15

Algorithmique III - Algorithmes de tri

Extrait du programme

Thème : Algorithmique

- **Contenus :**
  Tris par insertion,
  par sélection

Capacités attendus :
Écrire un algorithme de tri.
Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.

Commentaires :
La terminaison de ces algorithmes est à justifier.
On montre que leur coût est quadratique dans le pire cas.

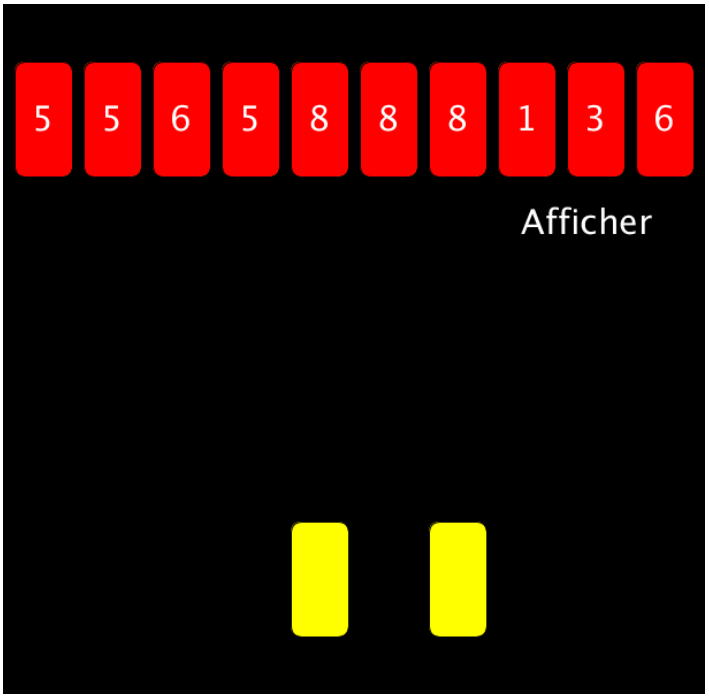
Introduction

Nous l'avons déjà vu lorsque un tableau est trié, certaines demandes sont plus rapides à exécuter. Par exemple la recherche d'un maximum ou d'un minimum. Nous verrons aussi dans un prochain chapitre, l'algorithme de la dichotomie qui est un algorithme de recherche très efficace ... à condition que le tableau soit trié.

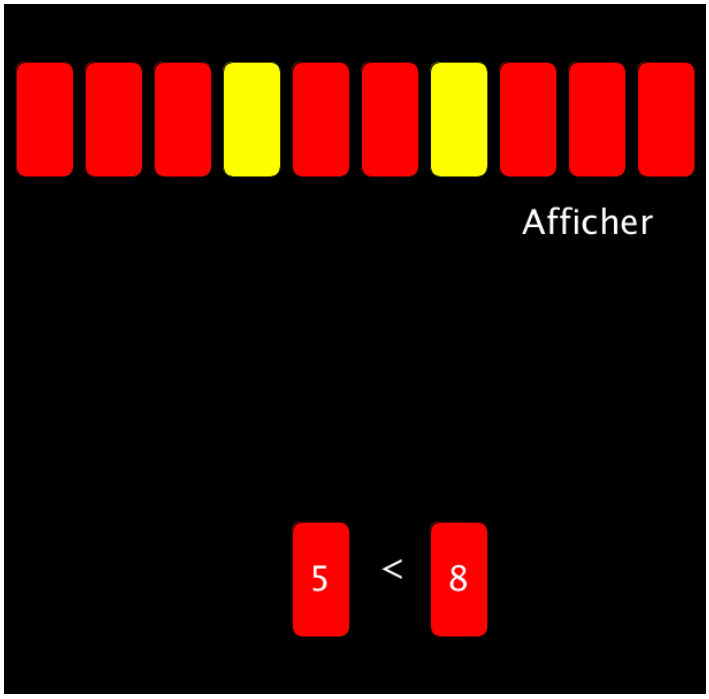
Trier n'est plus un problème en soit puisque nous l'avons déjà vu, il existe en Python (et dans tous les langages), des fonctions ou des méthodes de tri très efficaces [1].

```
>>> from random import randint
>>> L=[randint(0,100) for i in range(10)]
>>> L
[47, 18, 13, 46, 85, 1, 20, 33, 70, 57]
>>> L.sort()
>>> L
[1, 13, 18, 20, 33, 46, 47, 57, 70, 85]
```

Il existe en réalité bon nombre de méthodes pour trier un tableau et nous n'allons ici en étudier que deux. L'idée est de comprendre comment un ordinateur peut effectuer cette opération. En effet, contrairement à l'humain qui a une vision globale du tableau, l'ordinateur ne peut qu'agir étape par étape en comparant uniquement deux valeurs à la fois, les autres cases du tableau lui étant inconnues.



La vision humaine du tableau. Pour le trier, l'humain le voit en entier.



La vision d'un ordinateur du tableau. Pour le trier, il ne peut que comparer deux valeurs à la fois.

1. La recherche ne s'est par arrêté pour autant sur le sujet. On continue d'étudier de nouveaux algorithmes de tri car c'est une opération élémentaire sur les tableaux qui sont de tailles de plus en plus impressionnantes.

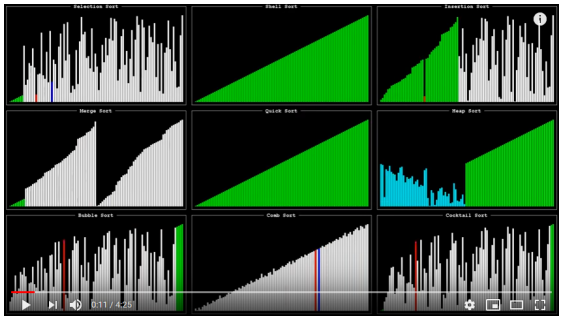
I. Spécifications

Rôle :	Trier les valeurs d'un tableau T dans l'ordre croissant.
Entrée :	Un tableau T de longueur n.
Précondition :	Les valeurs de T sont comparables.
Sortie :	Un tableau T de longueur n.
Postconditions :	Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent) ET Si $0 \leq p \leq k \leq n-1$ alors $T[p] \leq T[k]$.

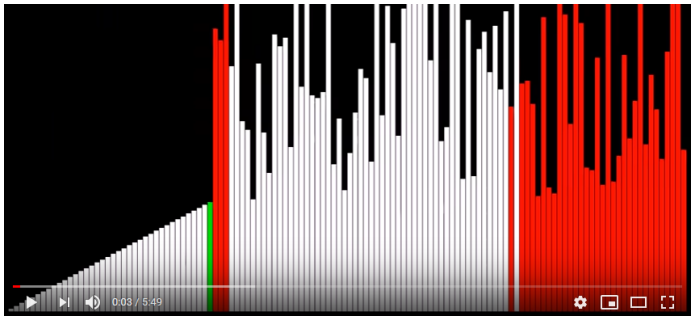
Comme déjà dit, il existe de nombreux algorithmes de tri. Nous n'en étudierons dans le cadre de ce cours que deux :

- l'algorithme de **tri par insertion** ;
- l'algorithme de **tri par sélection**.

Voici deux vidéos qui vous permettent de comparer différents algorithmes de tris. On peut constater l'efficacité de chacun.



Pour comparer 9 algorithmes en même temps, suivant quelques configurations différentes.
L'algorithme de tri par sélection est haut à gauche, celui par insertion est en haut à droite.
`https://www.youtube.com/watch?v=ZZuD6iUe3Pc`



15 algorithmes différents de tris, les uns après les autres, avec un effet musical.
Les deux premiers algorithmes sont l'algorithme par sélection, puis l'algorithme pas insertion.
`https://www.youtube.com/watch?v=kPRA0W1kECg`

II. Tri par insertion

a) Le principe

C'est un des tris les plus simples.
On parcourt le tableau de la gauche vers la droite et en maintenant une partie déjà triée sur la gauche :

déjà trié									pas encore trié			
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

On va alors prendre la valeur non encore triée la plus à gauche, $T[i]$. On va la comparer à la première valeur triée juste en-dessous, $T[i-1]$.

déjà trié									pas encore trié			
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

Deux solutions :

- Soit $T[i-1] \leq T[i]$, alors $T[i]$ est à sa bonne place et on ne fait rien :

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

- Soit $T[i-1] > T[i]$, alors, on redescend la valeur de $T[i]$ d'une case :

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

puis on recommence en comparant les valeurs $T[i-2]$ et $T[i]$:

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

...

en poursuivant ainsi, de proche en proche, on trouvera une valeur $T[j-1] \leq T[i]$ et $T[i]$ obtiendra sa place dans la partie du tableau triée.

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[i]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

b) Un exemple

Revoyons la méthode, pas à pas, avec un exemple.
Nous voulons trier le tableau T de longueur 6 suivant :

85	1	20	33	70	57

On va de la gauche, vers la droite. On commence donc par la première case. Le nombre est bien rangé (normal, il est tout seul).

déjà trié	pas encore trié				
85	1	20	33	70	57

On prend le nombre non trié le plus à gauche : 1. On le compare au nombre rangé le plus grand, autrement dit 85.

déjà trié	pas encore trié				
85	1	20	33	70	57

85>1, on va donc descendre le nombre 1.

déjà trié		pas encore trié			
1	85	20	33	70	57

Il n'y a plus rien à comparer, on a terminé pour le nombre 1.

déjà trié		pas encore trié			
1	85	20	33	70	57

On prend le nombre non trié le plus à gauche : 20. On le compare au nombre rangé le plus grand, autrement dit 85.

déjà trié		pas encore trié			
1	85	20	33	70	57

85>20, on va donc descendre le nombre 20.

déjà trié			pas encore trié		
1	20	85	33	70	57

On compare ensuite 20 avec 1. $1 \leq 20$. On a terminé pour le nombre 20.

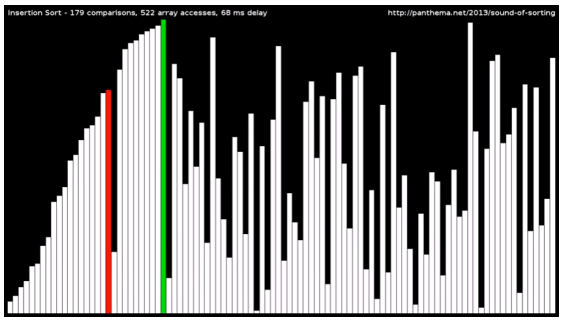
déjà trié			pas encore trié		
1	20	85	33	70	57

...

et si vous poursuiviez le tri à la main pour cette méthode ? ? ? ? ?

c) Deux vidéos

Vous pensez avoir compris ? Regarder ces deux vidéos :



Tri par insertion en musique
`https://www.youtube.com/watch?v=8oJS1BMKE64`

Un peu de danse folklorique Roumaine.
`https://www.youtube.com/watch?v=ROalU37913U`

d) L'algorithme

L'algorithme de tri par insertion :

```
1 pour i allant de 1 à n − 1 faire
2 |  j ← i ;
3 |  tant que j > 0 et T[j − 1] > T[j] faire
4 |  |  temp ← T[j − 1] ;
5 |  |  T[j − 1] ← T[j] ;
6 |  |  T[j] ← temp ;
7 |  |  j ← j − 1 ;
8 |  fin tq
9 fin pour
```

La version en langage Python :

```
1 for i in range(1,n) :
2     #on prend un nouvel indice
3     j = i
4     while (j > 0) and (T[j-1] > T[j]) :
5         #tant qu'à gauche, il y a un élé
      ment plus grand
6         #on les échange
7         T[j], T[j-1] = T[j-1], T[j]
8         j = j - 1
```

Dans l'algorithme, aux lignes 4 à 6, on échange les valeurs de $T[j]$ et $T[j-1]$. Il est nécessaire d'avoir une variable temporaire. En effet que se passerait-il si on avait écrit :

```
1 T[j] ← T[j − 1] ;
2 T[j − 1] ← T[j] ;
```

.....
.....

Pour l'écriture en Python, il n'y a pas de problème. On peut faire directement l'échange à la ligne 7 car on a utilisé des
.....

e) Terminaison

Une boucle **POUR** termine toujours.
Pour la boucle **TANT ... QUE**, il faut déterminer le **variant**.
Pour mémoire, le variant est une expression :

- impliquée dans la répétitive ;
- à valeur entière ;
- qui évolue de façon strictement décroissante au cours des itérations ;
- qui est positive ou nulle du fait des initialisations et de la condition de la répétitive.

Nous allons ici choisir ... simplement j qui décroit strictement (de 1 à chaque itération ligne 7) et est minoré par 0 (condition de la boucle ligne 3). Donc cet algorithme termine.

f) Correction partielle

Pour déterminer si l'algorithme effectue bien son travail, on doit déterminer son **invariant**. A nouveau, pour mémoire, un invariant est :

- est un prédicat sur les variables impliquées dans la répétitive,
- est vrai juste avant la répétitive,
  (induite par la précondition et les instructions en début d'algorithme)
- est vrai après chaque itération de la répétitive
  (induite par l'invariant et condition+instructions de l'itération courante)
- sera vrai après la fin de la répétitive

Pour déterminer cet invariant on va s'inspirer fortement de la postcondition et imaginer les différentes étapes d'exécution de l'algorithme.
Pour mémoire, la postcondition est :

> Les valeurs de T sont les mêmes en entrée et en sortie
> (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq n-1$ alors $T[p] \leq T[k]$.

On peut s'en inspirer pour déterminer l'invariant à la i-ème étape. Comme alors nous n'avons trié toutes les cases du tableau de $T[0]$ à $T[i]$, on a :

> Les valeurs de T sont les mêmes en entrée et en sortie
> (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

1. Initialisation de l'invariant : pour $i = 0$.
   On n'effectue pas la boucle **POUR**. On ne regarde que la première valeur du tableau et pour l'invariant :

   > Les valeurs de T sont les mêmes en entrée et en sortie
   > (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq 0$ alors $T[p] \leq T[k]$.

   donc $p = k = 0$ autrement dit $T[p] = T[k] = T[0]$.
   L'invariant est vrai.

2. Regardons le passage de la $(i-1)$-ième étape à la i-ième étape, afin de vérifier l'invariant. Supposons donc que l'invariant est vrai à la $(i-1)$-ième étape. Autrement dit, on a :

   > Les valeurs de T sont les mêmes en entrée et en sortie
   > (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq i-1$ alors $T[p] \leq T[k]$.

   Déroulons l'algorithme à la i-ème étape.
   Ligne 2, $j \leftarrow i$, donc j prend pour valeur i.
   Ligne 3, pour la boucle **TANT ... QUE**, on a deux possibilités :
   - Si $T[j-1] \leq T[j]$, la condition d'entrée dans la boucle n'est pas respectée. Donc ne fait rien.
     Comme $i = j$, on a $T[j-1] \leq T[j] = T[i]$. L'invariant devient :

     > Les valeurs de T sont les mêmes en entrée et en sortie
     > (mais dans un ordre différent)
     > ET
     > Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

     L'invariant est vrai.

- Si $T[j-1] > T[j]$, on va échanger dans la boucle **TANT ... QUE** les valeurs $T[j-1]$ et $T[j]$. A chaque échange, les valeurs avant et après $T[j]$ (jusqu'à $T[i]$) sont triées.
  Comme on échange jusqu'à ce qu'enfin $T[j-1] \leq T[j]$, quand on sort de la boucle **TANT ... QUE**, l'invariant obtenu est :

  > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
  > ET
  > Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

  L'invariant est vrai.

La correction partielle est terminée. L'algorithme trie bien les valeurs du tableau.

g) Complexité

L'algorithme de tri par insertion :

```
1 pour i allant de 1 à n − 1 faire
2 |   j ← i ;
3 |   tant que j > 0 et T[j − 1] > T[j] faire
4 |   |   temp ← T[j − 1] ;
5 |   |   T[j − 1] ← T[j] ;
6 |   |   T[j] ← temp ;
7 |   |   j ← j − 1 ;
8 |   fin tq
9 fin pour
```

On compte toutes les opérations :

Ligne 1 à 9 : on répète $n-1$ fois
- **ligne 1 :** 1 opération
  (on incrémente i de 1 jusqu'à $n-1$)
- **ligne 2 :** 1 opération
  (on affecte une valeur à j, $j \leftarrow i$)
- **Lignes 3 à 8** : on répète ? ? ? ? fois
  - **ligne 3 :** 2 opérations
    (on effectue deux tests pour savoir si on boucle)
  - **ligne 4 :** 1 opération
    (on affecte une valeur à $temp$)
  - **ligne 5 :** 1 opération
    (on affecte une valeur à $T[j-1]$)
  - **ligne 6 :** 1 opération
    (on affecte une valeur à $T[j]$)
  - **ligne 7 :** 2 opérations
    (on soustrait puis on affecte une valeur à j)

La question qui demeure est : « combien de fois répète-t-on les lignes 3 à 8 ? ». La réponse est malheureusement « ça dépend ! » mais cette réponse ne nous convient pas. Heureusement, ce que l'on recherche est plutôt la réponse à cette question :

« Combien de fois répète-t-on les lignes 3 à 8 dans le pire cas ? »

Ce qui nous amène à une autre question :

« Quel est le pire cas ? »

Réponse :

« Lorsque le tableau est rangé dans l'ordre décroissant ! »

En effet à ce moment là, la valeur non trié la plus à gauche doit être décalée, étape après étape, pour se retrouver sur la toute première case du tableau $T[0]$ [2].

Comptons alors le nombre de répétitions des lignes 3 à 8.

2. C'est vraiment dommage, si on recherche la valeur maximale du tableau et si on sait que le tableau est rangé dans l'ordre décroissant, ce n'est pas la peine de ranger le tableau dans l'ordre croissant, n'est-ce pas ?

- Si $i = 1$, on effectue une seule permutation entre $T[0]$ et $T[1]$.
  1 seule répétition.
- Si $i = 2$, on doit faire passer la valeur $T[2]$ en $T[0]$ avec deux permutations.
  2 répétitions.
- Si $i = 3$, on doit faire passer la valeur $T[3]$ en $T[0]$ avec trois permutations.
  3 répétitions.
  ...
- Pour i, on doit faire passer la valeur $T[i]$ en $T[0]$ avec i permutations.
  i répétitions.
  ...
- Si $i = (n-1)$, on doit faire passer la valeur $T[n-1]$ en $T[0]$ avec $(n-1)$ permutations.
  $(n-1)$ répétitions.

Donc nous avons au total :

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

Nos amis mathématiciens ont appris une formule[3] qui nous permet de dire que :

$$S = \frac{(n-1)n}{2}$$

Nous avons donc le nombre de répétitions dans le pire des cas. On peut maintenant compter le nombre total d'opérations :

$$Nb = \underbrace{(n-1) \times 1}_{\text{les répétitions de la ligne 1}} + \underbrace{\frac{(n-1)n}{2} \times 7}_{\text{les répétitions de toutes les boucles } \textbf{TANT ... QUE}}$$

$$Nb = n - 1 + \frac{7}{2}n^2 - \frac{7}{2}n$$

$$Nb = \frac{7}{2}n^2 - \frac{5}{2}n - 1$$

Le nombre d'opérations obtenu n'est pas une fonction affine. C'est une fonction polynôme du second degré.

Lorsque le nombre d'opérations d'un algorithme est une fonction polynôme du second degré, le nombre d'opérations est dans la famille des n^2.
On dit alors que la **complexité est quadratique** et on la note $O(n^2)$.

Autrement dit :
Si le tableau est de taille $n = 10$, comme $n^2 = 100$, on aura de l'ordre de la centaine d'opérations pour ranger le tableau.
Si le tableau est de taille $n = 100$, comme $n^2 = 10\,000$, on aura de l'ordre de dix milliers d'opérations pour ranger le tableau.
Si le tableau est de taille $n = 1\,000$, comme $n^2 = 1\,000\,000$, on aura de l'ordre du million d'opérations pour ranger le tableau.

3. $S = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$.

III. Tri par sélection

a) Le principe

C'est a aussi un des tris les plus simples.
Le tri par sélection consiste à sélectionner la valeur que l'on souhaite ranger au lieu de prendre la première venue.
On maintient toujours une partie déjà triée sur la gauche et, parmi les valeurs non encore triées du tableau, on va sélectionner celle qui est la plus petite. On placera alors cette valeur comme la plus grande des valeurs triées :

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

On regarde toutes les valeurs non encore triées et on cherche la plus petite.

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

Deux solutions :

- Soit $T[i]$ est la valeur la plus petite des valeurs non triées, alors $T[i]$ est à sa bonne place et on ne fait rien :

déjà trié						pas encore trié						
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

- Soit $T[j]$ est la valeur la plus petite :

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

On échange $T[j]$ avec la première valeur non triée $T[i]$:

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[j]$	$T[i+1]$	...	$T[j-1]$	$T[i]$	$T[j+1]$	...	$T[n-1]$

La valeur $T[j]$ est alors à sa place :

déjà trié						pas encore trié						
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[j]$	$T[i+1]$	...	$T[j-1]$	$T[i]$	$T[j+1]$	...	$T[n-1]$

On pousuit ensuite avec les valeurs non encore triées.

b) Un exemple

Revoyons la méthode, pas à pas, avec notre exemple :

85	1	20	33	70	57

On regarde tous les nombres et on sélectionne le plus petit ce sera 1 :

85	1	20	33	70	57

On le met alors en première position en l'échangeant avec le nombre 85 :

déjà trié	pas encore trié				
1	85	20	33	70	57

Parmi tous les nombres non encore triées, on sélectionne le plus petit, 20 :

déjà trié	pas encore trié				
1	85	20	33	70	57

On l'échange avec le premier nombre non trié, 85 :

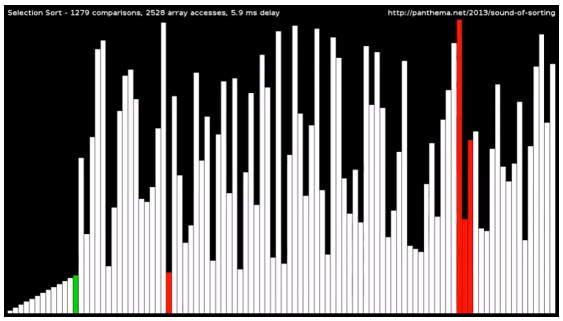
déjà trié		pas encore trié			
1	20	85	33	70	57

...

et si vous poursuiviez le tri à la main pour cette méthode ? ? ? ? ?

c) Deux vidéos

Vous pensez avoir compris ? Regarder ces deux vidéos :



Tri par sélection en musique
`https://www.youtube.com/watch?v=92BfuxHn2XE`

Un peu de danse folklorique Gipsy.
`https://www.youtube.com/watch?v=Ns4TPTC8whw`

d) L'algorithme

L'algorithme doit faire 2 choses. Chercher le minimum parmi les valeurs non triées puis l'échanger avec la première valeur non encore triée.

L'algorithme de tri par sélection :

```
1  pour i allant de 0 à n − 1 faire
2  |  indice_min ← i ;
3  |  pour j allant de i + 1 à n − 1 faire
4  |  |  si T[indice_min] > T[j] alors
5  |  |  |  indice_min ← j ;
6  |  |  fin si
7  |  fin pour
8  |  temp ← T[i] ;
9  |  T[i] ← T[indice_min] ;
10 |  T[indice_min] ← temp ;
11 fin pour
```

La version en langage Python :

```
1 for i in range(0,n) :
2     indice_min = i
3         #on cherche le minimum à partir
      de T[i]
4     for j in range(i+1,n) :
5                 if T[indice_min] > T[j] :
6                         indice_min = j
7         #on échange
8     T[indice_min], T[i] = T[i], T[
      indice_min]
```

Dans l'algorithme, aux lignes 7 à 9, on échange les valeurs de $T[indice\_min]$ et $T[i]$. Il est nécessaire d'avoir une variable temporaire.
Pour l'écriture en Python, il n'y a pas de problème. On peut faire directement l'échange à la ligne 8 car on a utilisé des

e) Terminaison

Une boucle **POUR** termine toujours. Ici, il y en a deux. Donc cet algorithme termine.

f) Correction partielle

Comme pour le tri par insertion, on garde à gauche une partie triée et à droite une partie non encore triée. On propose donc le même **invariant** tout en rajoutant une idée supplémentaire : la partie triée ne contient que des nombres plus petit que la partie non triée.

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$
> ET
> Pour tout $0 \leq k \leq i$ et tout $i+1 \leq j \leq n-1$, $T[k] \leq T[j]$.

1. Initialisation de l'invariant : pour $i = 0$.
   On recherche le minimum de toutes les valeurs et on le met à la position 0 :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq 0$ alors $T[0] = T[p] \leq T[k] = T[0]$ OK
   > ET
   > Pour tout $0 \leq k \leq 0$ et tout $1 \leq j \leq n-1$, $T[0] \leq T[j]$ OK puisque c'est le minimum

   L'invariant est vrai.

2. Regardons le passage de la $(i-1)$-ième étape à la i-ième étape, afin de vérifier l'invariant. Supposons donc que l'invariant est vrai à la $(i-1)$-ième étape. Autrement dit, on a :

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i-1$ alors $T[p] \leq T[k]$
> ET
> Pour tout $0 \leq k \leq i-1$ et tout $i \leq j \leq n-1$, $T[k] \leq T[j]$.

Déroulons l'algorithme à la i-ème étape.
On cherche $indice\_min$, l'indice de la valeur minimale de T entre $i+1$ et $n-1$.
Comme toutes les valeurs de $T[i]$ à $T[n-1]$ sont toutes plus grandes que toutes les valeurs de $T[0]$ à $T[i-1]$, on en déduit que $T[indice\_min]$ est bien plus grande que toutes les valeurs de $T[0]$ à $T[i-1]$.
Ensuite on échange $T[indice\_min]$ et $T[i]$, donc $T[i]$ est bien plus grande que toutes les valeurs de $T[0]$ à $T[i-1]$.
Comme de plus toutes les valeurs de $T[0]$ à $T[i-1]$ sont triées, on en déduit que toutes les valeurs de $T[0]$ à $T[i]$ sont triées.
Soit :

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$
> ET
> Pour tout $0 \leq k \leq i$ et tout $i+1 \leq j \leq n-1$, $T[k] \leq T[j]$.

L'invariant est vrai.

La correction partielle est terminée. L'algorithme trie bien les valeurs du tableau.

g) Complexité

L'algorithme de tri par sélection :

```
1  pour i allant de 0 à n-1 faire
2  |  indice_min ← i ;
3  |  pour j allant de i+1 à n-1 faire
4  |  |  si T[indice_min] > T[j] alors
5  |  |  |  indice_min ← j ;
6  |  |  fin si
7  |  fin pour
8  |  temp ← T[i] ;
9  |  T[i] ← T[indice_min] ;
10 |  T[indice_min] ← temp ;
11 fin pour
```

On compte toutes les opérations :

- **Ligne 1 à 11** : on répète n fois
  - **ligne 1 :** 1 opération (on incrémente i de 0 jusqu'à $n-1$)
  - **ligne 2 :** 1 opération (on affecte une valeur à $indice\_min$)
  - **Lignes 3 à 6** : on répète $n-i-1$ fois
    - **ligne 3 :** 1 opération (on incrémente j de $i+1$ jusqu'à $n-1$)
    - **ligne 4 :** 1 opération (on effectue un test)
    - **ligne 5 :** 1 opération (on affecte une valeur à $indice\_min$)
  - **ligne 8 :** 1 opération (on affecte une valeur à $temp$)
  - **ligne 9 :** 1 opération (on affecte une valeur à $T[i]$)
  - **ligne 10 :** 1 opération (on affecte une valeur à $T[indice\_min]$)

La question qui demeure est : « combien de fois répète-t-on la ligne 5 ? ». Comme il est impossible d'y répondre, à nouveau, la question réelle est :

« Combien de fois répète-t-on la ligne 5 dans le pire cas ? »

Ce qui nous ramène toujours à cette autre question :

« Quel est le pire cas ? »

Réponse :

« Lorsque le tableau est rangé dans l'ordre décroissant ! »

En effet à ce moment là, la valeur minimale parmi les valeurs non triés est toujours celle qui est la plus à droite.

Comptons alors le nombre de répétitions des lignes 3 à 6.

- Si $i = 0$, on effectue $n - 1$ tests pour savoir quelle la valeur minimale.
  $n - 1$ répétitions.
- Si $i = 1$, on effectue $n - 2$ tests pour savoir quelle la valeur minimale entre $T[1]$ et $T[n - 1]$.
  $n - 2$ répétitions. ...
- Pour i, on effectue $n - i - 1$ tests pour savoir quelle la valeur minimale entre $T[i]$ et $T[n - 1]$.
  $n - i - 1$ répétitions. ...
- Si $i = (n - 2)$, on effectue $n - (n - 2) - 1 = 1$ test pour savoir quelle la valeur minimale entre $T[n - 2]$ et $T[n - 1]$.
  1 répétition.

Donc nous avons au total :

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

A nouveau grâce à nos amis mathématiciens, nous avons donc le nombre de répétitions dans le pire des cas :

$$S = \frac{(n - 1)n}{2}$$

On peut maintenant compter le nombre total d'opérations :

$$Nb = \underbrace{n \times 5}_{\text{les répétitions des lignes 1, 2, 8, 9 et 10}} + \underbrace{\frac{(n - 1)n}{2} \times 3}_{\text{les répétitions de toutes les lignes 3, 4 et 5}}$$

$$Nb = 5n + \frac{3}{2}n^2 - \frac{3}{2}n$$

$$Nb = \frac{3}{2}n^2 - \frac{7}{2}n$$

Le nombre d'opérations obtenu est une fonction polynôme du second degré.

On dit alors que la **complexité est quadratique**. La complexité est en $O(n^2)$.

IV. Un petit bilan

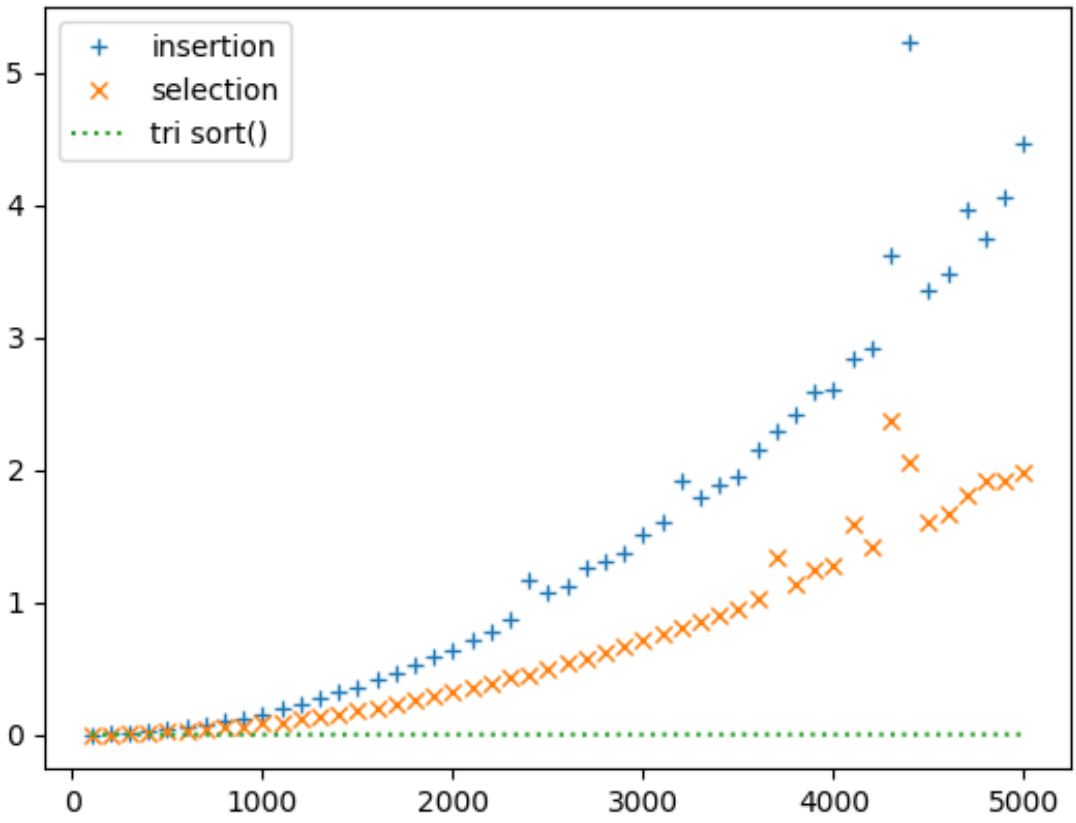
Les deux algorithmes ont la même complexité, ce qui signifie qu'ils sont dans la même famille d'efficacité.
De plus, on a constaté que le pire cas est dans les deux cas le même, c'est quand le tableau est .....
Par contre l'efficacité de deux algorithmes n'est pas la même dans le meilleure cas qui est : .....

- pour le tri par insertion dans le meilleur cas la complexité est : .....
- pour le tri par sélection dans le meilleur cas la complexité est : .....

Ainsi, si le tableau est presque trié par ordre ....., on choisira l'algorithme .....
et si le tableau est presque trié par ordre ....., on choisira l'algorithme .....

De la même manière, mais à un niveau beaucoup plus complexe, lorsqu'on utilise la méthode `.sort()` en Python, cette méthode va d'abord faire une analyse rapide du tableau pour déterminer quel est l'algorithme qui sera le plus efficace en fonction de la configuration du tableau [4]. Même mieux, l'algorithme va repérer différentes zones dans le tableau et y appliquer différents algorithmes.

Même si ce n'est pas une preuve, on peut installer une compteur/horloge au sein d'un programme en Python, pour tester les différents algorithmes et leur efficacité. Attention, ce programme [5] met un certain temps avant de terminer tous les tests. Voici le résultat :



4. Pour ceux qui en veulent toujours plus, la méthode `.sort()` à une complexité en $O(n\log(n))$ et on ne peut actuellement pas faire mieux.
5. Vous pouvez retrouver le code de ce programme sur mon Github.

V. Exercices

Exercice 1 (*)
Ranger à la main, par ordre croissant le tableau suivant à l'aide de l'algorithme de tri par insertion :

66	64	10	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5

Exercice 2 (*)
Ranger à la main, par ordre croissant le tableau suivant à l'aide de l'algorithme de tri par sélection :

66	64	10	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5

Exercice 3 (*)
`T` est un tableau contenant uniquement des booléens dans le désordre. On veut le ranger en plaçant tous les `False` en premier puis tous les `True`.
Proposer un algorithme (puis un programme en Python) puis étudier sa complexité.

Exercice 4 (*)
Écrire un algorithme (puis un programme en Python) qui permette de vérifier si un tableau est ordonnée, tout en précisant si dans l'ordre croissant ou décroissant.
Étudier sa complexité.

Exercice 5 (*)
Écrire un algorithme (puis un programme en Python) qui range dans l'ordre croissant un tableau déjà rangé dans l'ordre décroissant.

Exercice 6 (Ranger comme un prof de math - **)
Une légende raconte [6] que pour noter leurs copies, les profs de mathématiques ont chez eux un escalier comportant exactement 21 marches numérotées de 0 à 20. Ils lancent leurs copies et *automatiquement* les copies valant 0 tombent sur la marche notée 0, les copies valant 1 tombent sur la marche notée 1, ... , les copies valant 20 tombent sur la marche notée 20.
L'action aléatoire (lancer les copies en l'air) a eu pour effet de ranger les copies dans l'ordre !
Le but de cet exercice est d'écrire l'algorithme correspondant dans un cas un peu plus simple.
Écrire, en Python, un programme qui prend un tableau de 5 valeurs choisis totalement au hasard entre 0 et 5 en entrée et qui renvoie le même tableau avec les mêmes valeurs mais dans un ordre différent, choisi aléatoirement.
Recommencer jusqu'à ce que le tableau soit bien rangé.

6. Toujours se méfier des légendes ... mais dans toutes légendes, il y a une part de vérité.

Correction de l'exercice 1

66	64	10	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5
64	66	10	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5
10	64	66	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5
10	64	66	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5
10	55	64	66	99	7	63	98	41	1	59	89	96	47	12	0	46	5
7	10	55	64	66	99	63	98	41	1	59	89	96	47	12	0	46	5
7	10	55	63	64	66	99	98	41	1	59	89	96	47	12	0	46	5
7	10	55	63	64	66	98	99	41	1	59	89	96	47	12	0	46	5
7	10	41	55	63	64	66	98	99	1	59	89	96	47	12	0	46	5
1	7	10	41	55	63	64	66	98	99	59	89	96	47	12	0	46	5
1	7	10	41	55	59	63	64	66	98	99	89	96	47	12	0	46	5
1	7	10	41	55	59	63	64	66	89	98	99	96	47	12	0	46	5
1	7	10	41	55	59	63	64	66	89	96	98	99	47	12	0	46	5
1	7	10	41	47	55	59	63	64	66	89	96	98	99	12	0	46	5
1	7	10	12	41	47	55	59	63	64	66	89	96	98	99	0	46	5
0	1	7	10	12	41	47	55	59	63	64	66	89	96	98	99	46	5
0	1	7	10	12	41	46	47	55	59	63	64	66	89	96	98	99	5
0	1	5	7	10	12	41	46	47	55	59	63	64	66	89	96	98	99

Correction de l'exercice 2

66	64	10	99	55	7	63	98	41	1	59	89	96	47	12	0	46	5
0	64	10	99	55	7	63	98	41	1	59	89	96	47	12	66	46	5
0	1	10	99	55	7	63	98	41	64	59	89	96	47	12	66	46	5
0	1	5	99	55	7	63	98	41	64	59	89	96	47	12	66	46	10
0	1	5	7	55	99	63	98	41	64	59	89	96	47	12	66	46	10
0	1	5	7	10	99	63	98	41	64	59	89	96	47	12	66	46	55
0	1	5	7	10	12	63	98	41	64	59	89	96	47	99	66	46	55
0	1	5	7	10	12	41	98	63	64	59	89	96	47	99	66	46	55
0	1	5	7	10	12	41	46	63	64	59	89	96	47	99	66	98	55
0	1	5	7	10	12	41	46	47	64	59	89	96	63	99	66	98	55
0	1	5	7	10	12	41	46	47	55	59	89	96	63	99	66	98	64
0	1	5	7	10	12	41	46	47	55	59	89	96	63	99	66	98	64
0	1	5	7	10	12	41	46	47	55	59	63	96	89	99	66	98	64
0	1	5	7	10	12	41	46	47	55	59	63	64	89	99	66	98	96
0	1	5	7	10	12	41	46	47	55	59	63	64	66	99	89	98	96
0	1	5	7	10	12	41	46	47	55	59	63	64	66	89	99	98	96
0	1	5	7	10	12	41	46	47	55	59	63	64	66	89	96	98	99
0	1	5	7	10	12	41	46	47	55	59	63	64	66	89	96	98	99
0	1	5	7	10	12	41	46	47	55	59	63	64	66	89	96	98	99

Correction de l'exercice 3

```python
import random as rd

#taille du tableau
n=10

#un tableau de boolean
T=[bool(rd.randint (0 ,1)) for i in range(n)]
print(T)

#on compte le nombre de False
nb_False=0
for i in range(0,n) :
    if T[i]==False:
        nb_False += 1

#on remplit le tableau de False puis de True
for i in range(0,nb_False) :
    T[i] = False
for i in range(nb_False, n):
    T[i] = True

#petite vérification
print(T)
```

La correction est présente sur le lien : `https://repl.it/@lebonprof/ex3booleens`
La complexité est en $O(n)$... Pourquoi ?

Correction de l'exercice 4
J'ai rajouté une boucle `while(ordre==0) :` pour lancer le programme tant que l'on n'obtient pas un tableau bien rangé.

```
1  import random as rd
2
3  #taille du tableau
4  n=5
5
6  #pour rentrer dans la bouclue
7  ordre = 0
8  essais = 0
9
10 while(ordre==0) :
11     #un tableau de boolean
12     T=[rd.randint (0 ,5) for i in range(n)]
13     print(T)
14
15     #variable ordre
16     #décroissant = -1, rien = 0, croissant = 1
17     #on teste T[0] et T[1]
18     ordre = -1
19     if T[0]<= T[1]:
20         ordre = 1
21
22     #on vérifie si l'ordre est respectée
23     i=2
24     while i<len(T) and ordre !=0 :
25         if T[i-1] < T[i] and ordre == -1 :
26             ordre = 0
27         if T[i-1] > T[i] and ordre == 1 :
28             ordre = 0
29         i += 1
30
31     print(ordre)
32     essais += 1
33
34 print(essais,"essais")
```

La correction est présente sur le lien : `https://repl.it/@lebonprof/ex4range`
La complexité a étudier concerne uniquement la partie à l'intérieure de la boucle `while(ordre==0) :`, autrement dit uniquement entre les lignes 13 et 33. Elle est en $O(n)$... Pourquoi ?

Correction de l'exercice 5

```
1  # -*- coding: utf-8 -*-
2
3  import random as rd
4
5  #taille du tableau
6  n=10
7
8  ##########################################
9  #création d'un tableau dans l'ordre décroissant
10 ##########################################
11 #un tableau de taille n rempli de zéros
12 T=[0]*n
13 T[0] = rd.randint(0,1000)
14 for i in range (1,n):
15     T[i] = rd.randint(0,T[i-1])
16 #vérification
17 print(T)
18
19 ##########################################
20 #rangement du tableau dans l'ordre croissant
21 ##########################################
22 #on prend un tableau intermédiaire de même taille
23 Temp =[0]*n
24
25 #les derniers seront les premiers
26 for i in range(n):
27     Temp[i] = T[n-1-i]
28
29 #on remet les valeurs de Temp dans T dans le même ordre
30 for i in range(n):
31     T[i] = Temp[i]
32
33 #vérification
34 print(T)
```

La correction est présente sur le lien : `https://repl.it/@lebonprof/ex5decroitcroit`
La complexité a étudier concerne uniquement la deuxième partie des lignes 23 à 31. Elle est en $O(n)$... Pourquoi ?