Algorithmes de tri

Objectifs :

- Connaître l'algorithme de tri par insertion ;
- Connaître l'algorithme de tri par sélection ;
- Notion de preuve de correction ;
- Notion de complexité d'un algorithme ;
- Les fonctions de tri de Python.

Dans la langue courante *trier* et *classer* ont des sens différents. On peut trier ses déchets en plusieurs tas. On peut *classer* des mots par ordre alphabétique. En informatique, trier a le sens de classer.

Trier un tableau est un problème délicat. Cela peut être très long si la taille du tableau est conséquente, mais c'est là qu'utiliser l'informatique devient intéressant. Un algorithme peut être bien meilleur dans certains cas que dans d'autres. C'est pourquoi il existe de nombreux algorithmes de tri et que l'étude et l'amélioration des algorithmes de tri est toujours un sujet actuel de la recherche en informatique. D'ailleurs, l'algorithme de tri utilisé par Python est le TimSort qui a été inventé en 2002

I Intérêt du tri

Dès que l'on a besoin de classer des données, il faut les trier. Nous avons vu que pour trouver une donnée dans un tableau la bonne méthode est d'utiliser l'algorithme de recherche dichotomique. Cette méthode très rapide demande d'avoir un tableau trié. Calculer une médiane demande aussi d'avoir un tableau trié. De nombreux algorithmes, par exemple celui des k plus proches voisins que nous verrons plus tard, utilisent des tris. Un autre intérêt est qu'il est aussi beaucoup plus simple de comparer deux listes triées. En effet, pour savoir si deux tableaux sont égaux, il suffit de comparer éléments par éléments. Si deux éléments sont différents alors les listes sont différentes, si après avoir comparé tous les éléments, il n'y a pas eu de différence, alors les tableaux sont égaux.

Une implémentation Python possible de la comparaison de tableaux est :

```
def compare_tableaux(t1: list,t2: list) -> bool:
  """
  Retourne True si les tableaux sont égaux et False sinon
  """
  for i in range(len(t1)):
    if t1[i] != t2[i]:
      return False
  return True
```

II Tri par sélection

A Présentation de l'algorithme

L'algorithme : Il se fait en deux étapes :

- On sélectionne le plus petit élément.
- On le met à sa place.
- On effectue ensuite ces deux étapes en commençant au deuxième élément, puis le troisième et ainsi de suite jusqu'à l'avant-dernier.

Propriétés du tri par sélection

- tri en **place** : Il n'est pas nécessaire de faire une copie de la liste ;
- tri **non stable**: Deux éléments égaux ne resteront pas nécessairement dans le même ordre.

Une implémentation Python possible :

```
def tri_selection(L):
  for i in range(len(L)-1):
    #Recherche l'indice du plus petit élément de i à la fin
    mini = i
    for j in range(i+1,len(L)):
      if L[j] < L[mini]:
        mini = j
    # Échange les valeurs des indices i et mini
    tmp = L[i]
    L[i] = L[mini]
    L[mini] = tmp
```

B correction de l'algorithme

Définitions

- Un algorithme est **correct** lorsqu'il fait ce qu'on attend de lui.
  Pour un algorithme itératif (qui contient au moins une boucle) on montre la correction de l'algorithme à l'aide d'un **invariant de boucle.**
- Un **invariant de boucle** est une propriété qui est vraie avant et après chaque répétition de la boucle.

Montrons que l'invariant de boucle pour cet algorithme est : *à la fin du tour i de la boucle for, les éléments d'indices 0 à i sont dans l'ordre croissant.*

- Au premier tour de boucle, $i=0$, le plus petit élément est recherché dans toute la liste. Puis il est placé par échange dans la case 0.
- Supposons qu'à la fin du tour $i-1$ tous les éléments de la case 0 à la case $i-1$ soient à leur place définitive et dans l'ordre croissant.

mini = 3

Élément	1	3	4	7	8	6	9
Indice	0	1	2	3	4	5	6

Éléments placés i

- Au début du tour i, on recherche l'indice du plus petit élément de la case i à la fin.

mini = 5

Élément	1	3	4	7	8	**6**	9
Indice	0	1	2	3	4	**5**	6

Éléments placés/triés i

- L'élément d'indice mini est le $i^{ème}$ plus petit élément, il est plus grand que les éléments d'indice 0 à $i-1$ et le plus petit des éléments suivants. Ce plus petit élément est donc celui qui vient juste après celui d'indice $i-1$. Il est donc échangé avec le contenu de la case i. Les éléments d'indices 0 à i sont maintenant à leur place.

Élément	1	3	4	6	8	7	9
Indice	0	1	2	3	4	5	6

Éléments placés/triés

- Lors du dernier tour de boucle, i vaut len(L)-1. À la fin de ce tour, tous les éléments de la case 0 jusqu'à la case len(L)-1 incluse sont à leur place. La liste est donc triée.

L'algorithme est donc **correct** : il tri bien la liste comme il est censé le faire.

C Terminaison

À chaque étape, on retire un élément parmi ceux restants à trier : le nombre d'éléments qui reste à trier est un *variant* de la boucle, c'est-à-dire une quantité entière positive qui décroît à chaque itération.

D Complexité de l'algorithme

Définition: Déterminer la complexité d'un algorithme, c'est étudier la quantité de ressources (temps, espace mémoire, etc.) dont a besoin un algorithme pour résoudre un problème algorithmique donné. On se limite a étudier la complexité en temps.

On a vu que pour trouver le minimum dans un tableau de *n* éléments, il faut effectuer *n* comparaison.

Dans notre exemple qui contient 7 cartes :

- On effectue 6 comparaisons pour déterminer le plus petit des 7 éléments.
- On effectue 5 comparaisons pour déterminer le plus petit des 6 éléments restants.
- On effectue 4 comparaisons pour déterminer le plus petit des 5 éléments restants.
- On effectue 3 comparaisons pour déterminer le plus petit des 4 éléments restants.
- On effectue 2 comparaisons pour déterminer le plus petit des 3 éléments restants.
- On effectue 1 comparaison pour déterminer le plus petit des 2 éléments restants.
- On n'a pas besoin d'effectuer de comparaison pour déterminer que la dernière carte est la plus petite restante.

On a effectué $6+5+4+3+2+1+0=21$ comparaisons. Comme $1+2+\cdots+n=\frac{n(n+1)}{2}$

Plus généralement pour une liste de taille *n* on effectue $\frac{n(n+1)}{2}=\frac{n^2}{2}+\frac{n}{2}$ comparaisons. C'est un polynôme du second degré.

On dit que la complexité est **quadratique**. On le note $O(n^2)$ (se lit "grand o en n²").

Remarque: Cela permet de retrouver la terminaison de l'algorithme car pour un tableau de taille *n*, il y aura de l'ordre de n^2 comparaisons qui est un nombre fini.

III Tri par insertion

A Présentation de l'algorithme

L'algorithme :

Le principe du tri par insertion est de trier les éléments du tableau comme avec des cartes :

- On prend nos cartes mélangées dans notre main.
- On crée deux ensembles de carte, l'un correspond à l'ensemble de cartes triées, l'autre contient l'ensemble des cartes restantes (non triées).
- On prend au fur et à mesure, une carte dans l'ensemble non trié et on l'insère à la bonne place dans l'ensemble de cartes triées.
- On répète cette opération tant qu'il y a des cartes dans l'ensemble non trié.

Propriétés de tri par insertion :

- tri **en place** : il n'est pas nécessaire de faire une copie de la liste ;
- tri **stable** : deux éléments égaux resteront dans l'ordre ;
- trie **en ligne** : les éléments de la liste pourraient être fournis au fur et à mesure.

Une implémentation Python possible :

```
def tri_insertion(L: list) -> None:
  """
  Modifie la liste L passée en paramètre de façon à ce qu'elle soit
  triée en ordre croissant
  """
  for j in range(1,len(L)):
    cle = L[j]
    i = j-1
    while i>=0 and L[i]> cle:
      L[i+1] = L[i]
      i = i-1
    L[i+1] = cle
```

B Preuve de correction

Montrons qu'à la fin d'un tour de la boucle `for,` *les valeurs de la liste L sont triées jusqu'à la case j inclus*. C'est **l'*invariant de boucle*** de cet algorithme.

- Au début *j* vaut 1 et la liste est réduite à la case d'indice 0, est bien triée.
- Supposons qu'a la fin du tour $j-1$, les valeurs de la liste soient triées jusqu'à la case $j-1$ incluse. On veut montrer que lors du tour *j*, la case cle = L[j] sera insérée correctement et que la liste sera triée jusqu'à la case *j* incluse. On envisage deux cas.
  Cas 1: L[j]>= L[0]
  On est dans cette situation juste après le for:

3	5	7	8	**6**	2	11
	trié			**j**	Non trié	

  Juste avant le while:
  La clé est **6**

3	5	7	8	**6**	2	11
			i	**j**	Non trié	

On entre dans la boucle while. À la fin de la boucle, la situation est:
La clé est **6**

3	5	7	7	**8**	2	11
	i			**j**	Non trié	

Puis à la dernière ligne, à la fin de la boucle for:
La clé est **6**

3	5	**6**	7	**8**	2	11
	i			**j**	Non trié	

Cas 2: L[j] < L[0]. On vérifie sans mal que la boucle while quitte pour $i=-1$ et que la clé est bien insérée dans la case d'indice 0.

- La liste est donc bien triée jusqu'à la case *j*.

Conclusion: lorsque la boucle for termine son dernier tour *j* désine la dernière cas et on a montré que la liste était triée jusqu'à cette case. La liste est donc entièrement triée. L'algorithme est donc correct: il tri bien la liste.

C Terminaison

À chaque étape, on retire un élément parmi ceux restants à trier : le nombre d'éléments qui reste à trier est un *variant* de la boucle, c'est-à-dire une quantité entière positive qui décroît à chaque itération.

L'existence d'un variant prouve que *l'algorithme se termine en un temps fini*.

D Complexité de l'algorithme

Avec un tableau de taille 7:

- Le premier élément est tri. Il n'y a pas de comparaison.
- Le second élément demande 1 comparaison avec le premier.
- Le troisième élément sera comparé aux 2 premiers.
- Le quatrième aux 3 premiers.
- Le cinquième aux 4 premiers.
- Le sixième aux 5 premiers.
- Le septième et dernier au 6 premiers.

Il y a donc eu: $0+1+2+3+4+5+6=21$ comparaisons.

Plus généralement pour une liste de taille *n* on effectue $\frac{n(n+1)}{2}=\frac{n^2}{2}+\frac{n}{2}$ comparaisons.

La complexité est **quadratique**. On le note $O(n^2)$.

La complexité de ce tri est quadratique, mais dans le cas ou la liste est triée ou presque triée il serait linéaire. C'est-à-dire que le nombre de tours de boucle serait de l'ordre de la taille de la liste c'est à dire *n*. En effet dans ce cas il n'y a pas de comparaison à faire pour déterminer la place de l'élément à classer car il est déjà à sa place.

Remarque: Cela permet de retrouver la terminaison de l'algorithme car pour un tableau de taille *n*, il y aura de l'ordre de n^2 comparaisons qui est un nombre fini.

IV Les fonctions de tri en Python

Python dispose d'une fonction et d'une méthode de tri sur les tableaux. La fonction `sorted(t)` renvoie un tableau trié des éléments du tableau `t` qui n'est pas modifié. La méthode `t.sort()` trie le tableau t `t` sur place. Cela veut dire que le tableau `t` est modifié mais ses copies ne le sont pas. L'option `reverse = True` permet de trier dans l'ordre décroissant.
Ces fonctions sont d'une complexité optimale en $n\log(n)$.
L'option `key` permet de spécifier la fonction qui sera utilisée pour classer les éléments.

Exemples d'utilisation:

```
t = [5,0,4,11,7]
sorted(t) # Renvoie [0,4,5,7,11], t n'est pas modifié
sorted(t, reverse=True) # Renvoie [11,7,5,4,0], t n'est pas modifié
t.sort() # t == [0,4,5,7,11], t est modifié
t.sort(reverse=True) # t == [11,7,5,4,0], t est modifié

def f(x):
  #La première lettre en minuscule
  return x[0].lower()
print(sorted(['B','Ab','c','a'],key=f)) #Renvoie ['Ab', 'a', 'B',
'c']
# Le tri est en place avec cette fonction les éléments 'Ab' et 'a'
sont restés
# dans l'ordre initial
```

Pour aller plus loin:

- Cours sur les tris sur Lumni: https://www.lumni.fr/video/les-algorithmes-de-tri
- Visualiser des algorithmes de tri : https://visualgo.net/en/sorting
- Comprendre la notion de stabilité: http://lwh.free.fr/pages/algo/tri/stabilite_tri.html
- https://fr.wikipedia.org/wiki/Timsort

Sources :

- https://interstices.info/les-algorithmes-de-tri/
- https://fr.wikipedia.org/wiki/Correction_d%27un_algorithme
- https://fr.wikipedia.org/wiki/Th%C3%A9orie_de_la_complexit%C3%A9_(informatique_th%C3%A9orique)
- https://irem.univ-reunion.fr/IMG/html/tris.html