'entiers mais peuvent être appliquées à des tableaux d'éléments comparables avec les opérateurs  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$  et  $!=$ . De même on effectue des tris croissants mais il suffit d'inverser les opérateurs de comparaison pour obtenir des tris décroissants.

### Sources :

- « Manuel de première NSI » de Balabonski, Filiâtre, N'Guyen chez Ellipses.
- « Introduction to programming in Python » de Sedgewick, Wayne, Dondero chez Addison-Wesley.

Tous les codes sont à compléter dans ce notebook Capytale :

<https://capytale2.ac-paris.fr/web/c-auth/list?returnto=/web/code/967c-255137>

## 1 Quelques outils

### Exercice 1 Déterminer si un tableau est trié dans l'ordre croissant

Compléter le code de la fonction `tri_croissant(tab)` la spécification est donnée ci-dessous. Un jeu de tests unitaires est fourni.

```
def tri_croissant(tab):
    """
    Détermine si le tableau d'entiers tab
    est trié dans l'ordre croissant et t[0:i] trié et t[0:i] <= t[i+1:n] croissant

    Parameters:
        tab : tableau d'entiers
        Précondition : len(tab) > 0

    Returns: booléen
    """
    assert len(tab) > 0
    .....
    .....
    .....
    .....
    .....
```

.....

## Exercice 2 *Fonction mystère*

Renommer et spécifier la fonction `mystere(tab)` ci-dessous :

```
import random

def mystere(tab):
    n = len(tab)
    for i in range(n):
        r = random.randint(i, n)
        tmp = tab[r]
        tab[r] = tab[i]
        tab[i] = tmp
```

## Exercice 3 *Comparer deux tableaux*

Compléter la fonction `compare_tab(tab1, tab2)` qui prend en paramètres deux tableaux d'entiers avec la précondition qu'ils soient triés dans l'ordre croissant et de même taille, et détermine s'ils comportent les mêmes éléments. Un jeu de tests unitaires est fourni.

```
def compare_tab(tab1, tab2):
    """
    Détermine si les tableaux d'entiers tab1
    et tab2 triés et de même taille
    comportent les memes éléments

    Parameters: tab1, tab 2: tableaux d'entiers

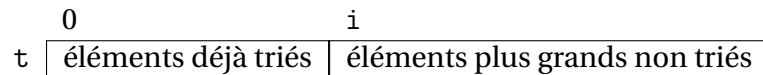
    Returns: booléen
    """
    .....
```

## 2 Algorithme de tri par sélection

### Point de cours 1 *Spécification du tri par sélection*

Soit une fonction `tri_selection(t)` qui applique le **tri par sélection** (ordre croissant) à un tableau `t`.

- **Paramètre d'entrée :** Un tableau `t` contenant  $n$  éléments comparables de même type.



- **Description :** Le **tri par sélection** dans l'ordre croissant parcourt le tableau de gauche à droite. Le tableau est séparé en deux parties : les éléments déjà triés et à leur place définitive sont à gauche et la partie droite contient tous les autres éléments plus grands et non triés. À chaque itération on recherche le plus petit élément de la partie non triée pour l'échanger avec le premier élément non trié (d'index  $i$ ), ainsi on étend par la droite la partie triée d'un élément.

Pour visualiser une trace d'exécution : <https://professeurb.github.io/articles/tris/>

```

1 def tri_selection(t):
2     n = len(t)
3     #boucle externe
4     for i in range(0, n):
5         #invariant : t[0:i] trié et t[0:i] <= t[i:n]
6         #boucle interne de recherche de l'index du minimum dans t[i:n]
7         imin = recherche_index_min(t, i)
8         #postcondition de boucle interne : t[imin] = min(t[i:n])
9         #echange de t[i] et t[imin]
10        echange(t, i, imin)

```

- **Sortie :** Le tri s'effectue **en place** c'est-à-dire que les éléments du tableau `t` sont déplacés à l'intérieur de la structure et le tri du tableau `t` le modifie par effet de bord. Ainsi la fonction ne renvoie rien, en fait `None` par défaut en Python.

## Exercice 4 *Exécution pas à pas le tri par sélection*

1. Exécuter pas à pas `tri_selection([10, 4, 5])` en complétant le tableau d'état ci-dessous où les valeurs des variables sont mesurées après l'exécution de la ligne correspondante.

Ligne	i	imin	t
ligne 4	0	...	...
ligne 7	0	...	...
ligne 10	0	...	...
ligne 4	1	...	...
ligne 7	1	...	...
ligne 10	1	...	...
ligne 4	2	...	...
ligne 7	2	...	...
ligne 10	2	...	...

2. Exécuter de même pas à pas `tri_selection([4, 5, 10])` et `tri_selection([10, 5, 4])`.

## Exercice 5 *Implémentation du tri par sélection*

1. Finaliser l'implémentation de la fonction `tri_selection(t)` en complétant les codes des fonctions :

- `recherche_index_min(t, i)`
- `echange(t, a, b)`

.....

.....

.....

.....

.....

.....

.....

.....

.....  
.....  
.....  
.....  
.....

2. Vérifier que `tri_selection(t)` passe le jeu de tests unitaires fourni avec `test_tri(tri_selection)`.

⚡ La fonction `tri_croissant(t)` de l'exercice 1 doit être bien définie.

3. Écrire une version `tri_selection2(t)` sans les fonctions auxiliaires `recherche_index_min(t, i)` et `echange(t, a, b)`.

.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

## Exercice 6 *Terminaison et correction du tri par sélection*

1. Soit `t` un tableau d'entiers quelconque, l'évaluation de `tri_selection(t)` se termine-t-elle nécessairement? Pourquoi?

*On démontre ainsi la **terminaison** de l'algorithme de tri par sélection.*

.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

2. Considérons un tableau  $t$  de taille  $n$ .

Pour tout indice  $i$  de la boucle externe de `tri_selection(t)` on considère la propriété  $\mathcal{P}(i)$  définie avant que le tour de boucle d'indice  $i$  s'exécute, par :

$$\mathcal{P}(i) := \text{« } t[0:i] \text{ trié et } t[0:i] \leq t[i:n] \text{ »}$$

a. Justifier que  $\mathcal{P}(0)$  est vraie.

*On démontre ainsi que la propriété est vraie avant l'exécution de la boucle.*

.....  
.....

b. Soit  $i$  l'indice d'un tour de boucle entre 0 et  $n$ , justifier que si  $\mathcal{P}(i)$  est vraie alors  $\mathcal{P}(i + 1)$  est vraie.

*On démontre ainsi que la propriété est conservée pour chaque itération de boucle.*

Une propriété conservée par chaque itération de boucle s'appelle un **invariant**.

.....  
.....  
.....  
.....  
.....

c. On a démontré précédemment la **terminaison** de `tri_selection(t)` pour tout tableau d'entiers  $t$ . La propriété  $\mathcal{P}(i)$  est vraie avant le premier tour de boucle et c'est un **invariant** de boucle donc  $\mathcal{P}(n)$  est vraie lorsque la boucle se termine. Exprimer  $\mathcal{P}(n)$ , que peut-on dire du tableau  $t$  à la fin de l'exécution de la boucle externe?

*On démontre ainsi la **correction** de l'algorithme de tri par sélection.*

.....  
.....  
.....  
.....  
.....

## Exercice 7 Stabilité du tri par sélection

Compléter le tableau ci-dessous avec la trace d'exécution de l'algorithme de tri par sélection (ordre croissant) appliqué au tableau : 

843	843	842
-----	-----	-----

.

On distinguera bien les deux occurrences de 843.

	t		
i	0	1	2
	843	843	842
0	...	...	...
1	...	...	...

Un algorithme de tri qui conserve l'ordre initial des éléments égaux est dit **stable**.

L'algorithme de tri par sélection est-il stable?

.....

.....

.....

## Méthode Mesure du temps d'exécution

Voici une fonction permettant de mesurer le temps d'exécution d'une fonction de tri en place de tableau sur un échantillon de nb\_essais tableaux de taille taille\_tab.

```
import time, random

def temps_echantillon(tri_en_place, taille_tab, nb_essais):
    total = 0
    for _ in range(nb_essais):
        tab_alea = [random.randint(-1000, 1000) for _ in range(taille_tab)]
        debut = time.perf_counter()
        tri_en_place(tab_alea)
        total = total + (time.perf_counter() - debut)
    return total
```

## Exercice 8 Complexité du tri par sélection

1. Compléter la spécification (dans la *docstring*) de la fonction `test_doubling_ratio(tri_en_place, nb_essais)` fournie dans le notebook en décrivant ce qu'elle fait.

.....

.....

.....

2. Vérifier qu'en évaluant `test_doubling_ratio(tri_selection, 10)`, vous obtenez des résultats similaires à :

```
>>> test_doubling_ratio(tri_selection, 10)
128 2.79999999999994314
256 3.448275862069025
```

```
512 5.029126213592477
1024 4.584980237154263
2048 4.226879861711322
```



Selon la version, cette fonction contient une boucle infinie qu'il convient d'interrompre avec le bouton STOP de l'IDE ou la combinaison de touches CTRL + C.

Que peut-on observer?

.....  
.....  
.....

3. Considérons un tableau de taille  $n$ , le nombre de comparaisons effectué par le tri par sélection dépend-il des données du tableau (déjà triées, toutes égales, triées dans l'ordre inverse, aléatoires ...)?

.....  
.....  
.....  
.....  
.....

4. En supposant que le temps d'exécution dépend essentiellement du nombre de comparaisons effectuées. Quelle conjecture peut-on faire sur l'ordre de grandeur du nombre de comparaisons effectuées par le tri par sélection en fonction de la taille  $n$  du tableau d'entrée?

.....  
.....  
.....  
.....

5. On choisit comme modèle le décompte des comparaisons. Démontrer la conjecture précédente.  
*L'ordre de grandeur du coût d'exécution d'un algorithme en nombre d'opérations élémentaires par rapport à la taille  $n$  de son entrée mesure sa **complexité temporelle**.*

.....  
.....  
.....  
.....  
.....



6. Est-il raisonnable de trier un tableau de  $10^6$  nombres avec le tri par sélection?

Vérifier en modifiant la fonction pour afficher le temps de l'échantillon courant puis en appelant `test_doubling_ratio(tri_selection, 1)`.

.....

.....

.....

.....

## 3 Algorithme de tri par insertion

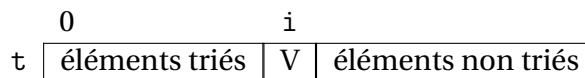


### Point de cours 2 *Tri par insertion*

Le **tri par insertion** est aussi appelé *tri des joueurs de cartes*.

Soit une fonction `tri_insertion(t)` qui applique le **tri par insertion** (ordre croissant) à un tableau `t`.

- **Paramètre d'entrée :** Un tableau `t` contenant  $n$  éléments comparables de même type.



- **Description :** Le **tri par insertion** dans l'ordre croissant parcourt le tableau de gauche à droite en maintenant à gauche une partie triée et à droite une partie non triée mais à chaque itération, au lieu de rechercher le plus petit élément de la partie non triée, il prend son premier élément d'index `i` et l'insère à sa place dans la partie triée en décalant vers la droite les éléments plus grands. Ainsi la partie triée gagne un élément à chaque itération mais les éléments triés ne sont pas à leur place définitive comme dans le tri par sélection.

Pour visualiser une trace d'exécution : <https://professeurb.github.io/articles/tris/>

```
def tri_insertion(t):
    n = len(t)
    #boucle externe
    for i in range(1, n):
        #invariant de boucle : t[0:i] trié
        #boucle interne d'insertion de t[i] à sa place dans t[0:i]
        insertion(t, i)
```

- **Terminaison :** L'algorithme se termine car il est constitué d'une boucle externe bornée qui fait varier l'index  $i$  de la partie non triée  $t[i : n]$  entre 1 et  $n - 1$  et d'une boucle interne `while` non bornée mais qui insère  $t[i]$  dans  $t[0 : i]$  en au plus  $i$  échanges de place entre un élément et son voisin de gauche.
- **Invariant et correction :** Au début de l'itération  $i$  de la boucle externe, la partie  $t[0 : i]$  est triée dans l'ordre croissant et l'insertion de  $t[i]$  dans  $t[0 : i]$  par la boucle interne maintient cet invariant, ce qui prouve la correction.
- **Sortie :** Le tri s'effectue **en place** comme pour le tri par sélection.
- **Stabilité :** Le **tri par insertion** est stable car il n'y a pas de permutations de valeurs dans la partie non triée.
- **Complexité temporelle :** Pour un tableau  $t$  de taille  $n$ , l'ordre de grandeur du nombre de comparaisons qu'effectue le **tri par insertion** est .....

## Exercice 9 Implémentation du tri par insertion

1. Finaliser dans le notebook, l'implémentation de la fonction `tri_insertion(t)` en complétant le code de la fonction `insertion(t, i)`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2. Vérifier que `tri_insertion(t)` passe le jeu de tests unitaires avec `test_tri(tri_insertion)`.

La fonction `tri_croissant(t)` de l'exercice 1 doit être bien définie.

3. Écrire une version `tri_insertion2(t)` sans la fonction auxiliaire `insertion(t, i)`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

4. Pour aller plus loin : Écrire une nouvelle version de la fonction `insertion(t, i)` où on effectue au plus  $i$  affectations lors de l'insertion (une permutation de `t[j]` et `t[j-1]` compte deux affectations).

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

### Exercice 10 *Complexité du tri par insertion*

1. Existe-t-il des tableaux de taille  $n$  pour lequel le tri par insertion effectue au plus  $n$  comparaisons (*meilleur des cas*) ?

.....

.....

.....

2. Existe-t-il des tableaux de taille  $n$  pour lesquels le nombre maximal de comparaisons est toujours atteint (*pire des cas*) ?

.....  
.....  
.....

3. Vérifier qu'en évaluant `test_doubling_ratio(tri_insertion, 10)`, vous obtenez des résultats similaires à :

```
>>> test_doubling_ratio(tri_insertion, 10)
128 2.9362770819578694
256 4.042328118210682
512 4.483588294003115
1024 4.38177808348038
2048 4.063818466385776
```



Selon la version, cette fonction peut contenir une boucle infinie qu'il convient d'interrompre avec le bouton STOP de l'IDE ou la combinaison de touches CTRL + C.

Que peut-on observer ?

.....  
.....  
.....

4. Si on applique le tri par insertion à des tableaux aléatoires, quelle conjecture peut-on faire sur l'ordre de grandeur moyen de sa complexité temporelle en fonction de la taille  $n$  du tableau d'entrée ?

.....  
.....  
.....  
.....

5. Pour conclure, quel est l'avantage du tri par insertion par rapport au tri par sélection ?

.....  
.....  
.....  
.....

## 4 Implémentations des tris par sélection et insertion dans d'autres langages

### Exercice 11

Parmi les implémentations ci-dessous en langages C ou Javascript, identifiez les algorithmes de tri par sélection ou par insertion.

Langage C

```
void mystery_sort1(int *a, int n) {
    for(size_t i = 1; i < n; ++i) {
        int tmp = a[i];
        size_t j = i;
        while(j > 0 && tmp < a[j - 1]) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = tmp;
    }
}
```

Langage Javascript

```
function mystery_sort2(nums) {
    var len = nums.length;
    for(var i = 0; i < len; i++) {
        var minAt = i;
        for(var j = i + 1; j < len; j++) {
            if(nums[j] < nums[minAt])
                minAt = j;
        }

        if(minAt != i) {
            var temp = nums[i];
            nums[i] = nums[minAt];
            nums[minAt] = temp;
        }
    }
    return nums;
}
```

## Langage Javascript

```
function mystery_sort3(a) {
  for (var i = 0; i < a.length; i++) {
    var k = a[i];
    for (var j = i; j > 0 && k < a[j - 1]; j--)
      a[j] = a[j - 1];
    a[j] = k;
  }
  return a;
}
```

## Langage C


```
void mystery_sort4(int *a, int n) {
  int i, j, m, t;
  for (i = 0; i < n; i++) {
    for (j = i, m = i; j < n; j++) {
      if (a[j] < a[m]) {
        m = j;
      }
    }
    t = a[i];
    a[i] = a[m];
    a[m] = t;
  }
}
```

## 5 D'autres algorithmes de tris hors programme

### Exercice 12 *Tri par bulles*

1. Le **tri par bulles** consiste à balayer successivement de gauche à droite le tableau à trier en faisant remonter vers la droite le plus grand élément non trié par permutations d'éléments adjacents.

On donne une première implémentation du tri par bulles sous la forme d'une fonction qui trie en place le tableau passé en paramètre.

 On utilise la fonction `echange(t, a, b)` définie dans l'exercice 5.

```
def tri_bulle1(tab):
  n = len(tab)
  for i in range(n):
    for j in range(n - 1):
      if tab[j] > tab[j + 1]:
        echange(tab, j, j + 1)
```

Exécuter `tri_bulle1` sur un tableau `tab` de valeurs `[4, 3, 2]` en complétant le tableau d'état ci-dessous avec les valeurs de `tab` après chaque exécution de la boucle interne pour des valeurs de `i`

et j fixées :

i	0	0	1	1	2	2
j	0	1	0	1	0	1
tab	[3, 4, 2]	[3, 2, 4]	...	...	...	...

2. Dans le tableau d'état précédent, on observe certains tours de boucles internes inutiles. Combien de comparaisons sont effectuées par `tri_bulle1(tab)` si `tab` est de taille  $n$ ?

.....  
 .....  
 .....

Comment modifier `tri_bulle1(tab)` pour que seules  $n-1+n-2+\dots+1 = \frac{n(n-1)}{2}$  comparaisons soient effectuées pour `tab` de taille  $n$  et donc 3 comparaisons si `tab` de valeur `[4, 3, 2]`?

.....  
 .....  
 .....

3. Dans l'exemple d'exécution ci-dessous, on voit que l'algorithme peut s'arrêter lorsque aucune « bulle » n'est remontée lors du dernier balayage.

3	5	2	1	8	9	7	3	tableau de départ
3	2	1	5	8	7	3	9	5 et 9 remontent
2	1	3	5	7	3	8	9	3 et 8 remontent
1	2	3	5	3	7	8	9	2 et 7 remontent
1	2	3	3	5	7	8	9	5 remonte
1	2	3	3	5	7	8	9	aucun élément ne remonte, liste triée

Compléter une troisième implémentation de la procédure de tri par bulles pour que le tri s'arrête si aucune permutation n'a été effectuée lors du dernier parcours du tableau.

```
def tri_bulle3(tab):
    n, i, permutation = len(tab), 0, True
    while permutation:
        permutation = .....
        for j in range(.....):
```

```

        if tab[j] > tab[j + 1]:
            echange(tab, ....., .....)
            permutation = .....
    i = i + 1

```

## Exercice 13 *Tri par dénombrement*

1. Compléter la fonction `tri_comptage(tab, binf, bsup)` fournie dans `tp10.py` avec sa spécification et un jeu de tests unitaires qu'il faut exécuter avec `test_tri_comptage(tri_comptage)`.

```

def tri_comptage(tab, binf, bsup):
    """
    Tri en place par comptage le tableau
    tab tel que pour tout 0 <= k < len(tab)
    on a binf <= tab[k] <= bsup
    Parameters
    -----
    tab : tableau d'entiers
    binf, bsup : int et int
    Returns
    -----
    None.
    """
    histo = [0 for _ in range(bsup - binf + 1)]
    # on remplit l'histogramme
    .....
    .....
    .....
    .....
    .....
    .....
    # on reconstitue le tableau
    .....
    .....
    .....
    .....
    .....

```

2. Mesurer l'évolution du temps d'exécution avec la fonction `tri_comptage_mille` qui fixe les bornes de `tri_comptage` à `-1000` et `1000`. Quelle conjecture peut-on faire sur l'ordre de grandeur de la complexité temporelle du tri par comptage? Donner un argument qui permet de la justifier.

```

def tri_comptage_mille(tab):
    return tri_comptage(tab, -1000, 1000)

```



---

.....

.....

.....

.....

## 6 Fonctions de tri de la bibliothèque Python



### Point de cours 3

Il existe deux fonctions pour trier un tableau `tab` en Python.

- ☞ `sort` réalise un *tri en place*, c'est une méthode des objets de type `list` qui s'applique avec la syntaxe `tab.sort()` : elle modifie le tableau `tab` et renvoie `None`.
- ☞ `sorted` réalise un *tri externe*, elle renvoie un nouveau tableau trié et s'applique avec la syntaxe `sorted(tab)`.

Le paramètre optionnel `reverse = True` permet pour `sort` et `sorted` un tri dans l'ordre décroissant, par défaut l'ordre croissant est appliqué.

```
>>> tab = [3, 1, 4, 1, 5]
>>> tab.sort()
>>> tab
[1, 1, 3, 4, 5]
>>> t1 = [3, 1, 4, 1, 5]
>>> t2 = t1[:]
>>> t1.sort()
>>> t1
[1, 1, 3, 4, 5]
>>> t2
[3, 1, 4, 1, 5]
>>> t3 = sorted(t2)
>>> t3
[1, 1, 3, 4, 5]
>>> t2
[3, 1, 4, 1, 5]
>>> sorted(t2, reverse = True)
[5, 4, 3, 1, 1]
```