Algorithmes de tri

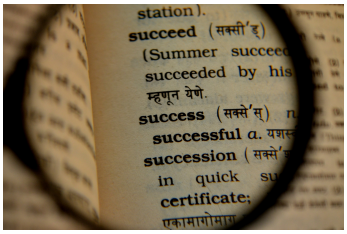
I Introduction

À la fin de cette heure, vous aurez appris de nouveaux algorithmes qui sont à la base de beaucoup d'autres. Vous saurez pourquoi ces algorithmes sont utilisés en permanence dans un ordinateur et enfin vous verrez qu'on peut apprendre l'informatique en dansant !

I.1 Pourquoi trier ?

Pour trouver plus vite ! Ce serait tellement plus lent de chercher dans un dictionnaire où les mots sont dans le désordre... Trier, c'est ce qu'un ordinateur fait tout le temps : pour optimiser l'organisation de la mémoire, pour chercher efficacement une information etc...

Les algorithmes de tri sont à la base des beaucoup d'autres algorithmes. Leur apprentissage a donc un intérêt pour pouvoir construire de nouveaux algorithmes efficaces.



I.2 Comment triez-vous ?

`https://nsi.infobrisson.fr/files/Cours/07_Algorithmes_de_tri/drag_and_drop.html`

I.2.1 Le tri par sélection

Savoir-faire — Tri par sélection

Dans le tri par sélection,

- on recherche le plus petit élément du tableau, et on l'échange avec l'élément d'indice 0 ;
- puis on recherche le deuxième plus petit élément du tableau, et on l'échange avec l'élément d'indice 1 ;
- on continue de cette façon jusqu'à ce que le tableau soit entièrement trié.

Exercice

Exercice 1

Appliquer l'algorithme du tri par sélection au tableau ci-dessous :

3	8	3	2	1	7	0	5	7	4

I.2.2 Le tri par insertion

Savoir-faire — Tri par insertion

Dans le tri par insertion, 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.
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.

Exercice

Exercice 2

Appliquer l'algorithme du tri par insertion au tableau ci-dessous :

3	8	3	2	1	7	0	5	7	4

Info

Le tri par sélection et le tri par insertion sont des tris dits *en place* ce qui signifie que l'on modifie le tableau de départ pour trier. Les tris qui ne travaillent pas en place ont besoin d'un second tableau (de même taille) pour tier ce qui peut poser problème si l'on a peu de place en mémoire ou que les tableaux sont très grands.

Astuce

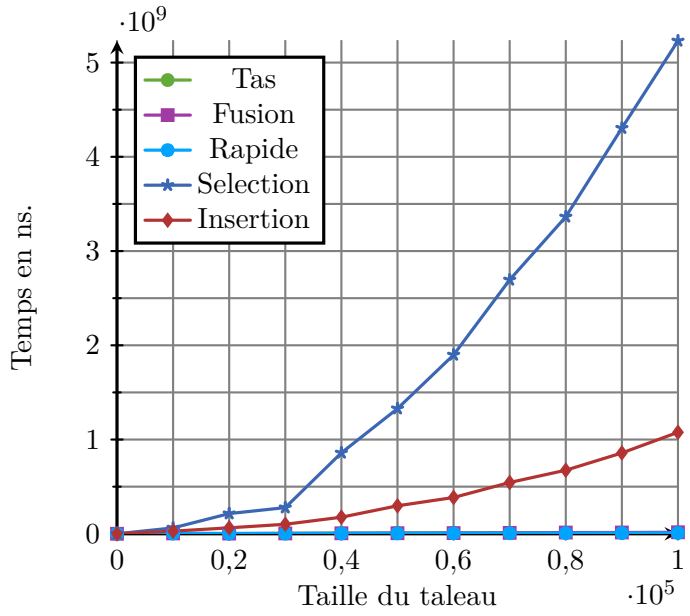
Jusqu'ici nous avons trié uniquement des tableaux d'entiers mais tous les algorithmes continuent de fonctionner pour des éléments que l'on sait comparer comme des chaînes de caractères par exemple.

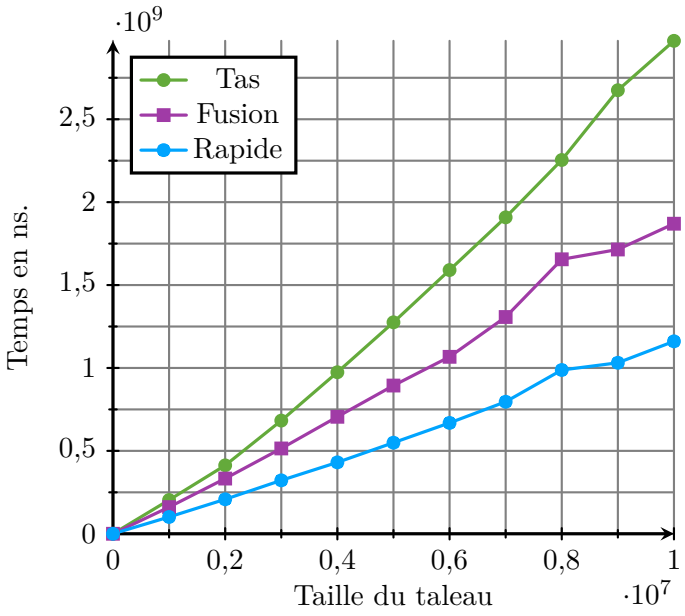
I.2.3 Danser pour trier

Astuce

Un moyen très visuel de comprendre les algorithmes de tri est proposé ici : `https://www.youtube.com/user/AlgoRythmics/videos`.

I.3 Faire le tri dans les algorithmes de tri





Info

Le *tri fusion* est l'un des algorithmes de tri les plus rapides. Il a été inventé en 1945 par John von Neumann.

II Tri par sélection

Le tri par sélection parcourt le tableau de la gauche vert la droite, en maintenant sur la gauche une partie déjà triée et à sa place définitive :

déjà trié	par encore trié

A chaque étape, on cherche le plus petit élément dans la partie de droite (non triée), puis on l'échange avec l'élément le plus à gauche de la partie non triée. Ainsi, la première étape va déterminer le plus petit élément et le placer tout à gauche du tableau. Puis, la seconde étape va déterminer le deuxième plus petit élément et le placer dans la deuxième case du tableau, et ainsi de suite. Par construction, la partie de gauche déjà triée ne contient que des éléments inférieurs ou égaux à ceux de la partie droite restant à trier. On commence par se donner une fonction **echange** pour échanger les deux éléments d'un tableau **t** situés aux indices **i** et **j**.

```
def echange(t, i, j):
  temp = t[i]
  t[i] = t[j]
  t[j] = temp
```

La fonction **echange** stocke ici la valeur **t[i]** dans une variable temporaire **temp**, puis copie la valeur de l'indice **j** dans la valeur **i**, et finalement écrit la valeur de la variable **temp** à l'indice **j**. Pour réaliser le tri par sélection, on va utiliser une boucle parcourant toutes les cases du tableau, de la gauche vers la droite.

```
for i in range(len(t)):
```

Il faut maintenant chercher le plus petit élément de la partie de droite, c'est-à-dire la partie du tableau allant des indices **i** à **len(t)-1** inclus. Ecrivont alors une boucle qui parcourt cette partie du tableau et on va utiliser une variable **memoire** pour retenir l'indice de la plus petite valeur rencontrée.

```
memoire = i
for j in range(i + 1, len(t)):
  if t[j] < t[memoire]:
    memoire = j
```

La variable **memoire** sera initialisée avec l'indice **i**. Dès qu'une valeur à l'indice **j** est plus petite que celle de l'indice **m**, on met à jour l'indice de cette plus petite valeur. Finalement, lorsque l'on sort de la boucle, on a trouvé la position **m** de la plus petite valeur de la partie de droite. Il ne reste donc qu'à effectuer l'échange des valeurs aux indices **i** et **m**, avec notre fonction **echange**. Voici le programme complet ci-dessous.

```
def echange(t, i, j):
  temp = t[i]
  t[i] = t[j]
  t[j] = temp

def tri_par_selection(t):
  """tri du tableau t dans l'ordre croissant"""
  for i in range(len(t)):
    # t[0..i[ est trié et <= à t[i..[
    # on cherche le minimum de t[i..[
    memoire = i
    for j in range(i + 1, len(t)):
      if t[j] < t[memoire]:
        memoire = j
    echange(t, i, j)
```

III Tri par insertion

L'algorithme de tri par insertion fonction sur le même principe que le tri par sélection, en parcourant le tableau de la gauche vers la droite et en maintenant une partie déjà triée sur la gauche :

déjà trié	par encore trié

Plutôt que de cherche la plus petite valeur dans la partie non encore triée, le tri par insertion va *insérer* la première valeur non encore triée (de droite) dans la partie déjà triée (de gauche). On va, pour cela, décaler d'une cas evers la droite tous les éléments déjà triés qui sont plus grands que la valeur à insérer, puis déposer cette dernière dans la case ainsi libérée. Commençons par écrire une fonction **insere(t, i, v)** qui réalise l'insertion d'une valeur **v** dans la partie du tableau **t[0], ..., t[i]**, sachant que les valeurs **t[0], ..., t[i-1]** sont déjà triées.

	0		i	
t	éléments triés		?	...

On va alors décaler un par un les éléments vers la droite jusqu'à trouver la bonne position **j** de la valeur **v**, pour se retrouver au final vers la situation suivante :

	0		j		i	
t	éléments triés		v	éléments triés		...

Pour écrire le tri par insertion, il suffit maintenant d'insérer successivement toutes les valeurs du tableau avec la fonction **insere**, en procédant de la gauche vers la droite avec une boucle *for*.

```
def insere(t, i, v):
  """insère v dans t[0..i[ supposé trié"""
  j = i
  while j > 0 and t[j - 1] > v:
    t[j] = t[j - 1]
    j = j - 1
  t[j] = v

def tri_par_insertion(t):
  """trie le tableau t dans l'ordre croissant"""
  for i in range(1, len(t)):
    insere(t, i, t[i])
```

IV Les tris fournis par Python

Python fournit des fonctions pour trier des tableaux, plus efficaces que les tris par sélection et par insertion. Elles se présentent de deux façons différentes, selon que l'on veut obtenir une copie triées du tableau sans le modifier, ou au contraire modifier le tableau pour le trier, comme précédemment. La fonction **sorted** correspond au premier cas. Elle prend en argument un tableau et renvoie un *nouveau tableau*, trié, contenant les mêmes éléments.

```
>>> t = [55, 2, 1, 34, 3, 21, 5, 8, 13, 1, 0]
>>> sorted(t)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> t
[55, 2, 1, 34, 3, 21, 5, 8, 13, 1, 0]
```

Le tableau *t* n'a pas été modifié. La construction de t.sort(), en revanche, modifie le tableau **t** pour le trier, sans rien renvoyer.

```
>>> t.sort()
>>> t
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Ces deux fonctions sont largement plus efficaces que les tris par sélection et par insertion. Trier un tableau d'un million d'élément, par exemple, est instantané

Savoir-faire — **Tri par sélection**

Le **tri par sélection** et le **tri par insertion** sont deux algorithmes de tri élémentaires, qui peuvent être utilisés pour **trier des tableaux**. Le tri par insertion a un meilleur comportament que le tri par sélection lorsque le tableau est presque déjà trié. Les deux restent **peu efficaces** dès que les tableaux contiennent plusieurs milliers d'éléments. Il existe de **meilleurs algorithmes** de tri, plus complexes, dont celui offert par Python avec les fonction **sort** et **sorted**.

V Détermination du coût d'un algorithme

Il existe donc deux tris que nous allons traiter ici, et il est logique de se demander pourquoi aborder deux tris différents, mais surtout lequel est le meilleur ! Hélas, il n'y a pas de réponse absolue. On évaluer l'efficacité d'un algorithme en déterminant son *coût*. Ce coût représente la quantité d'opération qui seront nécessitées à la machine pour traiter le problème. On évalue finalement ces opérations en temps d'exécution machine dans trois cas différents : le *meilleur*, le *pire* et le *moyen* cas. On note enfin le coût d'un algorithme à l'aide de la lettre Grecque Théta paramétrée de la complexité : θ(n).

- $\theta(n)$ pour un coût linéaire
- $\theta(n^2)$ pour un coût quadratique

V.1 Coût du tri par sélection

Le tri par sélection parcourt dans un premier temps la totalité du tableau à la recherche de la plus petite valeur, ce qui prend donc un temps proportionnel à la taille du tableau. Une fois trouvé, on va de nouveau parcourir le tableau à la recherche de la valeur directement supérieur, soit sur n-1 éléments. On va donc réaliser cette opération d'évaluation de la valeur contenue :

$$\text{n} + \text{n-1} + \text{n-2} + ... + 1$$

En factorisant le résultat précédent, on trouve que cette somme vaut :

$$\text{n(n+1)/2}$$

On peut simplifier cette expression en ne considérant plus que le produit n*n, soit n^2. On parle alors de **coût quadratique** $\theta(n^2)$ pour le meilleur, le pire et le cas moyen.

V.2 Coût du tri par insertion

Le tri par insertion n'est pas meilleur que le tri par sélection. A chaque fois, il est nécessaire de parcourir notre tableau pour déterminer l'emplacement de l'insertion. Dans le pire des cas (et aussi le cas moyen), on obtient le nombre d'opération suivant :

$$1 + 2 + 3 + ... + \text{n-1}$$

ce qui revient donc à un coût quadratique $\theta(n^2)$. Il existe néanmoins une configuration où le tri par insertion sera plus performant. Si la position où la valeur doit être insérée est proche du dernière indice considéré (à savoir t[i - 1]), alors il est inutile de parcourir une nouvelle fois le tableau. C'est donc le cas pour les tableau qui sont "presque" triés. On passe avec ce constat d'un coût quadratique (comme pour le tri par sélection), à un **coût linéaire** $\theta(n)$ (donc proportionnel à n) pour le meilleur cas.

VI Invariant de boucle et terminaison

Un **invariant de boucle** est une propriété qui est vraie **avant** et **après** l'exécution d'un tour de la boucle. Cette propriété permet de valider le comportement d'un algorithme. La terminaison permet, quant à elle, de vérifier que l'algorithme s'arrêtera. Cette terminaison pourra être appliquée sur une donnée ou sur toutes les données.

Tri par sélection

Prenons comme invariant de boucle pour le tri par sélection :

Les *i* premiers éléments sont triés.

Cet invariant est vrai **dès le départ** car on commence à $i = 1$. Cela **reste vrai** car on ajoute à droite (dans le tableau) un élément plus grand que les autres. Cela nous permet donc de conclure que l'algorithme est **correct** car le résultat obtenu est une liste triée La *terminaison* du tri par sélection est vérifiée puisque l'on a deux boucles for qui sont imbriquées, l'une ayant n-1 itérations, et l'autre au maximum n-1, puis n-2 ... jusqu'à 1.

Tri par insertion

Prenons comme invariant de boucle pour le tri par insertion :

Les *i* premiers éléments sont triés.

Cet invariant est vrai **dès le départ** car on commence à $i = 1$. Cela **reste vrai** car on ajoute le nouvel élément à sa place. Cela nous permet donc de conclure que l'algorithme est **correct** car le résultat obtenu est une liste triée La *terminaison* du tri par insertion est vérifiée puisqu'elle se compose d'une boucle for. Concernant la condition while, on démarre de la valeur **i** qui est égale à un entier naturel et l'on va au maximum jusqu'à 0, en décrémentant **i** à chaque étape.

VII Exercices de fin de chapitre

Exercice

Exercice 3

Il n'est pas possible de comparer deux tableaux de dimensions différentes, puisque chacun des indices des deux tableaux doivent être comparés en même temps. Si on essaye d'effectuer quand même la comparaison, le programme retournera une erreur : *out of range*. Modifier la fonction ci-dessous pour effectuer une précondition permettant d'assurer que les deux tableaux passés en paramètre ont la même dimension.

```
def compare_tableau(t, u):
  for i in range(len(t)):
    if t[i] != u[i]:
      return False
  return True
```

Exercice

Exercice 4

Écrire une fonction **est_trie(t)** qui renvoie **True** si le tableau est trié par ordre croissant et **False** sinon. Il sera intéressant de se demander ce que l'on peut faire si l'on détecte que le tableau passé en paramètre n'est pas trié.

Exercice

Exercice 5

Au lieu de modifier le contenu du tableau pour le trier, on peut aussi renvoyer le résultat du tri dans un nouveau tableau. Écrire une fonction **tri_par_insertion_externe** qui réalise cela avec le tri par insertion. On pourra se servir de la fonction **insere**.

Exercice

Exercice 6

Écrire une fonction **plus_petit** qui prend en paramètres un tableau **t** et un entier **k** supposé inférieur à la longueur de **t** et qui renvoie un tableau contenant, dans l'ordre, les **k** plus petits éléments de **t**. On pourra s'inspirer de l'exercice précédent et réutiliser la fonction **insere**.

Exercice

Exercice 7

Écrire une fonction qui prend en argument un tableau d'entiers *trié* et affiche son contenu sous la forme d'un histogramme, comme ci-dessous :

```
1 fois 0
2 fois 2
1 fois 3
5 fois 6
...
```