

# ALGORITHMES

---

Premiere SPE NSI

*Bruno DARID*

*CC BY NC SA*

## Table of contents

---

1. Algorithmique en 1re spé NSI	3
1.1 La recherche	3
1.2 Les tris	3
2. ALGORITHMES DE RECHERCHE	4
2.1 Recherche séquentielle	4
2.2 Recherche dichotomique	7
3. Tri par sélection	9
3.1 Principe	9
3.2 Algorithme et exemple d'implémentation en python	9
3.3 Validité de l'algorithme	9
3.4 Complexité en temps	10
4. Tri par insertion	12
4.1 Principe	12
4.2 Algorithme	12
4.3 Implémentation en python	12
4.4 Validité de l'algorithme	13
4.5 Efficacité: complexité temporelle de l'algorithme	13
5. Exercices sur les tris	15
5.1 Rappel des fonctions du cours	15
5.2 La complexité des algorithmes de tri	16
5.3 Trié par ordre croissant ?	17
5.4 Tri sans modification du tableau d'entrée	17
5.5 Valeur plus fréquente ***	17

# 1. Algorithmique en 1re spé NSI

---

## 1.1 La recherche

---

Contenus	Capacités attendues	Commentaires
Parcours séquentiel d'un tableau	Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque. Écrire un algorithme de calcul d'une moyenne.	On montre que le coût est linéaire.
Recherche dichotomique dans un tableau trié	Montrer la terminaison de la recherche dichotomique à l'aide d'un variant de boucle.	Des assertions peuvent être utilisées. La preuve de la correction peut être présentée par le professeur.

## 1.2 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.

## 2. ALGORITHMES DE RECHERCHE

### 2.1 Recherche séquentielle

#### 2.1.1 Introduction

Pour *Thomas Cormen*, très grand professeur d'informatique, un algorithme est **une procédure de calcul, bien définie, qui prend en entrée une valeur ou un ensemble de valeurs, et qui donne en sortie une valeur ou un ensemble de valeurs**. C'est un outil permettant de résoudre un problème de calcul bien spécifié.

Un algorithme doit se terminer dans un temps raisonnable et doit résoudre correctement le problème donné. Ces deux propriétés sont respectivement nommées:

- **complexité** (*en temps*)
- **correction**

Cette première leçon d'algorithmique sera orientée vers la recherche (*séquentielle*) d'un élément dans un tableau.

#### 2.1.2 Parcours séquentiel d'un tableau

##### Recherche d'une occurrence

Soit le problème suivant: étant donné un tableau  $t$  de  $n$  éléments, écrire un algorithme qui permet de dire si un élément  $x$  appartient à ce tableau.

**Le problème:**  $x$  appartient-il au tableau ?

**Les entrées:** un tableau  $t$  et un élément  $x$ ;

**La sortie:** une réponse vraie ou fausse ( `True` ou `False` )

Une solution:

```

1  -----
2  Algorithme appartient(x, t)
3  -----
4  Entrées:
5  - t: tableau
6  - x: élément de même type que les éléments du tableau t
7  Sortie: booléen (réponse)
8
9  Variable: n, entier naturel
10
11 n = taille(t)
12 Pour i allant de 0 à n-1
13     Si t[i] = x
14         Renvoyer Vrai
15 Renvoyer Faux

```

```

1  #Implémentation en python
2  def appartient_v1(x, t):
3      """
4      renvoie un booléen correspondant à la présence ou l'absence de x dans t;
5      t: tableau
6      x: élément de même type que les éléments de t
7      """
8      n = len(t)
9      for i in range(n):
10         if t[i] == x:
11             return True
12     return False

```

```
1  appartient_v1(-5, [5, 10, 15, 20])
```

```
1  False
```

## Travail

Réaliser l'implémentation en python en utilisant une boucle `while` plutôt qu'une boucle `for`.

```

1 def appartient_v1b(x, t):
2     """
3     renvoie un booléen correspondant à la présence ou l'absence de x dans t;
4     t: tableau
5     x: élément de même type que les éléments de t
6     """
7     pass
8 # TESTS -
9 appartient_v1b(-1, [5, 10, 15, 20])

```

## Remarque

On peut boucler sur éléments plutôt que les indices; cela nous dispense de la variable `n`.

```

1 def appartient_v2(x, t):
2     """
3     renvoie un booléen correspondant à la présence ou l'absence de x dans t;
4     t: tableau
5     x: élément de même type que les éléments de t
6     """
7     for elt in t:
8         if elt == x:
9             return True
10    return False

```

## Recherche d'un extrémum

Deuxième problème: étant donné un tableau `t` de  $n$  éléments (des entiers pour se fixer les idées). Le deuxième algorithme proposé permet de trouver le minimum parmi les éléments de ce tableau.

**Le problème:** quelle est la plus petite valeur du tableau ?

```

1 -----
2 Algorithme minimum(t)
3 -----
4 Entrée: un tableau t;
5 Sortie: le plus petit élément de t
6 Variables:
7     n, entier naturel
8     mini, même type que les éléments de t
9 Précondition: t non vide
10
11 n = taille(t)
12 mini = t[0]
13 Pour i allant de 1 à n-1:
14     Si t[i] < mini
15         mini = t[i]
16 Renvoyer mini

```

## Implémentation en python

```

1 def minimum(t):
2     """
3     Renvoie le plus petit élément de t
4     """
5     assert t != [], "Erreur: tableau vide"
6     n = len(t)
7     mini = t[0]
8     for i in range(1,n):
9         if t[i] < mini:
10            mini = t[i]
11    return mini
12
13 # Tests
14 minimum([5, 10, -15, 20])
15 minimum([0, 12])

```

**i Travail**

Ecrire et tester l'implémentation d'une fonction retournant l'élément **le plus grand** de  $t$

**Calcul d'une moyenne**

Troisième problème: étant donné un tableau  $t$  de  $n$  éléments que l'on supposera entiers. L'algorithme proposé permet de trouver la moyenne des éléments de ce tableau.

**Le problème:** quelle est la valeur moyenne des éléments du tableau ?

```

1  -----
2  Algorithme moyenne(t)
3  -----
4  Entrée: un tableau t de flottants;
5  Sortie: la moyenne de tous les éléments de t
6  Variables:
7      n, entier naturel
8      somme, flottant
9  Précondition: t non vide
10
11 n = taille(t)
12 somme = 0
13 Pour i allant de 0 à n-1:
14     somme = somme + t[i]
15 Renvoyer somme / n

```

**i Travail**

Implémenter cet algorithme en python

```

1  def moyenne(t):
2      """
3      Renvoie la moyenne des éléments d'un tableau t non vide;
4      """
5      pass
6
7  # Tests
8  moyenne([5.5, 10.0, 15.0, 19.5])

```

## 2.1.3 Efficacité des algorithmes

En informatique, la question de la performance des algorithmes est centrale. D'une manière générale, le traitement d'une certaine quantité de données requiert un temps d'exécution lié à cette quantité.

Ainsi dans nos exemples où  $n$  est la taille du tableau d'entrée, la boucle est parcourue:

- $n$  fois dans le pire des cas (*si  $x$  n'appartient pas au tableau*) dans le problème 1
- $n - 1$  fois dans le problème 2;
- $n$  fois dans le problème 3;

Le temps d'exécution sera lié au nombre d'opérations effectuées, qui dépend ici de la taille de l'entrée  $n$ , plus exactement **proportionnel** à  $n$  (ou  $n - 1$ ). On dit que **la complexité en temps** de ces algorithmes est **linéaire**.

**⚠ Remarque**

Il n'est pas question de rechercher de manière exacte le temps d'exécution mais plutôt d'avoir une estimation.

## 2.2 Recherche dichotomique

### 2.2.1 Principe

On souhaite rechercher une valeur  $x$  dans un tableau  $t$  trié (*par ordre croissant pour se fixer les idées*). L'idée principale est de comparer  $x$  avec la valeur située au milieu du tableau:

```

1   g           m           d
2   +-----+-----+-----+
3   |   ....   |t[m]|   ....   |
4   +-----+-----+-----+
```

- si elle est plus petite, il suffit de restreindre la recherche dans la partie gauche du tableau;
- sinon, on la restreint à la partie droite de  $t$ .
- ou alors on a trouvé  $x$

En répétant ce procédé on divise la zone de recherche par deux à chaque fois. On peut alors représenter la situation avec le schéma ci-dessous. Il s'agit d'une application du principe *diviser pour régner* qui sera vu en terminale.

```

1   0           g           d           n-1
2   +-----+-----+-----+
3   | éléments < x |   ....   | éléments > x |
4   +-----+-----+-----+
```

### 2.2.2 Algorithme

```

1   -----
2   Algorithme dichotomie(x, t)
3   -----
4   Entrées:
5       tableau t de taille n;
6       x élément à rechercher, de même type que les éléments de t
7   Renvoie index de x ou rien si non trouvé.
8   Pré-condition: t trié
9
10  Variables: g, d et m des index repérant les "bords" du tableau et le milieu respectivement
11
12  Fixer g à 0 et d à n-1
13  Tant que g <= d
14      calculer index m du milieu du tableau [g..d]
15      Si t[m] = x
16          Renvoyer m
17      Sinon si x > t[m]
18          g = m + 1
19      Sinon si x < t[m]
20          d = m - 1
21  Renvoyer non trouvé
```

### 2.2.3 Implémentation en python

```

1   # Fonction à utiliser pour vérifier qu'un tableau est trié
2   def est_trie(t):
3       """
4       Vérifie si un tableau t est trié et renvoie un booléen comme réponse
5       """
6       return all(t[i] <= t[i + 1] for i in range(len(t) - 1))
```

## Travail

Implémenter l'algorithme de la recherche dichotomique en python, en complétant le code ci-dessous.

```

1 # RECHERCHE DICHOtomIQUE
2
3 def dichotomie(x, t):
4     """
5     Renvoie la position de x dans le tableau t trié; None si non trouvé
6     Précondition: t trié
7     """
8     assert est_trie(t), "Erreur: tableau non trié"
9
10    g = 0 # index gauche
11    d = len(t) - 1 # index droit
12    # A compléter

```

## 2.2.4 Terminaison de l'algorithme

Lorsqu'un programme contient une boucle `while` il est susceptible de **diverger** (on dit aussi qu'il peut entrer dans une boucle *infinie*). Pour montrer l'arrêt de la boucle, on utilise une technique de raisonnement appelé **technique du variant de boucle**. Il s'agit de trouver parmi les éléments du programme une quantité:

- entière;
- positive;
- qui décroît strictement à chaque tour de boucle.

Si on peut exhiber un variant de boucle, l'algorithme termine.

	g	d	m	t[m]	d-g	trouvé ?
Initialisation						
Fin itération 1						
Fin itération 2						
Fin itération 3						

## Application

Soit `t = [1,7,8,9,12,15,15,22,30,31]`. On recherche `x = 15` dans `t`. Compléter le tableau ci-dessus. Conclure

## 3. Tri par sélection

### 3.1 Principe

On commence par rechercher le plus petit élément du tableau puis on l'échange avec le premier élément. Ensuite, on cherche le deuxième plus petit élément 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](#)

### 3.2 Algorithme et exemple d'implémentation en python

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

```

1  Tri_selection(t)
2  t: tableau de n éléments (t[0..n-1])
3  Pour i allant de 0 à n-2:
4      idxmini = i
5      Pour j allant de i+1 à n-1:
6          Si t[j] < t[idxmini]:
7              idxmini = j
8      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.

```

1  def echange(t, i, j):
2      """
3      Permute les éléments situés aux index i et j du tableau t
4      t: tableau non vide
5      i, j: entiers dans l'intervalle [0, len(t)-1]
6      """
7      tmp = t[i]
8      t[i] = t[j]
9      t[j] = tmp
10
11  def tri_selection(t):
12      """
13      trie par ordre croissant les éléments de t
14      """
15      n = len(t)
16      #Compléter le code
```

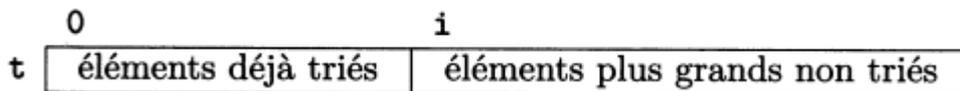
```

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

### 3.3 Validité de l'algorithme

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

Par ailleurs, la situation au  $i^{\text{ème}}$  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.

## 3.4 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:

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

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<sup>er</sup> janvier 1970.

Le code qui suit permet par exemple d'afficher le temps pris par l'exécution du tri d'un tableau.

```
1 from time import time
2 top = time()
3 tri_selection(t)
4 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.

## 4. Tri par insertion

### 4.1 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 vidéo*).

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.

### 4.2 Algorithme

#### Notation

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

```

1  Algorithme Tri_insertion(t)
2  -----
3  t: tableau de n éléments comparables (t[0..n-1])
4
5  Pour i allant de 1 à n-1:
6      amener t[i] à sa place parmi t[0..i-1]
```

### 4.3 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.

```

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

#### Travail

Implémenter le tri par insertion en python et le tester.

```

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

```

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

## 4.4 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

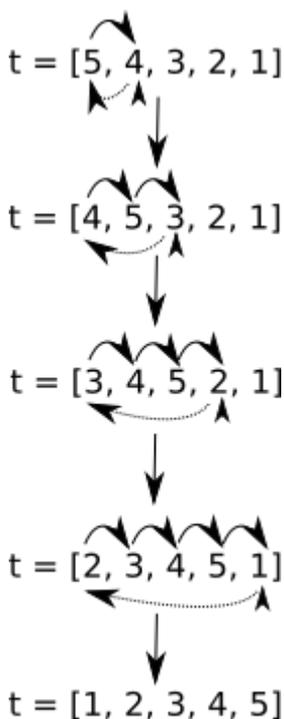
À la  $i$ -ème itération, le sous-tableau `t[0..i-1]` est trié.

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`.

## 4.5 Efficacité: complexité temporelle de l'algorithme

Afin d'évaluer le coût de l'algorithme dans le pire des cas, on doit s'intéresser au 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 conditionnelle 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**.

---

## 5. Exercices sur les tris

---

### 5.1 Rappel des fonctions du cours

---

```
1 import random
2 import time
3 import matplotlib.pyplot as plt
4
5
6 %matplotlib notebook
7
8 def echange(t, i, j):
9     """
10    Permute les éléments situés aux index i et j du tableau t
11    t: tableau non vide
12    i, j: entiers dans l'intervalle [0, len(t)-1]
13    """
14    tmp = t[i]
15    t[i] = t[j]
16    t[j] = tmp
17
18 def selection(t):
19     """
20    trie par ordre croissant les éléments de t
21    """
22    n = len(t)
23    #Compléter le code
24    for i in range(n-1):
25        idxmini = i
26        for j in range(i+1, n):
27            if t[j] < t[idxmini]:
28                idxmini = j
29        echange(t, i, idxmini)
30
31 def place(t, i):
32     """ amène t[i] à sa place dans t[0..i-1] supposé trié"""
33     elt_a_classifier = t[i]
34     j = i
35     # décalage des éléments du tableau à droite, pour trouver la place de t[i]
36     while j > 0 and t[j - 1] > elt_a_classifier:
37         t[j] = t[j - 1]
38         j = j - 1
39     # on insère l'élément à sa place
40     t[j] = elt_a_classifier
41
42 def insertion(t):
43     """
44    trie par ordre croissant les éléments de t
45    """
46     for i in range(1, len(t)):
47         place(t, i)
```

## 5.2 La complexité des algorithmes de tri

### 5.2.1 Mesure du temps d'exécution d'un algorithme

#### Indications pour cette partie

La fonction `help` de python permet d'obtenir de l'aide sur une commande/fonction.

La fonction `time.time()` du module `time` donne l'heure en seconde. En l'appelant avant puis après une fonction, on a une estimation du temps d'exécution de celle-ci.

Une fonction peut être passée en paramètre d'une autre fonction. Par exemple si `f` est une fonction, l'appel `ma_fonction(n, f)` est valide en python.

1. La fonction `randint` est présente dans le module `random`. Que réalise cette fonction ?
2. Compléter la spécification de la fonction suivante

```

1 def alea(n):
2     """
3     A compléter
4     """
5     return [random.randint(0, 100) for i in range(n)]

```

3. 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.

```

1 def tempstri(n, algo):
2     """
3     Renvoie le temps mis par un algorithme pour trier un tableau de taille n
4     n: entier naturel
5     algo: fonction de tri
6     """
7     t = alea(n)
8     start = time.time()
9     # A compléter
10
11 # TESTS - exemples
12 print(tempstri(1000, insertion))
13 print(tempstri(1000, selection))
14 0.05924797058105469
15 0.037672996520996094

```

## 5.2.2 Interprétation de la complexité d'un algorithme

### Indications pour cette partie

Pour les constructions de tableaux, penser aux constructions par compréhension.

Utiliser la fonction `tempstri()`

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`.

```

1 def trace(algo):
2     """
3     trace Le graphe du temps d'exécution d'un algorithme en fonction de la taille du tableau
4     algo: nom d'une fonction
5     """
6     taille = list(range(1000,11000,1000)) #Un tableau qui contient les tailles de tableau
7     # Compléter la ligne suivante
8     temps = [...]
9     # Tracé - ne rien modifier ici
10    fig = plt.figure()
11    plt.title("Complexite tri par " + algo.__name__)
12    plt.plot(taille,temps)#On trace la figure les tailles en abscisse,
13    #les temps en ordonnées
14    plt.xlabel('Taille tableau')
15    plt.ylabel('Temps')
16    plt.grid()
17    plt.show()

```

2. Tracer le graphique donnant le temps d'exécution si on utilise un tri par insertion puis par sélection.
3. 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.

## 5.3 Trié par ordre croissant ?

Ecrire une fonction `est_trie(t)` qui renvoie `True` si le tableau `t` est trié dans l'ordre croissant et `False` sinon.

## 5.4 Tri sans modification du tableau d'entrée

Les implémentations rencontrées jusqu'à présent effectuaient le *tri en place*, c'est-à-dire modifiaient le tableau à trier.

On demande ici d'écrire une fonction `insert_non_mutable(t)` qui réalise un tri par insertion du tableau `t` mais qui **renvoie le résultat dans un nouveau tableau**, laissant `t` inchangé. On modifiera légèrement la fonction `place(t, i)` pour résoudre ce problème.

## 5.5 Valeur plus fréquente \*\*\*

En utilisant un tri par insertion, écrire ne fonction `plus_frequente(t)` qui prend en paramètre un tableau d'entiers et qui renvoie la valeur la plus fréquente rencontrée dans le tableau. On peut remarquer que dans un tableau trié, des valeurs égales se retrouvent *cote à cote*.