Cours / TP 21 - Les Algorithmes de tri

Introduction

Trier des données, consiste à les ranger suivant un ordre défini au préalable. Par exemple nous pouvons trier des données numériques en utilisant l'ordre défini en mathématiques : a est avant b si $a < b$.

Si nous trions l'ensemble de nombres 3, 8, 5 et 2, nous obtenons l'ensemble 2, 3, 5 et 8 et nous disons que les nombres sont rangés suivant l'ordre **croissant**. Si nous les rangeons dans l'ordre inverse, nous parlons d'ordre **décroissant**.

Nous procédons par des comparaisons successives entre deux éléments et effectuons éventuellement une permutation des éléments comparés. Le nombre de permutations est donc toujours inférieur au nombre de comparaisons.

Le **coût** ou la **complexité** d'un algorithme sera fonction du nombre de comparaisons et permutations effectuées par cet algorithme. Pour un algorithme donné, le coût peut être différent suivant les cas à traiter, il y a des cas favorables, et d'autres non.

Le tri par sélection

On parcourt la liste en sélectionnant le plus petit élément que l'on place en début de liste.

Présentation de l'algorithme

Pour chaque position de la liste, on va **sélectionner** le plus petit élément dans la suite de la liste et l'échanger avec la position actuelle (s'il est plus petit), d'où le nom de l'algorithme. Regardons une vidéo explicative.

- Algorithme :

```
pour i allant de 0 à taille(liste) − 1
    mini ← i
    pour j allant de i+1 à taille(liste)
        si liste[j] < liste[mini]
            mini ← j
    tmp ← liste[i]
    liste[i] ← liste[mini]
    liste[mini] ← tmp
```

Un exemple

```
12 -  5 - 11 - 20 -  7 (i=0) le plus petit élément de [5,11,20,7] est 5 (<12) -> on l'échange
avec 12
 5 - 12 - 11 - 20 -  7 (i=1) le plus petit élément de [11,20,7] est 7 (<12) -> on l'échange
avec 12
 5 -  7 - 11 - 20 - 12 (i=2) le plus petit élément de [20,12] est 12 (>11) -> on ne fait rien
 5 -  7 - 11 - 20 - 12 (i=3) le plus petit élément de [12] est 12 (<20) -> on échange avec 20
 5 -  7 - 11 - 12 - 20
```

Terminaison et correction

- Nous avons deux boucles `pour` imbriquées donc le nombre de passages dans ces deux boucles est parfaitement déterminé et il est évidemment **fini**.
- L'invariant de boucle sera dans ce cas : "*Pour chaque valeur de `i` traitée, les valeurs `liste[0:i−1]` seront déjà triées.*"
  L'invariant est vrai pour `liste[0]` puisqu'un seul élément est forcément trié...
  Après un passage pour indice égal à `i`, la liste `liste[0:i−1]` est triée et tous les éléments de `liste[i:n]` sont supérieurs à tous les éléments de `liste[0:i−1]`, alors au passage suivant le minimum de la liste `liste[i:n]` est placé en position d'indice i => Donc la liste `liste[0:i]` sera triée.
  L'invariant est vrai à l'initialisation et si il est vrai pour `i`, il sera vrai pour `i+1`. Donc l'invariant sera vrai à la fin de la boucle, soit pour `i=taille(liste)`.

  => Donc à la terminaison de l'algorithme, **toutes les valeurs de la liste seront triées.**

Complexité

- Si `n` est la taille de la liste nous avons deux boucles `pour` imbriquées, et pour chaque valeur de `i`, `j` prendra les valeurs `i+1`, `i+2`, `i+3`,... jusqu'à $n-1$ soit $n-i-1$ valeurs.
- Au total, cela nous fait : $(n-1)+(n-2)+...+2+1$ comparaisons.
- Un calcul mathématique nous donne $\frac{n(n+1)}{2}$ comparaisons soit donc de l'odre de n^2 opérations.

  => La complexité du tri par sélection est donc **quadratique.**

Application

- Implémentez l'algorithme du tri par sélection :

In [1]:

```python
# à compléter
def tri_selection(l):
    for i in range(len(l)-1):
        mini=i
        for j in range(i+1,len(l)):
            if l[j] < l[mini]:
                mini = j
        l[i],l[mini] = l[mini],l[i]
```

Attention, sauvegardez votre notebook avant de lancer une fonction qui traite un gros volume de données...

- Testez-le maintenant en faisant varier le nombre d'éléments de départ (allez doucement, le tri peut-être très long) :

In [2]:

```python
from timeit import default_timer as timer
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Démarrage du chrono
debut_chrono = timer()
# Tri de la liste
tri_selection(maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",duree,"secondes")
```

```
Les 2000 éléments de la liste ont été triés en 0.4295000000000009 secondes
```

- Essayez maintenant de comparer la durée en triant la liste avant, que remarquez-vous ?

In [3]:

```python
from timeit import default_timer as timer
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Tri de la liste
maliste.sort()
# Démarrage du chrono
debut_chrono = timer()
# Tri de la liste
tri_selection(maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",duree,"secondes")
```

```
Les 2000 éléments de la liste ont été triés en 0.4435000000000002 secondes
```

Saisissez votre réponse ici :

Le fait que la liste soit déjà triée n'impacte pas la performance du tri.

- Maintenant, comme nous l'avons fait dans le *TP 19bis* sur la dichotomie, comptez le nombre d'opérations du tri en fonction de la taille de la liste.
  La fonction `tri_selection2` devra retourner 2 entiers : `n` le nombre de comparaisons et `m` le nombre de permutations.

In [4]:

```python
# à compléter
def tri_selection2(l):
    n=0
    m=0
    for i in range(len(l)-1):
        mini=i
        for j in range(i+1,len(l)):
            n+=1
            if l[j] < l[mini]:
                m+=1
                mini = j
        l[i],l[mini] = l[mini],l[i]
    return n,m
```

In [5]:

```python
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Tri de la liste
(n,m)=tri_selection2(maliste)
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",n,"comparaisons et",m,"permutations" )
```

```
Les 2000 éléments de la liste ont été triés en 1999000 comparaisons et 12005 permutations
```

- Vérifiez maintenant l'influence du tri sur le nombre d'opérations :

In [6]:

```python
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
maliste.sort()
# Tri de la liste
(n,m)=tri_selection2(maliste)
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",n,"comparaisons et",m,"permutations" )
```

```
Les 2000 éléments de la liste ont été triés en 1999000 comparaisons et 0 permutations
```

Le tri par insertion

On parcourt la liste en insérant l'élément à sa place dans la première partie de la liste.

Présentation de l'algorithme

Le tri par insertion est celui que l'on utilise naturellement pour trier une poignée de cartes.
On parcourt tous les éléments , et on va l'**insérer** à sa place dans les éléments déjà triés qui le précèdent, d'où le nom de l'algorithme.
Regardons une vidéo explicative.

- Algorithme :

```
pour i allant de 1 à taille(liste)
    cle ← liste[i]
    j ← i − 1
    tant que j>=0 et liste[j]>cle
        liste[j+1] ← liste[j]
        j ← j-1
    liste[j+1] ← cle
```

Un exemple

```
12 -  5 - 11 - 20 -  7  (i=1) cle=5 et 5 < 12 donc on décale 12
   - 12 - 11 - 20 -  7        fin de boucle, on insère 5
 5 - 12 - 11 - 20 -  7
 5 - 12 - 11 - 20 -  7  (i=2) cle=11 et 11 < 12 donc on décale 12
 5 -    - 12 - 20 -  7        11 > 5 donc fin de boucle, on insère 11
 5 - 11 - 12 - 20 -  7
 5 - 11 - 12 - 20 -  7  (i=3) cle=20 et 20 > 12 donc fin de boucle, on ne fait rien
 5 - 11 - 12 - 20 -  7
 5 - 11 - 12 - 20 -  7  (i=4) cle=7 et 7 < 20 donc décale 20
 5 - 11 - 12 -    - 20        7 < 12 donc on décale 12
 5 - 11 -    - 12 - 20        7 < 11 donc on décale 11
 5 -    - 11 - 12 - 20        7 > 5 donc fin de boucle, on insère 7
 5 -  7 - 11 - 12 - 20
```

Terminaison et correction

- Nous avons une boucle `pour` donc le nombre de passages dans cette boucle est parfaitement déterminé, la boucle `tant que` contient la décrémentation de `j` qui conditionne sa sortie donc la boucle se termine également. L'algorithme est donc **fini**.
- L'invariant de boucle sera aussi dans ce cas : "*Pour chaque valeur de `i` traitée, les valeurs `liste[0:i−1]` seront déjà triées.*"
  L'invariant est vrai pour `liste[0]` puisqu'un seul élément est forcément trié...
  Après un passage pour indice égal à `i`, la liste `liste[0:i−1]` est triée, alors au passage suivant l'élément `cle` sera ajouté à la bonne place dans cette liste. => Donc la liste `liste[0:i]` sera triée.
  L'invariant est vrai à l'initialisation et si il est vrai pour `i`, il sera vrai pour `i+1`. Donc l'invariant sera vrai à la fin de la boucle, soit pour `i=taille(liste)`.

  => Donc à la terminaison de l'algorithme, **les valeurs `liste[0:taille(liste)]` seront triées.** (soit la liste complète)

Complexité

- Si `n` est la taille de la liste nous une première boucle `for` faisant varier `i`, et pour chaque valeur de `i`, `j` prendra les valeurs `i−1`, `i−2`, `i−3`,... jusqu'à `0` soit i valeurs.
- Au total, cela nous fait : $1+2+...+(n-1)+n$ comparaisons.
- Un calcul mathématique nous donne $\frac{n(n+1)}{2}$ comparaisons soit donc de l'odre de n^2 opérations.

  => La complexité du tri par insertion est donc également **quadratique.**

Application

- Implémentez l'algorithme du tri par insertion :

In [7]:

```python
# à compléter
def tri_insertion(l):
    for i in range(len(l)):
        cle=l[i]
        j=i-1
        while (j>=0) and l[j]>cle:
            l[j+1]=l[j]
            j-=1
        l[j+1]=cle
```

Attention, sauvegardez votre notebook avant de lancer une fonction qui traite un gros volume de données...

- Testez-le maintenant en faisant varier le nombre d'éléments de départ (allez doucement, le tri peut-être très long) :

In [8]:

```python
from timeit import default_timer as timer
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Démarrage du chrono
debut_chrono = timer()
# Tri de la liste
tri_insertion(maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",duree,"secondes")
```

```
Les 2000 éléments de la liste ont été triés en 0.4688999999999979 secondes
```

- Essayez maintenant de comparer la durée en triant la liste avant, que remarquez-vous ?

In [9]:

```python
from timeit import default_timer as timer
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Tri de la liste
maliste.sort()
# Démarrage du chrono
debut_chrono = timer()
# Tri de la liste
tri_insertion(maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",duree,"secondes")
```

```
Les 2000 éléments de la liste ont été triés en 0.0030000000000001137 secondes
```

Saisissez votre réponse ici :

Le fait que la liste soit déjà triée influe fortement sur la performance du tri

- Comptez maintenant le nombre d'opérations du tri en fonction de la taille de la liste.
  La fonction `tri_insertion2` devra retourner 2 entiers : `n` le nombre de comparaisons et `m` le nombre de décalages.

In [10]:

```python
# à compléter
def tri_insertion2(l):
    n=0
    m=0
    for i in range(len(l)):
        cle=l[i]
        j=i-1
        n+=1
        while (j>=0) and l[j]>cle:
            m+=1
            l[j+1]=l[j]
            j-=1
        l[j+1]=cle

    return n,m
```

In [11]:

```python
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
# Tri de la liste
(n,m)=tri_insertion2(maliste)
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",n,"comparaisons et",m,"permutations" )
```

```
Les 2000 éléments de la liste ont été triés en 2000 comparaisons et 970639 permutations
```

- Vérifiez maintenant l'influence du tri sur le nombre d'opérations :

In [12]:

```python
from random import randint

# Nombre d'éléments de la liste
nbelt=2000
# Génération de la liste
maliste=[randint(0,nbelt) for x in range(nbelt)]
maliste.sort()
# Tri de la liste
(n,m)=tri_insertion2(maliste)
# Affichage du résultat
print("Les",nbelt,"éléments de la liste ont été triés en",n,"comparaisons et",m,"permutations" )
```

```
Les 2000 éléments de la liste ont été triés en 2000 comparaisons et 0 permutations
```

Autres algorithmes

Il existe d'autres algorithmes de tri célèbres. Nous ne les étudierons pas en détail en classe de première mais leur découverte est intéressante.

Le tri à bulles

Le tri à bulles est un algorithme qui consiste à faire **remonter** progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.
Ce tri est peu performant et il n'est donc quasiment pas utilisé en pratique.

Le tri rapide ou « quick sort »

L'algorithme de tri rapide (« *quick sort* » en anglais) a été inventé dans les années 60. Il s'agit d'un algorithme de type **dichotomique**. Le principe est de séparer l'ensemble des données en deux parties. La séparation se fait par rapport à une valeur pivot, en deux ensembles de valeurs inférieures ou supérieures à la valeur pivot.
Les deux ensembles sont ensuite traités séparément de la même manière.

Comparatif des différents tris

Le site suivant donne un exemple de chacun des tris et montre un tres bon exemple de comparaison des algorithmes de tri les plus célèbres sur une liste : Algorithmes de tri