ALGORITHMES

Premiere SPE NSI

Bruno DARID

Table of contents

1. Algorithmique en 1re spé NSI

1.1 Les tris

Contenus	Capacités attendues	Commentaires
Tris par insertion, par sélection	Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.	La terminaison de ces algorithmes est à justifier. On montre que leur coût est quadratique dans le pire cas.

1.2 La recherche

2. Tri par sélection

2.1 Principe

On commence par rechercher le plus petit élement du tableau puis on l'échange avec le premier élement. Ensuite, on cherche le deuxième plus petit élement et on l'échange avec le deuxième élément du tableau et ainsi de suite jusqu'à ce que le tableau soit entièrement trié.

Voir l'animation proposée. lien

2.2 Algorithme et exemple d'implémentation en python

On peut formaliser l'algorithme du tri par sélection avec le pseudo-code suivant:

```
Tri_selection(t)
t: tableau de n éléments (t[0..n-1)
Pour i allant de 0 à n-2:
    idxmini = i
    Pour j allant de i+1 à n-1:
        Si t[j] < t[idxmini]:
            idxmini = j
    Echanger t[i] et t[idxmini]
```

Travail

- Appliquer cet algorithme à la main sur le tableau `t = [3, 4, 1, 7, 2]`.
- donner une implémentation possible en python de cet algorithme et tester.

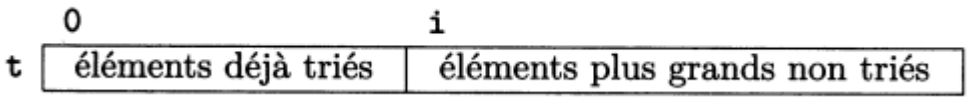
```
def echange(t, i, j):
    """
    Permute les éléments situés aux index i et j du tableau t
    t: tableau non vide
    i, j: entiers dans l'intervalle [0, len(t)-1]
    """
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def tri_selection(t):
    """
    trie par ordre croissant les éléments de t
    """
    n = len(t)
    #Compléter le code
```

```
# Test
t = [5, 6, 1, 1, 15, 0, 4]
tri_selection(t)
assert t == [0, 1, 1, 4, 5, 6, 15]
```

2.3 Validité de l'algorithme

La **terminaison** est assurée car l'algorithme fait intervenir deux boucles bornées (boucle `for`).

Par ailleurs, la situation au $i^{ème}$ tour de boucle peut être représentée de la manière suivante:



Tous les éléments d'indice compris entre 0 et $i-1$ inclus sont triés **et** ils sont tous inférieurs ou égaux aux éléments de la partie non triée, se trouvant entre i et $n-1$.
La preuve de cette proposition logique peut être délicate à établir en classe de 1re. Cette proposition est un **invariant** pour l'algorithme `Tri_selection`.

Définition

Un invariant de boucle est un *prédicat* (proposition logique) qui est:

- initialement vrai;
- vrai à l'entrée d'une itération ainsi qu'à la sortie de celle-ci

Vocabulaire

Le terme *correction* est à prendre ici au sens *correct*.

Trouver le *bon* invariant garantit que l'algorithme renvoie un résultat conforme aux spécifications et assure ainsi sa **correction partielle**. La combinaison de la correction partielle et de la terminaison permet de conclure à la **correction totale** de l'algorithme.

2.4 Complexité en temps

Le contenu de la boucle interne prend un temps d'exécution constant. Evaluons le nombre de fois qu'elle est exécutée.

Pour $i=0$, elle est exécutée $(n-1)-(0+1)+1=n-1$ fois.
Pour $i=1$, elle est exécutée $(n-1)-(1+1)+1=n-2$ fois.
Si on généralise, le nombre d'exécutions de la boucle interne est:

$$(n-1)+(n-2)+\cdots+2+1$$

Cette somme correspond à la *somme des termes consécutifs d'une suite arithmétique*, dont la valeur pour $n>1$ est donnée par:

$$\frac{n}{2}\times(n-1)=\frac{1}{2}n^2-\frac{1}{2}n$$

Pour une taille n très grande de l'entrée, le terme en n^2 devient prépondérant. Autrement dit, le nombre d'opérations effectuées, donc le temps d'exécution, est proportionnel à n^2.
La complexité du tri par sélection est quadratique.

Ce qu'il faut retenir

Le tri par sélection (du minimum) consiste à chercher le plus petit élément de la partie de tableau non triée et à le mettre à sa place définitive.
Ce problème est résolu habituellement par un algorithme faisant intervenir deux boucles bornées. La **terminaison** est donc assurée.
Un **invariant de boucle** permet de conclure à sa correction partielle. La conjugaison de ces deux propriétés assure la **correction totale** de l'algorithme proposé.
Cet algorithme a une **complexité temporelle quadratique**.

Application directe

En supposant que le tri par sélection prenne un temps directement proportionnel à n^2 et qu'un tri de 16000 valeurs nécessite 6.8 s. Calculer le temps nécessaire pour le tri d'un million de valeurs avec cet algorithme.

Exercice: temps d'exécution

Pour mesurer le temps d'exécution d'un programme, on importe la fonction `time` du module `time`. Cette fonction renvoie le temps en secondes écoulé depuis le 1^{er} janvier 1970.
Le code qui suit permet par exemple d'afficher le temps pris par l'exécution du tri d'un tableau.

```
from time import time
top = time()
tri_selection(t)
print(time() - top)
```

On souhaite comparer les temps d'exécution des tri sélection et insertion sur deux types de tableau: un tableau de nombre au hasard et un tableau de nombres déjà triés. On reprend le code des fonctions de tri du cours.

1. Construire un tableau de 3000 entiers pris au hasard entre 1 et 10000, bornes comprises. Mesurer le temps d'exécution du programme de tri sélection et de tri insertion pour trier ce tableau. *Attention: il faut reconstruire le tableau entre les deux tris*. Quel commentaire peut-on faire concernant les deux résultats ?
2. Construire un tableau de 3000 entiers de 0 à 2999, bornes comprises. Mesurer le temps d'exécution du programme de tri sélection et de tri insertion pour trier ce tableau. Quel commentaire peut-on faire concernant les deux résultats ?
3. Mesurer sur un tableau de 100000 entiers, choisis de manière aléatoire entre 1 et 100000, le temps d'exécution de la méthode `sort()` de python. Syntaxe: `t.sort()`. Commentez.

3. Tri par insertion

3.1 Principe

Visionner la séquence vidéo proposée. Lien

Le tri par insertion est le tri effectué par le joueur de carte. En supposant que l'on maintienne une partie triée, on décale les cartes de cette partie, de manière à placer la carte à classer (*voir video*).

En informatique, on va très souvent travailler avec un tableau et le **parcourir de la gauche vers la droite, en maintenant la partie déjà triée sur sa gauche** (voir lien wikipedia).
Concrètement, on va décaler d'une case vers la droite tous les éléments déjà triés, qui sont plus grands que l'élément à classer, puis déposer ce dernier dans la case libérée.

3.2 Algorithme

Notation

La notation `t[0..i-1]` désigne ici les i premiers éléments d'un tableau `t`, c'est-à-dire `t[0], t[1], ..., t[i-1]`.

```
Algorithme Tri_insertion(t)
---------------------------
t: tableau de n éléments comparables (t[0..n-1])

Pour i allant de 1 à n-1:
    amener t[i] à sa place parmi t[0..i-1]
```

3.3 Implémentation en python

On commence par donner une réalisation de `amener t[i] à sa place parmi t[0..i-1]` en écrivant une fonction `place(t, i)` qui amène l'élément d'index i à sa place parmi les éléments d'index 0 à $i-1$ déjà classés.

```
def place(t, i):
    """ amène t[i] à sa place dans t[0..i-1] supposé trié"""
    elt_a_classer = t[i]
    j = i
    # décalage des éléments du tableau à droite, pour trouver la place de t[i]
    while j > 0 and t[j - 1] > elt_a_classer:
        t[j] = t[j - 1]
        j = j - 1
    # on insère l'élément à sa place
    t[j] = elt_a_classer
```

Travail

Implémenter le tri par insertion en python et le tester.

```
def insertion(t):
    # compléter le code de la fonction insertion(t), sans oublier la spécification
    pass
```

```
# Test
t = [7, 2, -3, 5]
insertion(t)
assert t == [-3, 2, 5, 7]
```

3.4 Validité de l'algorithme

L'algorithme `Tri_insertion` **termine** car il présente une boucle bornée. La boucle conditionnelle présente dans la réalisation `amener t[i] à sa place parmi t[0..i-1]` termine également, la quantité *j* étant un **variant** de boucle.

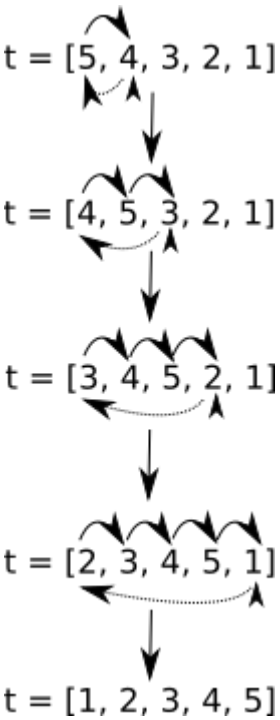
> **Invariant de boucle**
>
> A la i-ème itération, le sous tableau `t[0..i-1]` est trié.

De manière intuitive, on comprend qu'à chaque tour de boucle on se rapproche de la solution recherchée. On agrandit la zone triée de un élément. Exhiber une telle propriété (*un invariant de boucle*) permet de conclure à la correction partielle de l'algorithme.

La combinaison de la correction partielle avec la terminaison permet de conclure à la **correction totale de l'algorithme** `Tri_insertion`.

3.5 Efficacité: complexité temporelle de l'algorithme

Afin d'évaluer le coût de l'algorithme dans le pire des cas, on doit s'intéresser aux nombre d'opérations effectuées, qui est ici lié au nombre de décalage avant de trouver la place de l'élément à classer. Le pire des cas se produit lorsque le tableau est classé en sens inverse. Visualisons cela sur un tableau à 5 éléments, simple à trier: `t = [5, 4, 3, 2, 1]`.



Le nombre de décalage nécessaire est: $1 + 2 + 3 + 4 = 10$.

On généralise sans peine: dans le pire des cas, pour un tableau de taille `n`, il faudra effectuer:

décalages. Comme pour le tri par sélection, le coût (on dit aussi *complexité*) en temps du tri par insertion, dans le pire des cas, est **quadratique**. On dit aussi que la complexité est en $O(n^2)$.

> **Notation**
>
> La notation $O(n^2)$ se lit *grand O de n carré*

Ce qu'il faut retenir

Le tri par insertion consiste à maintenir une partie d'un tableau triée et à parcourir la partie non triée en mettant chaque élément rencontré à sa place définitive dans la partie triée.
Ce problème est résolu habituellement par un algorithme faisant intervenir une boucle bornée et une boucle conditionnelle. La **terminaison** de la boucle bornée est évidente et celle de la boucle conditionelle facile à montrer avec un **variant de boucle**.
L'**invariant de boucle** *A la i-ème itération, le sous tableau t[0..i-1] est trié*, permet de conclure à sa correction partielle. La conjugaison de ces deux propriétés assure la **correction totale** de l'algorithme proposé.
Cet algorithme a une **complexité temporelle quadratique**.

4. Les tris

```
import random
import time
import matplotlib.pyplot as plt


%matplotlib notebook

def echange(t, i, j):
    """
    Permute les éléments situés aux index i et j du tableau t
    t: tableau non vide
    i, j: entiers dans l'intervalle [0, len(t)-1]
    """
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def selection(t):
    """
    trie par ordre croissant les éléments de t
    """
    n = len(t)
    #Compléter le code
    for i in range(n-1):
        idxmini = i
        for j in range(i+1, n):
            if t[j] < t[idxmini]:
                idxmini = j
        echange(t, i, idxmini)

def place(t, i):
    """ amène t[i] à sa place dans t[0..i-1] supposé trié"""
    elt_a_classer = t[i]
    j = i
    # décalage des éléments du tableau à droite, pour trouver la place de t[i]
    while j > 0 and t[j - 1] > elt_a_classer:
        t[j] = t[j - 1]
        j = j - 1
    # on insère l'élément à sa place
    t[j] = elt_a_classer

def insertion(t):
    """
    trie par ordre croissant les éléments de t
    """
    for i in range(1, len(t)):
        place(t, i)
```

4.1 La complexité des algorithmes de tri

4.1.1 Mesure du temps d'exécution d'un algorithme

1. La fonction `randint` fait partie du module `random`. Que réalise cette fonction ? *Indication*: utiliser la fonction `help` de python.

2. Compléter la spécification de la fonction suivante

```
def alea(n):
    """
    A compléter
    """
    return [random.randint(0, 100) for i in range(n)]
```

1. Compléter la fonction `tempstri(n, algo)` qui prend en paramètre un entier naturel n correspondant à la taille d'un tableau et une fonction de tri `algo`. Cette fonction renvoie le temps mis par l'algorithme de tri `algo` pour trier un tableau de taille n. Tester votre fonction sur les deux tris vus en cours et dont les codes python sont disponible au début de ce sujet.
   Aide : la fonction `time.time()` du module `time` : donne l'heure en seconde. Appelez-la avant puis après de faire le tri du tableau.

```
def tempstri(n, algo):
    """"""
    t = alea(n)
    start = time.time()
    algo(t)
    stop = time.time()
    return stop - start

# TESTS
print(tempstri(1000, insertion))
print(tempstri(1000, selection))
```

0.05924797058105469
0.037672996520996094

4.1.2 Interprétation de la complexité d'un algorithme

1. On souhaite visualiser le temps d'exécution d'un algorithme en fonction de la taille du tableau. On construit d'abord un tableau `taille` qui contient les tailles des différents tableaux (de 1000 à 10000 par pas de 1000). On construit ensuite un tableau `temps` qui contient les temps mis par un algorithme `algo` pour trier un tableau dont la taille est un élément de `taille`. Donner le code qui permet de construire le tableau `temps`. Indications: utiliser la fonction précédente ainsi qu'une construction de tableau par compréhension.

```
def trace(algo):
    """
    trace le graphe du temps d'exécution d'un algorithme en fonction de la taille du tableau
    algo: nom d'une fonction
    """
    taille = list(range(1000,11000,1000)) #Un tableau qui contient les tailles de tableau
    # Compléter la ligne suivante
    temps = [tempstri(n, algo) for n in taille]
    # Tracé - ne rien modifier ici
    fig = plt.figure()
    plt.title("Complexite tri par " + algo.__name__)
    plt.plot(taille,temps)#On trace la figure les tailles en abscisse,
    #les temps en ordonnées
    plt.xlabel('Taille tableau')
    plt.ylabel('Temps')
    plt.grid()
    plt.show()
```

5. Tracer le graphique donnant le temps d'exécution si on utilise un tri par insertion puis par sélection.

```
def alea(n):
    """
    A compléter
    """
    return [random.randint(0, 100) for i in range(n)]
```

```
def tempstri(n, algo):
    """"""
    t = alea(n)
    start = time.time()
    algo(t)
    stop = time.time()
    return stop - start
```

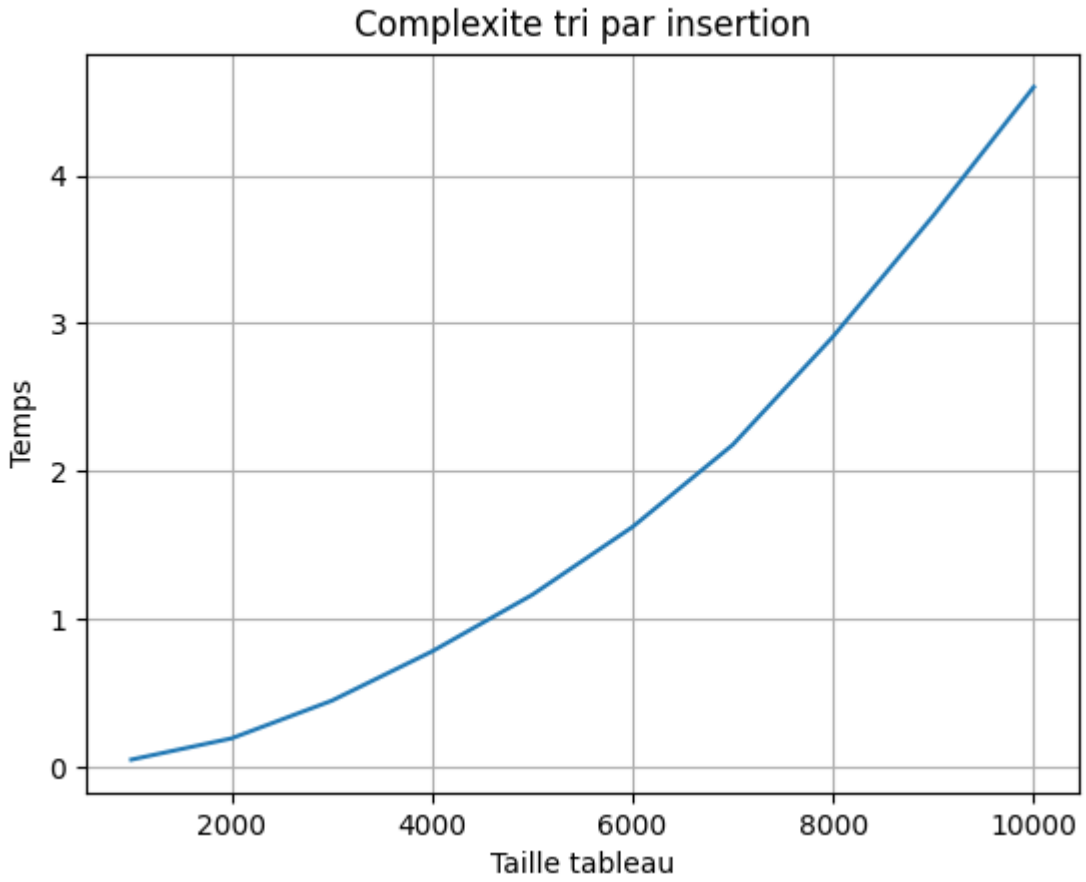
```
# TESTS
print(tempstri(1000, insertion))
print(tempstri(1000, selection))
```

```
0.05924797058105469
0.037672996520996094
```

```
 def trace(algo):
     """
     trace le graphe du temps d'exécution d'un algorithme en fonction de la taille du tableau
     algo: nom d'une fonction
     """
     taille = list(range(1000,11000,1000)) #Un tableau qui contient les tailles de tableau
     # Compléter la ligne suivante
     temps = [tempstri(n, algo) for n in taille]
     # Tracé - ne rien modifier ici
     fig = plt.figure()
     plt.title("Complexite tri par " + algo.__name__)
     plt.plot(taille,temps)#On trace la figure les tailles en abscisse,
     #les temps en ordonnées
     plt.xlabel('Taille tableau')
     plt.ylabel('Temps')
     plt.grid()
     plt.show()
```

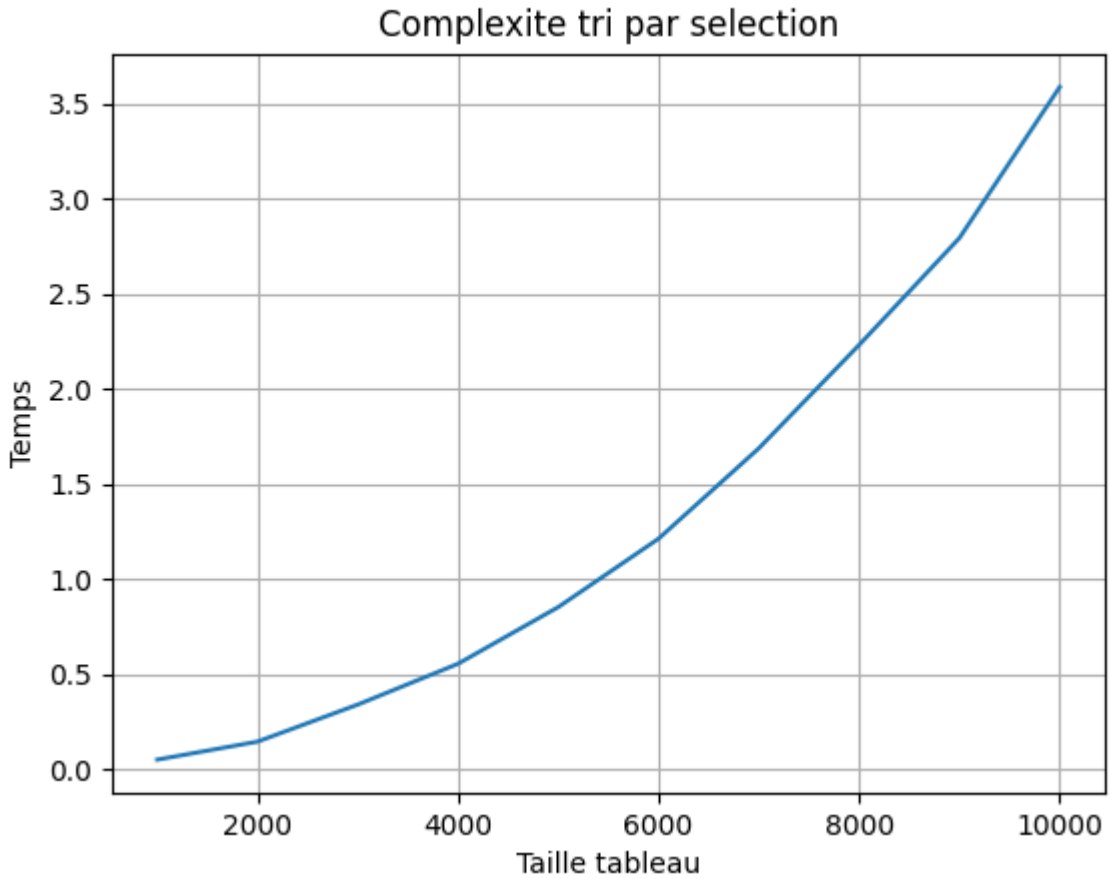
```
# Tri par insertion - compléter
trace(insertion)
```

```
<IPython.core.display.Javascript object>
```




```
# Tri par sélection - Compléter
trace(selection)
```

```
<IPython.core.display.Javascript object>
```



1. Justifier que la complexité de ces tri est quadratique. On pourra par exemple comparer les temps d'exécution pour trier un tableau de 2000 éléments puis de 6000 ou 10000 éléments.

```
1
```