cours_3

March 10, 2021

Cours 3 (Tris I)

Le cours du jour traite des points suivants:

- Définition du problème du tri
- Classification des algorithmes de tri
- Tri par sélection
- Tri par insertion
- Bonus

0.1 Introduction

0.1.1 Définition

On définit le problème du tri comme:

Entrée: une séquence de valeurs $a_1, a_2, ..., a_n$ comparables

Sortie: une **permutation** σ (un réarrangement) telle que la séquence

$$a_{\sigma(1)}, a_{\sigma(2)}, ..., a_{\sigma(n)}$$

soit **ordonnée** (croissant ou décroissant)

0.1.2 Clé de tri

On parle de **clé de tri** pour désigner l'élément sur lequel l'ordre est mis en place.

- Type de clé usuelles: nombre, chaine, liste
- Clé définie par l'utilisateur. Ex. pour une chaine
  - ordre lexicographique (par défaut)
  - nombre de caractères pour une chaine
  - nombre de consonnes puis nombre de voyelles
  - mois de l'année: 'Janvier' < 'Février'
- Clé simple dans une structure plus complexe.

  Ex.: Tuple (JJ, MM, YYYY)

Le langage de programmation fait donc des choix de clés par défaut.

Il revient au programmeur de les adapter si besoin est, et de définir sa propre de comparaison (voir Bonus).

0.2 Classification des algorithmes de tri

0.2.1 Tri en place

Un algorithme de tri est dit **en place** lorsque jamais plus d'un *nombre constant* d'éléments est stocké hors du tableau quelque soit sa taille.

0.2.2 Tri stable

Un algorithme de tri est dit **stable** lorsque l'ordre des éléments ayant une *clé identique* est maintenu.

Exemple:

- clé de tri : nombre de caractères pour une chaine
- T = ['cb', 'e', 'hi', 'A'] -> Tri stable -> ['e', 'A', 'cb', 'hi']
- T = ['cb', 'e', 'hi', 'A'] -> Tri instable -> ['A', 'e', 'cb', 'hi']

On peut rendre stable un algorithme de tri en introduisant une clé secondaire.

0.2.3 Tri par comparaison ou non

- Un algorithme de tri peut utiliser des comparaisons "<"
- ou non... (voir TP)

0.2.4 Tri incrémental

Un algorithme de tri est **incrémental** (ou en ligne) si il est capable de traiter les données une par une sans disposer du tableau en entier.

Pour nous, le tri sera incrémental si l'arrivée d'une donnée ne force pas à tout retrier.

0.3 Tri par sélection

0.3.1 L'algorithme

- Données : T un tableau à n valeurs comparables
- Résultat: T est trié par ordre croissant
- Principe: Pour un i allant de 1 à n-1 par ordre croissant:
  - Chercher le minimum de T[i..n] -> imin
  - Echanger T[imin] et T[i]

Algorithme 4: Tri par sélection

Données : T un tableau de n nombres
Résultat : Le tableau T est trié par ordre croissant

```
1 n ← longueur(T);
2 pour i ← 1 à n − 1 faire
3 |   imin ← i;
4 |   pour j ← i + 1 à n faire
5 |   |   si T[j] < T[imin] alors imin ← j;
6 |   fin
7 |   Échanger T[i] et T[imin];
8 fin
9 return T;
```

0.3.2 Caractéristiques: Tri par sélection

En place ? *etc* ?

Il est:

- en place
- stable
- par comparaison
- non incrémental

0.3.3 Analyse du Tri par sélection

- *Terminaison*: OK
- Correction de l'algorithme par invariant
- Complexité (asymptotique) de l'algorithme.

Preuve de correction par récurrence sur *i*:

- Pour i = 1, l'algorithme recherche le plus petit élément de T et le met en première position. => Le sous-tableau T[1..1] est trié
- Supposons T[1..i-1] trié.

  L'algorithme recherche le plus petit élément de T[i..n] et le met en position i.

  - T[i] <= à tous les éléments de T[i..n] (car min).
  - T[i] >= à tous les éléments de T[1..i-1] car sinon il aurait < qu'un des éléments de T[1..i-1] (impossible).
- On en conclut que T[1..i] est trié à la fin de chaque boucle sur i.
- A la fin, i = n-1 donc T[1..n-1] est trié et comme T[n] >= à tous les éléments de T[1..n-1], CQFD.

Efficience de l'algorithme

Nombre d'échanges ?

Pour i fixé, 1 échange.

Nombre total d'**échanges**: n-1

Nombre de comparaisons ?

Pour i fixé, n-i comparaisons.

- i = 1 -> n - 1 comparaisons
- i = 2 -> n - 2 comparaisons
- ...
- i = n - 2 -> 2 comparaisons
- i = n - 1 -> 1 comparaison

Nombre total de **comparaisons**:

$$1 + 2 + 3 + ... + (n-2) + (n-1) = \frac{n * (n-1)}{2} \sim n^2$$

0.3.4 Implémentation en Python

[1]:
```
def tri_selection(T):
    n = len(T)
    for i in range(0, n-1):
        imin = i
        for j in range(i+1, n):
            if T[j] < T[imin]:
                imin = j
        T[i], T[imin] = T[imin], T[i]
    return T
```

[2]:
```
def estTrie(L, deb=0, fin=None):
    if fin is None:
        fin = len(L)
    for i in range(1, fin):
        if L[i-1] > L[i] :
            return False
    return True
```

[3]:
```
# Vérification rapide qui n'est pas une preuve de correction
T = [4, 6, -4, 2, 5, 3]
T2 = tri_selection(T.copy())
print(f'{T2} est trié : {estTrie(T2)}')
```

```
[-4, 2, 3, 4, 5, 6] est trié : True
```

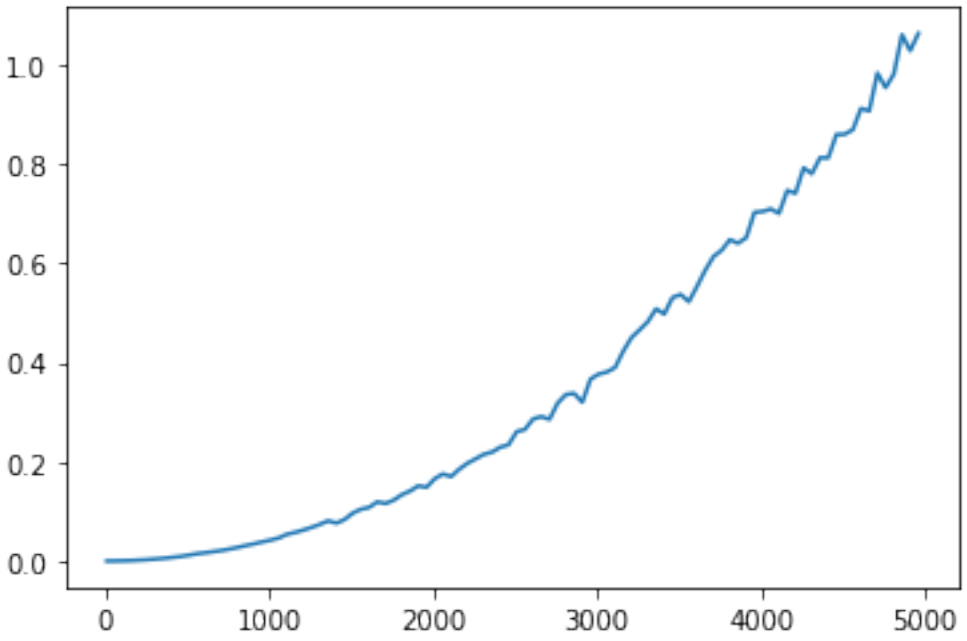
[4]:
```
import random

def tableau(n, min=0, max=100):
    return [random.randint(min, max) for _ in range(n)]
```

[5]:
```
import matplotlib.pyplot as plt
from time import time

tps_select = []
N = list(range(10, 5000, 50))
for n in N:
    T = tableau(n)
    debut = time()
    _ = tri_selection(T)
    tps_select.append(time() - debut)
```

[6]:
```
plt.plot(N, tps_select)
plt.show()
```



0.3.5 Tri par insertion

0.3.6 L'algorithme

- Données : T un tableau à n valeurs comparables
- Résultat: T est trié par ordre croissant
- Principe: Pour i allant de 2 à n par ordre croissant:

  Inserér T[i] dans le tableau T[1..i-1] trié en décalant vers la droite toutes les valeurs qui lui sont >

source: Tri par insertion (Wikipédia)

0.4 Tri par insertion

Algorithme 5: Tri par insertion

Données : T un tableau de n entiers
Résultat : Le tableau T est trié par ordre croissant

```
1  pour i ← 2 à n faire
2  |  clé ← T[i];
3  |  j ← i − 1;
4  |  tant que j > 0 et clé<T[j] faire
5  |  |  T[j + 1] ← T[j];
6  |  |  j ← j − 1;
7  |  fin
8  |  T[j + 1] ← clé;
9  fin
10 return T;
```

0.4.1 Caractéristiques du Tri par insertion

Il est:

- en place
- stable (car inf. strict)
- par comparaison
- incrémental

0.4.2 Analyse du Tri par insertion

- *Terminaison de l'algorithme*: OK
- *Correction de l'algorithme* par invariant:
  - T[1] Trié
  - T[1..i-1] est trié -> après -> T[1..i] trié
  - Donc T[1..n] trié à la fin.

Efficience de l'algorithme

Le nombre de comparaisons est proportionnel au nombre d'échanges.

- Pour i fixé, on distingue plusieurs cas, clé = T[i]:
  - 2 comparaisons si T[i-1] $\leq$ clé
  - 2*(i-1) comparaisons si clé < T[j] pour tout j=1..i-1.
  - i (= 2*(i/2)) comparaisons en moyenne

Nombre total de **comparaisons**:

- Cas favorable:

$$nbComp(n) \geq \sum_{i=2}^{n} 2 = 2 * \sum_{i=1}^{n-1} 1 = 2(n-1) \sim n$$

- Cas défavorable

$$nbComp(n) \leq \sum_{i=2}^{n} 2 * (i-1) = 2 * \sum_{i=1}^{n-1} i = n * (n-1) \sim n^2$$

0.4.3 Implémentation en Python

[16]:
```
def tri_insertion(T):
    n = len(T)
    for i in range(1, n):
        cle = T[i]
        j = i - 1
        while j >= 0 and cle < T[j]:
            T[j+1] = T[j]
            j = j - 1
        T[j+1] = cle
    return T
```

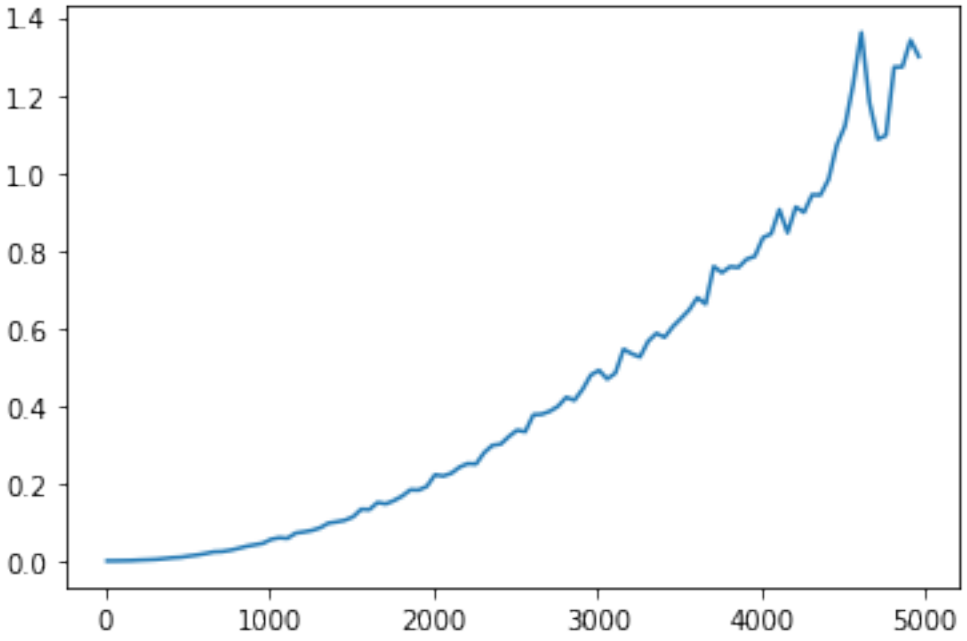
[17]:
```
# Vérification rapide qui n'est pas une preuve de correction
T = [4, 6, -4, 2, 5, 3]
T2 = tri_insertion(T)
print(f'{T} est trié : {estTrie(T)}')
```

```
[-4, 2, 3, 4, 5, 6] est trié : True
```

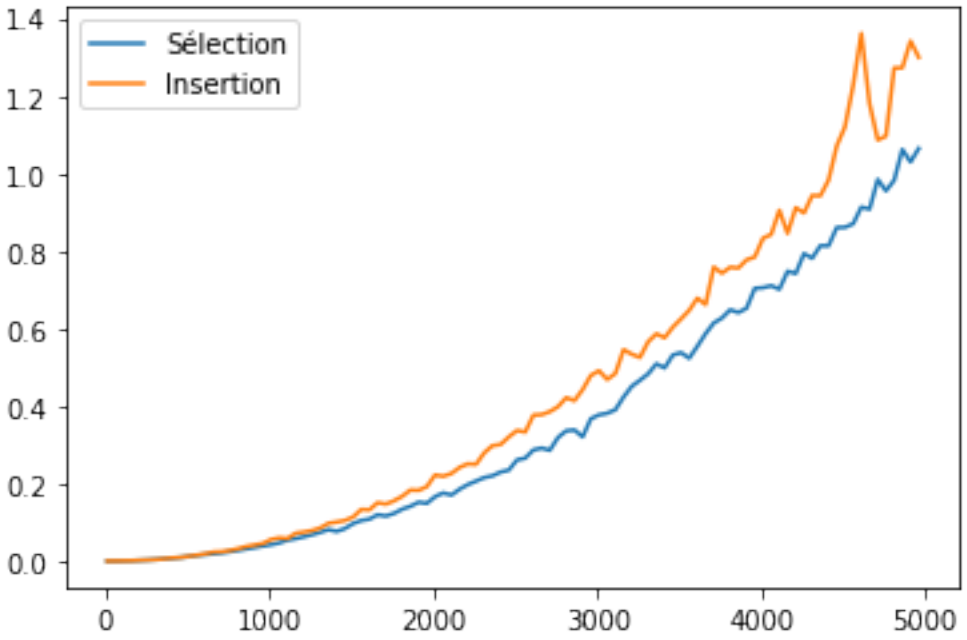
[9]:
```
import matplotlib.pyplot as plt
from time import time

tps_insertion = []
for n in N:
    T = tableau(n)
    debut = time()
    _ = tri_insertion(T)
    tps_insertion.append(time() - debut)
```

[10]:
```
plt.plot(N, tps_insertion)
plt.show()
```



[11]:
```
plt.plot(N, tps_select, label="Sélection")
plt.plot(N, tps_insertion, label="Insertion")
plt.legend()
plt.show()
```



[12]:
```
T = tableau(5000)
T2 = T[:]
T_trie = tri_selection(T)
T_trie_inv = T_trie[::-1]    # T_trie_inv = reversed(T_trie)
deb = time(); _ = tri_insertion(T_trie); print(f"Meilleur des cas: {time() -␣
 ↪deb}")
deb = time(); _ = tri_insertion(T2); print(f"Un cas: {time() - deb}")
deb = time(); _ = tri_insertion(T_trie_inv); print(f"Pire des cas: {time() -␣
 ↪deb}")
```

```
Meilleur des cas: 0.0011920928955078125
Un cas: 1.3179478645324707
Pire des cas: 2.343794822692871
```

0.4.4 Remarques finales

- Il existe une variante de ce tri avec des échanges à la place de la recopie.
- Peut on insérer plus efficacement ?

  La dichotomie !!

0.5 Bonus du jour

- Dictionnnaire en compréhension
- (optionnel) total_ordering dans functools

[13]:
```
import string

ch = string.ascii_lowercase
print(ch)
```

```
abcdefghijklmnopqrstuvwxyz
```

[14]:
```
L = [i for i in range(len(ch))]
print(L)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25]
```

[15]:
```
D = {k:v+1 for k, v in zip(ch, L)}
print(D)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j':
10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18, 's':
19, 't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26}
```

[ ]:
```

```

https://docs.python.org/3/library/functools.html

[ ]: