La notion de complexité

Complexité en temps

C'est le temps mis par un algorithme pour traiter un exemplaire.
C'est une fonction de la taille d'un exemplaire.

Complexité en espace

C'est l'espace mémoire utilisé par un algorithme pour traiter un exemplaire en plus de celui de l'exemplaire lui-même.
C'est une fonction de la taille d'un exemplaire.

On distinguera complexité dans le pire des cas, dans le meilleur des cas et en moyenne.

Lorsqu'on parle simplement de complexité on fait généralement référence à la complexité dans le pire des cas.

II Faisons le point

d'un point de vue pratique
- choisir l'opération à compter
- compter dans les boucles
- établir des équations de récurrence
- distinguer le pire des cas et le meilleur des cas

d'un point de vue théorique
- la notion de complexité

Analyse des algorithmes de tri

- les tris envisagés :
  - bulle
  - selection
  - insertion, avec différentes manières d'insérer
  - fusion
  - quicksort
- protocole de test :
  - sur des tableaux de taille 1 à 100
  - sur des tableaux croissants
  - sur des tableaux décroissants
  - sur des tableaux aléatoires : moyenne réalisée sur 1000 échantillons

Le tri bulle

- principe
- code :

```
1  def bubble_sort(t):
2    global nb_cmp
3    tt = copy.deepcopy(t)
4    n = len(t)
5    for i in range (2,n+1):
6      for j in range (0,n-i+1):
7        nb_cmp = nb_cmp + 1
8        if tt[j] > tt[j+1]:
9          # echange des valeurs aux positions j et j + 1
10         aux = tt[j]
11         tt[j] = tt[j+1]
12         tt[j+1] = aux
13   return tt
```

- estimation du nombre d'opérations de comparaison
- décompte exact

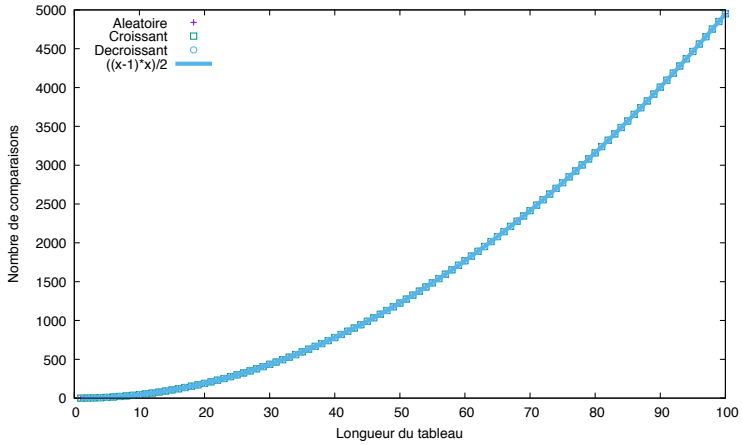
Détermination expérimentale

```
pour l allant de 1 à 100 faire
  creer un tableau croissant de taille l, le trier
  imprimer le nombre de comparaisons, raz du nb. comparaisons
  creer un tableau decroissant de taille l, le trier
  imprimer le nombre de comparaisons, raz du nb. comparaisons
  pour i allant de 1 à 1000 faire
    creer un tableau aleatoire de taille l, le trier
    sauvegarder le nombre de comparaisons
  fin pour
  imprimer le nombre de comparaisons
fin pour
```

```
...
10 45.000000 45.000000 45.000000
11 55.000000 55.000000 55.000000
12 66.000000 66.000000 66.000000
13 78.000000 78.000000 78.000000
...
```

```
gnuplot 'mydata.txt' using 1:2 title 'Croissant', \
        '' using 1:3 title 'Decroissant', \
        '' using 1:4 title 'Aleatoire'
```


Tri bulle - analyse



Le tri sélection

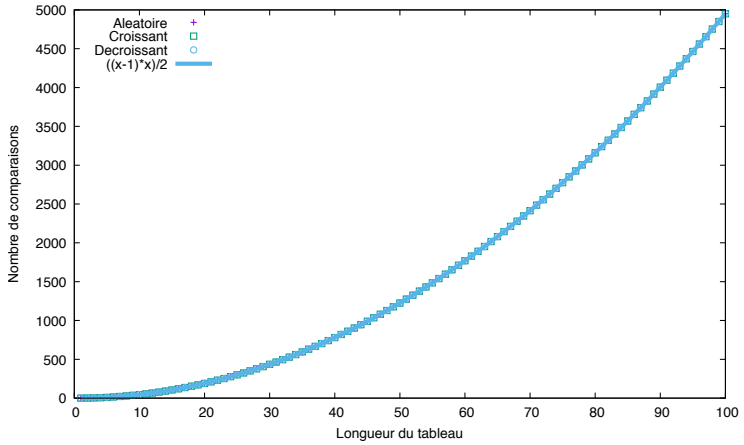
- principe, exemple
- code

```
1  def selection_sort(t):
2      global nb_cmp
3      tt = copy.deepcopy(t)
4      n = len(t)
5      for i in range (0,n-1):
6          # recherche du plus petit element dans tt
7          # entre les indices i+1 et n-1
8          index = index_minimum (tt,i,n-1)
9          if i != index:
10             # echange des valeurs aux positions j et j + 1
11             aux = tt[i]
12             tt[i] = tt[index]
13             tt[index] = aux
14     return tt
```

- estimation du nombre d'opérations de comparaison
- décompte exact


```
1  def index_minimum (t,a,b):
2      global nb_cmp
3      index = a
4      for i in range(a+1,b+1):
5          nb_cmp = nb_cmp + 1
6          if t[i] < t[index]:
7              index = i
8      return index
```

Tri sélection - analyse



Le tri insertion

- principe, exemple
- code

```
1  def insertion_sort(t):
2      tt = copy.deepcopy(t)
3      n = len(t)
4      for i in range (1,n):
5          # insere t[i] dans la tranche t[0:i]
6          insert(tt,i)
7      return tt
```

- décompte exact

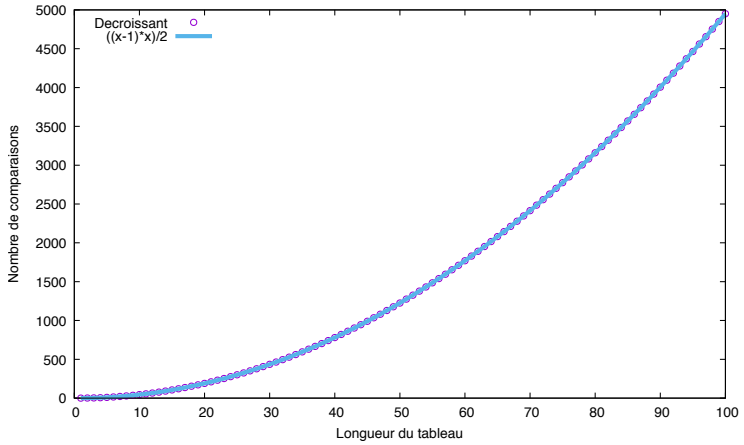

```
1  def insert(t,i):
2      global nb_cmp
3      k = i
4      aux = t[i]
5      # insere t[i] dans la tranche t[0:i]
6      while k >= 1 and aux <= t[k-1]:
7          nb_cmp = nb_cmp + 1
8          t[k] = t[k-1]
9          k = k - 1
10     if k >= 1:
11         nb_cmp = nb_cmp + 1
12     t[k] = aux
```

il ne faut rien oublier de compter

Pour éviter cet écueil : on surcharge l'opérateur de comparaison pour que le comptage se fasse à chaque appel à la fonction de comparaison

```
1  def cmp(a,b):
2      global nb_cmp
3      nb_cmp = nb_cmp + 1
4      return a <= b
5
6  def insert(t,i):
7      k = i
8      aux = t[i]
9      # insere t[i] dans la tranche t[0:i]
10     while k >= 1 and cmp(aux,t[k-1]):
11         t[k] = t[k-1]
12         k = k - 1
13     t[k] = aux
```

Tri insertion - analyse



II Faisons le point

la complexité est bien une fonction de la taille de la donnée

le meilleur des cas et le pire des cas, lorsqu'ils existent, ne sont pas définis par rapport à la taille de la donnée

≪ *pour n fixé, le meilleur des cas est . . .* ≫