Projet complexité - TP Tris

Version initiale le 18 février 2021. Dernière mise à jour le 19 avril 2021

Auteurs : Sébastien LOZANO - Fabien SERENKO

Sommaire

I Les consignes pour le projet

I.1 Sujet

Le projet consiste à construire un sujet de projet visant à mesurer empiriquement les temps de calcul d'algorithmes de tri, et de comparer ces temps de calcul avec les résultats théoriques.

I.2 Attendus

- ⇝ Rédiger une fiche de projet qui demande à l'élève les tâches suivantes :
  - ⩾ Programmer les algorithmes de tri : tri bulle, tri sélection, tri rapide (non récursif).
  - ⩾ Rappeler pour chaque tri quelle est la complexité dans le pire cas, dans quel cas, pour chaque tri, la tâche est difficile ou facile.
  - ⩾ Produire des jeux de tests visant à vérifier la correction de l'implantation.
  - ⩾ Tester chaque tri sur des tableaux d'entiers de taille variable, et remplis aléatoirement, et mesurer à chaque fois le temps de calcul, et le nombre d'actions atomiques (affectation, comparaison). Il faudra aussi tester le tri rapide natif de Python. Il y a donc 4 tris à comparer.
  - ⩾ Proposer une représentation graphique des résultats.
  - ⩾ Discuter de la correspondance ou non entre les résultats empiriques et les résultats théoriques.
- ⇝ Rédiger un document (1 page) décrivant les difficultés que vous voyez a priori dans un tel sujet, pour les élèves, et listant les apports pédagogiques que vous devrez faire pour accompagner les élèves (par exemple, comment mesurer le temps de calcul d'une exécution d'une fonction, comment « bien » faire un graphique (peut-être en collaboration avec le professeur de physique ou mathématique), etc.).
- ⇝ Rédiger un document (1 page) décrivant les critères d'évaluation du projet que vous pourriez prévoir ; une soutenance de projet est-elle souhaitable ? Un rapport est-il souhaitable ? Avec quel contenu ? Cette partie devra aborder quels jeux de test vous attendez, et comment vous évaluez ces jeux de test (pertinence, élaboration pratique, etc.)
- ⇝ Rédiger un document (1 page) critiquant le sujet du projet.
  - ⩾ Le sujet entre-t-il dans le cadre des programmes ? Fait-il appel à des compétences d'autres disciplines ?
  - ⩾ Évaluation du nombre de séances nécessaires pour accompagner les élèves.
  - ⩾ Évaluation de sa difficulté, de la taille du groupe pouvant aborder un tel travail.
  - ⩾ Évaluation de son intérêt pédagogique.

I.3 Points de vigilance

- ⇝ Les structures de données en Python (ou autre langage) sont très optimisées, et cette optimisation peut mener à des résultats empiriques surprenants. Comment montrer ceci, sans « casser » le discours sur la théorie de la complexité ?
- ⇝ C'est à vous de rédiger la fiche de projet de manière à accompagner pédagogiquement les élèves, avec le niveau que vous leur connaissez, ou que vous pensez connaître d'eux en informatique, en autonomie.
- ⇝ Vous n'avez pas vous-mêmes à « faire le projet » (même si dans une situation réelle, il faudrait le faire afin de prévenir toute surprise).

II La séquence

Cette séquence s'adresse à des élèves de $1^{ère}$ NSI au troisième trimestre.

II.1 Prérequis

Avant d'aborder cette séquence, les élèves doivent :

- ⇝ Savoir programmer en Python, notamment :
  - ⩾ Traiter des données en tables.
  - ⩾ Réaliser des fonctions.
  - ⩾ Importer et utiliser des modules.
  - ⩾ Importer et utiliser des bibliothèques.
- ⇝ Avoir des notions concernant les tris, notamment :
  - ⩾ Qu'entend-on par trier une liste ?
  - ⩾ Quels types de structures et de données en python peut-on trier ?
- ⇝ Avoir des notions concernant la complexité temporelle, notamment :
  - ⩾ Que le temps d'exécution dépend de la taille des données à traiter
  - ⩾ De ce que sont les complexités , $O(\sqrt{n})$, $O(log(n))$, $O(n)$, $O(n.log(n))$, $O(n)^2$, ...
- ⇝ Savoir ce que sont des préconditions, des postconditions
- ⇝ Savoir ce qu'est une assertion et la programmer.

Durée :

3 séances de 2 heures en classe associées à un peu de travail à la maison.

Déroulé

- ⇝ Les élèves de bon niveau pourront avancer à leur rythme en suivant les instructions des documents écrits et quelques informations données par le professeur ;
- ⇝ Les élèves moyens, ou faibles, seront guidés en classes par l'enseignant pour la très grande majorité des travaux à effectuer, et réaliseront avec l'enseignant une exploitation commune des résultats. Des outils pourront être fournis afin que chacun aboutisse à des résultats concrets.

II.2 Descriptif global

Introduction :

La séquence vise à :

- ⇝ Mesurer empiriquement les temps de calcul d'algorithmes de tri
- ⇝ Comparer ces temps obtenus empiriquement avec les résultats théoriques.

Objectifs

- ⇝ Comprendre un cahier des charges ;
- ⇝ Préparer une ou des fonctions de test en vue de s'assurer du respect d'un cahier des charges d'une future fonction ou procédure à programmer ;
- ⇝ Programmer des assertions pour tester des préconditions ;
- ⇝ Traduire un pseudo code pouvant être complexe en Python.
- ⇝ Visualiser et "ressentir" la complexité en temps de différents algorithmes ;
- ⇝ Voir, par la complexité, l'intérêt d'optimiser un algorithme.

Compétences travaillées

- ⇝ Décomposer un problème en sous-problèmes, reconnaître des situations déjà analysées et réutiliser des solutions.
- ⇝ Concevoir des solutions algorithmiques.
- ⇝ Traduire un algorithme dans un langage de programmation, en spécifier les interfaces et les interactions.
- ⇝ Comprendre et réutiliser des codes sources existants.

Décomposition en séances

Globalement la séquence sera découpée en 3 séances de 2 heures.

- ⇝ **Séance 1 - Établissement de la procédure de test des fonctions de tri.**
- ⇝ **Séance 2 - Travail sur la programmation des fonctions de tri.**
- ⇝ **Séance 3 - Travail sur la complexité.**

Le contenu des séances est détaillé ci-après, certaines parties spécifiques seront reprises dans des sections particulières, notamment ce qui concerne :

- ⇝ Les difficultés a priori.
- ⇝ Les critères d'évaluation.
- ⇝ La critique du sujet.

II.3 Séance 1 - 2h - Établissement de la procédure de test des fonctions de tri

- cliquer ici pour voir la fiche élève -

Introduction

Activité fonction test tri est à faire de manière individuelle en classe avec l'aide du professeur. La fonction à tester sera fourni par le professeur. Il s'agit dans un premier temps d'une fonction de tri encapsulant la fonction native sort() de Python.

Objectifs

- ⇝ Comprendre un cahier des charges ;
- ⇝ Préparer une ou des fonctions de test en vue de s'assurer du respect d'un cahier des charges d'une future fonction ou procédure à programmer ;
- ⇝ Programmer des assertions ;

Difficultés et pistes - séance 1

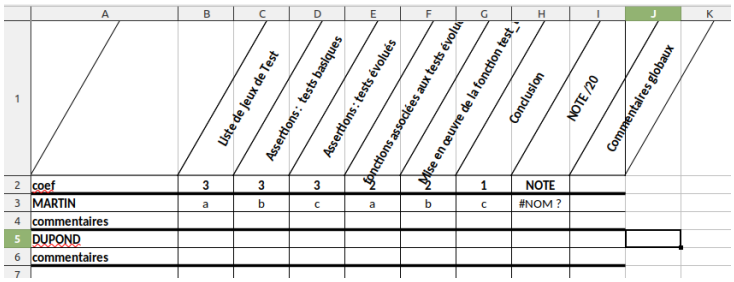
- ⇝ Comprendre le cahier des charges et cerner l'intérêt de faire une fonction test pour l'instant découplée de quelque chose de concret ;
- ⇝ Penser à réaliser des fonctions nécessaires aux tests des assertions ;
- ⇝ Mettre en œuvre la fonction test réalisée.

Aide à prévoir

- ⇝ L'enseignant montre en exemple (programme en python et démonstration) une fonction encapsulée d'extraction de racine carrée, et la procédure permettant de la tester ;
- ⇝ Aides diverses aux petits blocages.

Évaluation - séance 1

Voir grille EvaluationActiviteFonction_test_tri.xls (Click Me !)



	A	B	C	D	E	F	G	H	I	J	K
1		Liste de jeux de Test	Assertions : tests basiques	Assertions : tests évolués	Fonctions associées aux tests évolués	Mise en œuvre de la fonction test_	Conclusion	NOTE /20	Commentaires globaux		
2	coef	3	3	3	2	2	1	NOTE			
3	MARTIN	a	b	c	a	b	c	#NOM ?			
4	commentaires										
5	DUPOND										
6	commentaires										
7											

II.4 Séance 2 - 2h - Travail sur la programmation des fonctions de tri

- cliquer ici pour voir la fiche élève Tri à Bulles-

- cliquer ici pour voir la fiche élève Tri Sélection-

- cliquer ici pour voir la fiche élève Tri Rapide Non Récursif-

Introduction

Activité fonction tri est à faire par groupe de 3, chacun devant réaliser un des algorithmes de tri avec ou sans l'aide des autres. Les trois fonctions seront pour la prochaine séance regroupées dans un même module.

Le travail sera à finir à la maison avec une semaine de marge.

Les plus habiles peuvent programmer l'ensemble des fonctions de tri.

Objectifs

- ⇝ Traduire du pseudo code en Python ;
- ⇝ Déterminer quand il y a besoin de créer des fonctions annexes et les intégrer, si besoin, dans le corps de la fonction de tri ;
- ⇝ Définir et programmer toutes assertions de préconditions ;

Difficultés et pistes - séance 2

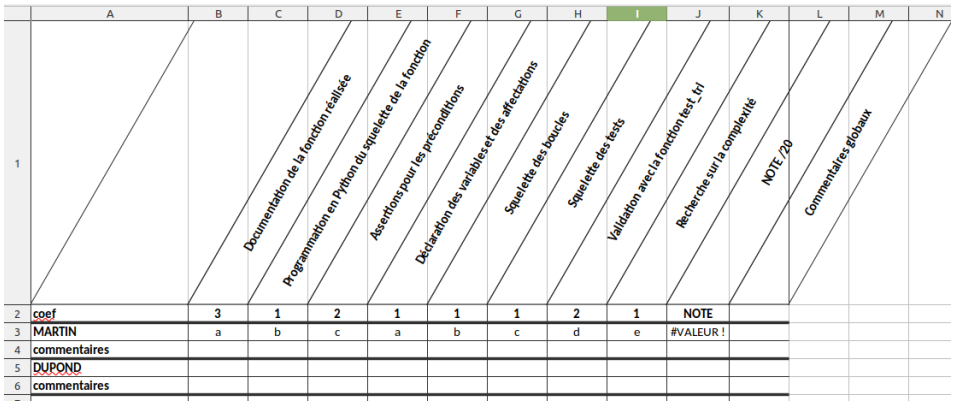
- ⇝ S'assurer que chacun travaille au moins sa fonction ;
- ⇝ Penser à utiliser la fonction de test précédemment réalisée pour valider la fonction ;
- ⇝ Documenter correctement la fonction réalisée.

Aide à prévoir

- ⇝ Faire une démonstration débranchée de tri pour chacune des méthodes (avec un jeu de carte) ;
- ⇝ Aides diverses aux petits blocages et à la programmation en python.

Évaluation - séance 2

Voir grille EvaluationActiviteFonctionTri.xls (Click Me !)



		Documentation de la fonction réalisée	Programmation en Python du squelette de la fonction	Assertions pour les préconditions	Déclaration des variables et des affectations	Squelette des boucles	Squelette des tests	Validation avec la fonction test_tri	Recherche sur la complexité	NOTE_/20	Commentaires globaux
coef		3	1	2	1	1	1	2	1	NOTE	
MARTIN		a	b	c	a	b	c	d	e	#VALEUR !	
commentaires											
DUPOND											
commentaires											

II.5 Séance 3 - 2h - Travail sur la complexité

- cliquer ici pour voir la fiche élève -

Introduction

Activité Complexité Temps Tri est à faire par groupe de 3, les mêmes que pour l'activité fonction tri.
Le déroulement de cette activité se fait en étant accompagné par le professeur.
Une mise en commun des données est effectuée, et un fichier peut être distribué contenant les résultats des expérimentations et leur mise en forme.

Objectifs

- ⇝ Utiliser une bibliothèque "artisanale" ;
- ⇝ "Ressentir " le temps d'exécution en fonction de la taille des données à traiter ;
- ⇝ Visualiser par des graphiques l'influence de la taille des données à traiter sur le temps d'exécution, ainsi que l'influence du type d'algorithme sur ce temps ;
- ⇝ Visualiser par des graphiques que le temps d'exécution d'un algorithme est directement lié au nombre d'opérations unitaires effectuées ;
- ⇝ Visualiser par des graphiques ce qu'est une classe de complexité en temps d'un algorithme.

Difficultés et pistes - séance 3

- ⇝ Utiliser la bibliothèque "complexite_ temps" ;
- ⇝ Manipuler un tableur. Regrouper les données de différents fichiers csv. Tracer des graphiques explicites ;
- ⇝ Comprendre, par la lecture du code de la fonction "mesure_ temps", comment on peut mesurer le temps d'exécution d'une fonction.
- ⇝ Établir un protocole pour estimer la classe de complexité de la fonction native sort().

Aide à prévoir

- ⇝ Aide à l'utilisation de vrais tableurs pour tracer des nuages de point : l'intérêt est d'avoir une grande flexibilité dans une présentation qui peut être un atout majeur lors d'un oral ;
- ⇝ Montrer qu'il est plus opportun de parler en termes de nombre d'opérations unitaires, plutôt que de temps d'exécution, car ce dernier dépend des machines et du contexte de la machine ;
- ⇝ Accompagner les élèves dans leurs réflexions et leurs conclusions ;
- ⇝ Guider pour l'établissement d'un protocole.

Évaluation - séance 3

pas d'évaluation directe, des questions portant sur des connaissances générales sur la notion de complexité seront par exemple intégrées dans un prochain QCM.

- Activité -
Réalisation d'une fonction de test

Nom : **Prénom :** **Classe :**

- cliquer ici pour voir le descriptif professeur -

Objectifs

Programmer une fonction "test_tri" qui servira à tester si une fonction nommée "tri", fonction donnée ci-dessous, effectue sa tâche correctement.

Énoncé/Questions

1/ Fonction "tri"

1.1 – Soit la fonction suivante

```
from copy import *

def tri(l_a_trier:list)->list:
    """ Documenter cette fonction ici """
    pass

    # Préconditions
    assert isinstance(l_a_trier,list),"le paramètre doit être une liste"
    assert len(l_a_trier)>=1,"la liste est vide"
    assert (all(isinstance(elt,int) for elt in l_a_trier)),"éléments non entiers"
    # ajouter un commentaire ici
    l_a_trier.sort()
    # ajouter un commentaire ici
    return l_a_trier
```

La fonction "tri" prend en paramètre une liste d'entiers et renvoie cette même liste avec les entiers triés dans l'ordre croissant.

2/ Cahier des charges de la procédure "test_tri"

- ⇝ La procédure "test_tri" prend en paramètre la fonction précédente "tri" ;
- ⇝ Si la fonction "tri" effectue correctement sa tâche, la fonction "test_tri" n'affichera rien ;
- ⇝ Si la fonction "tri" n'effectue pas correctement sa tâche, la fonction "test_tri" affichera un message d'erreur issue des assertions qu'elle contient ;
- ⇝ La procédure "test_tri" est composée exclusivement d'assertions ;
- ⇝ La procédure "test_tri" peut s'appuyer sur quelques fonctions accessoires.

a) Lister un jeu de tests permettant de valider le bon fonctionnement de la fonction "tri" testée.

b) Réaliser et programmer dans un premier temps la procédure "test_tri" composée de ses assertions dans le fichier "tri_et_test.py".

c) Programmer dans un deuxième temps les fonctions nécessaires aux assertions. Ces fonctions pourront être intégrées au corps de la procédure "test_tri".

3/ Test de la procédure

Ajouter à la fin du fichier "tri_et_test.py" les lignes suivantes puis l'exécuter

1.2 – à la fin du fichier "tri_et_test.py"

```

if __name__=="__main__":
    test_tri(tri)
```

4/ Conclusion

Conclure sur le test effectué, et sur le rôle d'une fonction de test.

- Activité -
Fonction Tri Bulles

Nom : **Prénom :** **Classe :**

- cliquer ici pour voir le descriptif professeur -

Objectifs

Programmer une fonction nommée "tri_bulle", utilisant un algorithme de tri à bulle, qui servira trier une liste d'entiers dans l'ordre croissant.

Énoncé/Questions

1/ Cahier des charges de la fonction "tri_bulle"

- ⇝ La fonction "tri_bulle" prend en paramètre une liste d'entiers ;
- ⇝ La fonction "tri_bulle" renvoie la liste des entiers triés dans l'ordre croissant ;
- ⇝ L'algorithme employé par la fonction "tri_bulle" est un algorithme de tri à bulle ;
- ⇝ La fonction "tri_bulle" est composée d'assertions testant les préconditions nécessaires au bon fonctionnement de la fonction ;
- ⇝ La fonction "tri_bulle" peut s'appuyer sur quelques fonctions accessoires.

2/ Algorithme de tri bulle

Algorithme 1 : fonction tri_bulle(tableau)

Entrées : un tableau
Sorties : le tableau trié

```
1 n ← taille du tableau;
2 pour i ← 0 à n − 1 faire
3     pour j ← 0 à n − i − 2 faire
4         si l'élément d'indice j > l'élément d'indice j + 1 alors
5             Permuter les éléments d'indices j et j + 1
6 retourner le tableau
```

3/ Documentation (prototypage) de la fonction "tri_bulle"
Documenter la fonction "tri_bulle" en précisant dans le corps de celle-ci les paramètres et ce qu'elle renvoie.

4/ Programmation de la fonction "tri_bulle"
Programmer la fonction "tri_bulle" dans le fichier "tri_VotreNom.py", en respectant le cahier des charges, avec sa documentation, penser à programmer les fonctions accessoires dans le corps de la fonction "tri_bulle" si besoin.

5/ Test et validation de la fonction "tri_bulle" avec "test_tri"
À l'aide de la fonction "test_tri" réalisée lors de l'activité précédente, tester et valider le fonctionnement de la fonction "tri_bulle".

6/ Complexité en temps de la fonction "tri_bulle"
Rechercher la complexité en temps de cette fonction (recherche sur internet).
Expliquer, concrètement, ce que signifie cette donnée.

- Activité -
Fonction Tri Sélection

Nom : **Prénom :** **Classe :**

- cliquer ici pour voir le descriptif professeur -

Objectifs

Programmer une fonction nommée "tri_selection", utilisant un algorithme de tri par sélection, qui servira trier une liste d'entiers dans l'ordre croissant.

Énoncé/Questions

1/ Cahier des charges de la fonction "tri_selection"

- ⇝ La fonction "tri_selection" prend en paramètre une liste d'entiers ;
- ⇝ La fonction "tri_selection" renvoie la liste des entiers triés dans l'ordre croissant ;
- ⇝ L'algorithme employé par la fonction "tri_selection" est un algorithme de tri par sélection ;
- ⇝ La fonction "tri_selection" est composée d'assertions testant les préconditions nécessaires au bon fonctionnement de la fonction ;
- ⇝ La fonction "tri_selection" peut s'appuyer sur quelques fonctions accessoires.

2/ Algorithme de tri par sélection

Algorithme 2 : fonction tri_selection(tableau)

Entrées : un tableau
Sorties : le tableau trié

```
1  n ← taille du tableau;
2  i ← 0;
3  tant que i ≤ n − 2 faire
4      imin ← i // indice du plus petit élément restant;
5      j ← i + 1 // on initialise le compteur pour les éléments restants;
6      tant que j ≤ n − 1 faire
7          si l'élément d'indice j < l'élément d'indice imin alors
8              imin ← j
9          incrémenter j;
10     si imin ne vaut plus i alors
11         échanger les éléments de rang i et imin
12     incrémenter i;
13 retourner le tableau
```

3/ Documentation (prototypage) de la fonction "tri_selection"
Documenter la fonction "tri_selection" en précisant dans le corps de celle-ci les paramètres et ce qu'elle renvoie.

4/ Programmation de la fonction "tri_selection"
Programmer la fonction "tri_selection" dans le fichier "tri_VotreNom.py", en respectant le cahier des charges, avec sa documentation, penser à programmer les fonctions accessoires dans le corps de la fonction "tri_selection" si besoin.

5/ Test et validation de la fonction "tri_selection" avec "test_tri"
À l'aide de la fonction "test_tri" réalisée lors de l'activité précédente, tester et valider le fonctionnement de la fonction "tri_selection".

6/ Complexité en temps de la fonction "tri_selection" Rechercher la complexité en temps de cette fonction (recherche sur internet).
Expliquer, concrètement, ce que signifie cette donnée.

- Activité -
Fonction Tri Rapide Non Récursif

Nom : **Prénom :** **Classe :**

- cliquer ici pour voir le descriptif professeur -

Objectifs

Programmer une fonction nommée "tri_rapide", utilisant un algorithme de tri rapide non récursif, qui servira trier une liste d'entiers dans l'ordre croissant.

Énoncé/Questions

1/ Cahier des charges de la fonction "tri_rapide"

- ⇝ La fonction "tri_rapide" prend en paramètre une liste d'entiers ;
- ⇝ La fonction "tri_rapide" renvoie la liste des entiers triés dans l'ordre croissant ;
- ⇝ L'algorithme employé par la fonction "tri_rapide" est un algorithme non récursif de tri rapide ;
- ⇝ La fonction "tri_rapide" est composée d'assertions testant les préconditions nécessaires au bon fonctionnement de la fonction ;
- ⇝ La fonction "tri_rapide" peut s'appuyer sur quelques fonctions accessoires.

2/ Algorithme non récursif de tri rapide L'idée du tri rapide repose sur le paradigme "Diviser pour mieux régner". Il se sépare en deux :

a) partitionner

- ⇝ On choisit arbitrairement un élément du tableau que l'on appelle pivot. Nous prendrons ici toujours le dernier élément.
- ⇝ On place ensuite le pivot à sa place définitive dans le tableau trié de la façon suivante :
  - ⩾ Tous les éléments inférieurs au pivot à sa gauche
  - ⩾ Tous les éléments supérieurs au pivot à sa droite

  De cette manière, on est sûr que le pivot est à sa place.

b) La fonction de tri itérative

- ⇝ On place les éléments de proche en proche à l'aide d'une pile auxiliaire

Algorithme 3 : fonction partitionner(tableau,start,end)

Entrées : un tableau, l'indice de début, l'indice de fin
Sorties : l'indice du pivot à sa place dans le tableau trié

```
1 i ← start - 1 // un indice pour le plus petit élément du tableau;
2 pivot ← l'élément du tableau d'indice end // arbitrairement le dernier élément comme pivot;

3 // pour tous les éléments du tableau d'indices entre start et end;
4 pour j ← start à end-1 faire
5     si l'élément d'indice j <= pivot alors
6         incrémenter i ;
7         permuter les éléments d'indices j et i

8 retourner i + 1 l'indice de la place définitive du pivot
```

Algorithme 4 : fonction tri_rapide(tableau,debut,fin)

Entrées : un tableau, start l'indice de début, end l'indice de fin
Sorties : le tableau trié

```
1  // On crée une pile le stockage des indices;
2  taille ← end - start + 1 // La taille du tableau;
3  stack ← un tableau de taille 0;
4  top ← -1 // Initialiser le haut de la pile;

5  // Pousser les valeurs initiales de start et end au début de la pile;
6  incrémenter top ;
7  ajouter start au début de stack // En position top;
8  incrémenter top ;
9  ajouter end au début de stack à la suite de start // En position top;

10 //Tant que la pile n'est pas vide;
11 tant que top >= 0 faire
12     // On récupère start et end de la pile;
13     end ← stack[top];
14     décrémenter top;
15     start ← stack[top];
16     décrémenter top;

17     //Placer le pivot à sa place définitive avec la fonction partitionner;
18     p = partition(tableau,start,end);

19     //Si il y a des éléments inférieurs au pivot;
20     //Pousser leurs indices à gauche de la pile;
21     si p-1 > start alors
22         incrémenter top ;
23         stack[top] ← start;
24         incrémenter top ;
25         stack[top] ← p - 1;

26     // Si il y a des éléments supérieurs au pivot;
27     // Pousser leurs indices à droite de la pile;
28     si p + 1 < end alors
29         incrémenter top ;
30         stack[top] ← p + 1;
31         incrémenter top ;
32         stack[top] ← end;

33 retourner le tableau trié
```

3/ Documentation (prototypage) de la fonction "tri_rapide"
Documenter la fonction "tri_rapide" en précisant dans le corps de celle-ci les paramètres et ce qu'elle renvoie.

4/ Programmation de la fonction "tri_rapide"
Programmer la fonction "tri_rapide " dans le fichier "tri_VotreNom.py", en respectant le cahier des charges, avec sa documentation, penser à programmer les fonctions accessoires dans le corps de la fonction "tri_rapide" si besoin.

5/ Test et validation de la fonction "tri_rapide" avec "test_tri"
À l'aide de la fonction "test_tri" réalisée lors de l'activité précédente, tester et valider le fonctionnement de la fonction "tri_rapide ".

6/ Complexité en temps de la fonction "tri_rapide"
Rechercher la complexité en temps de cette fonction (recherche sur internet). Expliquer, concrètement, ce que signifie cette donnée.

- Activité -
Complexité Temps Tri

Nom : **Prénom :** **Classe :**

- cliquer ici pour voir le descriptif professeur -

Objectifs

À l'aide de la procédure "mesure_temps_tri", mesurer le temps d'exécution de différents algorithmes de tri, et l'ordre de grandeur du nombre d'opérations unitaires effectué par ces algorithmes afin d'en déterminer expérimentalement la complexité en temps.

Énoncé/Questions

1/ Algorithmes de tri à tester

Les algorithmes dont on cherche à estimer la complexité en temps sont ceux établis précédemment au sein des fonctions "tri_selection", "tri_bulle", "tri_rapide" et "tri_sort".

- ⇝ "tri_selection" effectue un tri croissant d'une liste d'entiers en utilisant un algorithme de tri par sélection ;
- ⇝ "tri_bulle" effectue un tri croissant d'une liste d'entiers en utilisant un algorithme de tri par sélection ;
- ⇝ "tri_rapide" effectue un tri croissant d'une liste d'entiers en utilisant un algorithme de tri par pivot non récursif ;
- ⇝ "tri_sort" effectue un tri croissant d'une liste d'entiers en utilisant l'algorithme de tri natif de python.

2/ Mesure du temps d'exécution de chaque fonction de tri

Les fonctions "mesure_temps" et "mesure_complexite" font partie du module "complexite_temps.py" : il faut donc importer ce module au sein de votre module comportant les fonctions de tri.
Ces deux fonctions nécessitent l'utilisation de listes, d'entiers triés ou non triés, enregistrées dans des fichiers de type csv : Le professeur peut fournir ces fichiers, ou ils peuvent être générés à l'aide des fonctions "generateurListeCsv" ou "genereToutesLesListes" contenues aussi dans le module "complexite_temps.py".

a) À partir de l'aide de la fonction "mesure_temps", des listes d'entiers non triés sous forme de fichiers de type csv, et des différentes fonctions de tri à tester, proposer un programme en python permettant de mesurer le temps d'exécution des quatre algorithmes de tri en fonction du nombre n d'entiers à trier.

b) Relever dans le code de la fonction "mesure_temps" les lignes de code servant à mesurer le temps d'exécution de la fonction passée en paramètre.

c) Réaliser et exécuter ce programme pour mesurer le temps d'exécution des quatre algorithmes de tri en fonction du nombre n d'entiers à trier.

d) Rassembler ces données sur un même fichier de tableur, et tracer pour chaque fonction sur le même graphe $t = f(n)$, avec t le temps d'exécution de la fonction et n le nombre d'entiers dans la liste à trier.
Adopter des échelles permettant d'obtenir une bonne lisibilité des graphes.

e) Conclure en comparant le temps d'exécution de chaque algorithme et observer l'influence de la taille des listes à trier sur ce temps.

3/ Estimation de la complexité en temps de chaque algorithme de tri

Pour estimer la complexité des algorithmes de tri, on va compter le nombre de tours de boucle effectué dans chaque algorithme pour traiter les listes.
Ceci nous donnera un ordre de grandeur du nombre d'opérations unitaires effectuées, et donc nous permettra de déterminer expérimentalement la complexité de l'algorithme étudié.

a) À partir de l'aide de la fonction "mesure_complexite", des listes d'entiers non triés sous forme de fichiers de type csv, et des différentes fonctions de tri à tester, réaliser le programme en python permettant de compter le nombre de tours de boucle effectué par les trois algorithmes de tri, sélection, bulle et rapide, en fonction du nombre n d'entiers à trier.

b) Rassembler ces données sur un même fichier de tableur, et tracer pour chaque fonction sur le même graphe $ntb = f(n)$, avec ntb le nombre de tours de boucle effectué et n le nombre d'entiers dans la

liste à trier.
Adopter des échelles permettant d'obtenir une bonne lisibilité des graphes.

c) Ajouter sur le précédent graphe les courbes suivantes :

⇝ $ntb = n.log(n)$; avec log le logarithme en base 2

⇝ $ntb = n$

⇝ $ntb = n^2$

Ces trois courbes donnent l'allure des courbes représentant les complexités suivantes : $O(n.log(n))$, $O(n)$, et $O(n^2)$.

d) Par comparaison des courbes empiriques, question c), avec les courbes correspondantes aux algorithmes testés, question b), en déduire la complexité en temps des algorithmes de tri par sélection, bulle et rapide.

Si deux des courbes $ntb = f(n)$ ont la même allure, on peut alors dire que les deux algorithmes ont la même complexité.

e) Conclure quant à la complexité observée expérimentalement, et empirique de chaque algorithme de tri étudié.

4/ Estimation de la complexité en temps de l'algorithme de tri natif de python (fonction sort())

N'ayant pas accès au code de la fonction sort() de python, on ne peut pas compter les tours de boucle qu'elle effectue. En revanche, il est possible de l'estimer.
Proposer, et réaliser, un protocole expérimental permettant de tracer comme précédemment la courbe $ntb = f(n)$ pour la fonction "tri_sort" encapsulant la fonction native sort() de python.

On peut se baser sur la mesure du temps d'exécution

En déduire la classe de complexité de la fonction native sort() de python.

5/ Étude des pires cas pour chaque fonction de tri

À l'aide des fonctions "mesure_temps" ou "mesure_complexite", des listes d'entiers triés, sous ordre croissant et décroissant, sous forme de fichiers de type csv, et des différentes fonctions de tri à tester :

⇝ **proposer un programme** en python permettant de mesurer le temps d'exécution et le nombre de tours de boucles effectués des quatre algorithmes de tri en fonction du nombre n d'entiers à trier.

⇝ **Réaliser et exécuter ce programme** pour mesurer le temps d'exécution et le nombre de tours de boucles effectués des quatre algorithmes de tri en fonction du nombre n d'entiers à trier pour tous les cas.

⇝ **Présenter les résultats obtenus** sous forme de courbes explicites (un jeu de courbes superposées avec les différentes complexités par algorithme).

⇝ **En déduire quel est le pire cas** pour chaque algorithme de tri ainsi que la complexité moyenne et la complexité dans le pire cas.

III Rédiger un document (1 page) décrivant les difficultés que vous voyez a priori

⇝ pour les élèves,

⇝ listant les apports pédagogiques que vous devrez faire pour accompagner les élèves. Par exemple :

- ⩾ comment mesurer le temps de calcul d'une exécution d'une fonction,
- ⩾ comment « bien » faire un graphique (peut-être en collaboration avec le professeur de physique ou mathématique),
- ⩾ etc.)

Difficultés et pistes - séance 1

⇝ Comprendre le cahier des charges et cerner l'intérêt de faire une fonction test pour l'instant découplée de quelque chose de concret ;

⇝ Penser à réaliser des fonctions nécessaires aux tests des assertions ;

⇝ Mettre en œuvre la fonction test réalisée.

Difficultés et pistes - séance 2

⇝ S'assurer que chacun travaille au moins sa fonction ;

⇝ Penser à utiliser la fonction de test précédemment réalisée pour valider la fonction ;

⇝ Documenter correctement la fonction réalisée.

Difficultés et pistes - séance 3

⇝ Utiliser la bibliothèque "complexite_ temps" ;

⇝ Manipuler un tableur. Regrouper les données de différents fichiers csv. Tracer des graphiques explicites ;

⇝ Comprendre, par la lecture du code de la fonction "mesure_ temps", comment on peut mesurer le temps d'exécution d'une fonction.

⇝ Établir un protocole pour estimer la classe de complexité de la fonction native sort().

IV Rédiger un document (1 page) décrivant les critères d'évaluation du projet

Cette partie devra aborder quels jeux de test vous attendez, et comment vous évaluez ces jeux de test (pertinence, élaboration pratique, etc.)

- ⇝ une soutenance de projet est-elle souhaitable ?
- ⇝ Un rapport est-il souhaitable ?
- ⇝ Avec quel contenu ?

1/ Il est attendu des élèves qu'ils testent les fonctions de tri sur des listes d'entiers. Ces tests s'entendent en temps et en complexité. Les fichiers csv avec les listes d'entiers sont fournis. Voir annexes pour les fonctinons génératrices éventuellement données aux élèves les plus à l'aise.

2/ Aucune soutenance n'est attendue, cette séquence est menée sous forme de TP accompagné car l'ensemble des tâches est complexe pour des élèves de 1ère et terminale.

3/ De même un rapport n'est pas attendu, l'objectif étant que tous les élèves puissent avancer et aller au bout des activités. Si lors d'une séance, un groupe ne finalisait pas l'activité, les scripts de corrections seraient fournis pour qu'il puisse poursuivre la fois suivante.

4/ La notation s'entend donc via les grilles suivantes.

♪ Évaluation - séance 1

Voir grille EvaluationActiviteFonction_test_tri.xls (Click Me !)

	A	B	C	D	E	F	G	H	I	J	K
1		Liste de jeux de Test	Assertions : tests basiques	Assertions : tests évolués	Fonctions associées aux tests évolués	Mise en œuvre de la fonction test_	Conclusion	NOTE /20	Commentaires globaux		
2	coef	3	3	3	2	2	1	NOTE			
3	MARTIN	a	b	c	a	b	c	#NOM ?			
4	commentaires										
5	DUPOND										
6	commentaires										
7											

♪ Évaluation - séance 2

Voir grille EvaluationActiviteFonctionTri.xls (Click Me !)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		Documentation de la fonction réalisée	Programmation en Python du squelette de la fonction	Assertions pour les préconditions	Déclaration des variables et des affectations	Squelette des boucles	Squelette des tests	Validation avec la fonction test_tri	Recherche sur la complexité	NOTE /20	Commentaires globaux			
2	coef	3	1	2	1	1	1	2	1	NOTE				
3	MARTIN	a	b	c	a	b	c	d	e	#VALEUR !				
4	commentaires													
5	DUPOND													
6	commentaires													

♪ Évaluation - séance 3

pas d'évaluation directe, des questions portant sur des connaissances générales sur la notion de complexité seront par exemple intégrées dans un prochain QCM.

V Rédiger un document (1 page) critiquant le sujet

⇝ **Le sujet entre-t-il dans le cadre des programmes ?**
Mis à part quelques éléments, le sujet rentre dans le cadre du programme de NSI. Seuls les algorithmes de tri bulles, et de tri rapide non récursif ne sont pas au programme. À défaut d'être au programme, ils présentent l'intérêt de montrer au moins deux choses aux élèves :

- ⩾ il existe des méthodes de tri autres que l'insertion ou la sélection.
- ⩾ ces techniques, qui a priori peuvent sembler originales aux élèves, peuvent se révéler plus efficaces.

Remarque supplémentaire

Le tri rapide non récursif est très complexe à expliquer et à comprendre. Même sans avoir de notions sur la récursivité, le tri rapide récursif est beaucoup plus accessible. Fabien SERENKO a fait faire ce projet à ses élèves de 1ère NSI, mais ils ont travaillé sur le tri rapide récursif.

⇝ **Le sujet fait-il appel à des compétences d'autres disciplines ?**
Savoir tracer un graphique est une compétence nécessaire à toute les matières, scientifiques dures, ou pas, mais bien souvent mal maîtrisée.

⇝ **Évaluation du nombre de séances nécessaires pour accompagner les élèves**
Au troisième trimestre de 1ère NSI, 3 à 4 séances de 2h, guidées par l'enseignant, avec peu de travail à réaliser à la maison, le travail se faisant très majoritairement en classe.

⇝ **Évaluation de sa difficulté, de la taille du groupe pouvant aborder un tel travail**

- ⩾ Une **première difficulté** est de déterminer les jeux de test à réaliser pour la fonction "test_tri" : on peut proposer deux niveaux de tests, certains très basiques, comparer le travail de la fonction de tri à une liste connue, sont accessibles facilement à l'ensemble des élèves, et d'autres plus évolués peuvent être mis en place par les élèves les plus performants. Une mise en commun des travaux pour la classe complète permet de faire comprendre tout ce qu'on peut demander à la fonction "test_tri" de réaliser.
- ⩾ Une **deuxième difficulté** est, pour l'élève, la réalisation des fonctions dont la forme doit répondre à un cahier des charges précis.
- ⩾ Une **troisième difficulté** peut être la mise en œuvre du module "complexite_temps".
- ⩾ Une **quatrième difficulté** peut être l'exploitation des fichiers csv obtenu pour en obtenir des graphiques parlants.
- ⩾ Une **dernière difficulté** pourra être l'exploitation de courbes obtenue afin d'y observer des similitudes entre elles.

Bien que certains travaux se fassent par équipes de 3, par exemple la réalisation des fonctions de tri, une mise en commun des travaux de la classe complète est à chaque fois réalisée afin que chacun puisse avoir des éléments concrets et en tirer des apprentissages.

⇝ **Évaluation de son intérêt pédagogique**
La réalisation d'un tel projet mené entièrement par des élèves en autonomie, même avec de l'aide de l'enseignant, ne semble pas à la portée d'élèves de 1ère ou terminale NSI, ou juste réservée aux excellents élèves, peut-être 10% d'une promotion. De plus, le temps nécessaire à la réalisation de ce dernier est difficile à trouver en raison de l'ampleur du programme. En terminale, le programme devant être fini pour Mars, un tel projet ne peut être réalisé qu'après, pour préparer le grand oral. La difficulté est alors de l'adapter au grand oral, ce qui parait impossible en raison du format imposé à l'épreuve, pas de documents, ni de support informatique pour la présentation ! La seule possibilité serait de faire réaliser ce projet au troisième trimestre en 1ère NSI avec un guidage important de la part de l'enseignant afin que chaque élève, même s'il est peu autonome, puisse en tirer des apprentissages et de l'entraînement par rapport au programme de 1ère.

Les apports pédagogiques seront :

- ⩾ **s'entraîner à programmer** en python à partir de pseudo-code ;
- ⩾ **s'entraîner à debugger** ;
- ⩾ **apprendre à réaliser un programme** devant répondre à un cahier des charges précis, et utilisable pour d'autres programmeurs : suivre des étapes dans la réalisation d'un projet ;
- ⩾ **voir**, de visu, **ce qu'est la complexité en temps** d'un algorithme ;
- ⩾ pour les meilleurs, **comprendre** ce qu'est en complexité **la notion de fonction O**().

VI Annexes - scripts proposés en guise de correction

VI.1 Les tris et la fonctions de test

1.3 – Modules utiles

```
import random
import copy
import complexite_temps #module maison
```

1.4 – Fonction test_tri

```
        for elt in liste :
            if not isinstance( elt, int) :
                return False
        return True
    assert isinstance( tab , list ) , "tab n'est pas une liste"
    assert len(tab)>=1, "tab est vide"
    assert estListeEntiers(tab), "tab n'est pas composée d'entiers"
    '''
    ###############################################################################
    n=len(tab)
    tri_rapide_recursif(tab, 0, n-1)
    return tab

def test_tri( fonction_tri ):
    # outils pour tests d'assertion ################################################

    #fonction vérifiant si une liste est triée dans l'ordre croissant
    def est_triee( liste ):
        n=len( liste )
        for i in range( n-1 ):
            if liste[i] > liste[i+1] :
                return False
        return True

    # fonction renvoyant une liste d'indice d'un élément spécifié dans une liste
    def recherche( liste, element ):
        positions = []
        n = len( liste )
        for i in range( n ) :
            if liste[ i ] == element :
                positions.append( i )
        return positions

    #fonction vérifiant si tous les éléments d'une liste sont présents dans une autre
    def sont_memes_elements( liste1, liste2 ):
        for elt in liste1 :
            if len( recherche( liste1, elt) ) != len( recherche( liste2, elt) ) :
                return False
        return True

    # fonction créant une nouvelle liste d'éléments pris au hasard
    def nouvelle_liste( nombreElements, gamme ):
        liste=[]
        for i in range( nombreElements ) :
            liste.append( random.randint(-gamme, gamme) )
        return liste

    # tests basiques ###########################################################
    assert fonction_tri([3,9,6,0,-3,2])==[-3,0,2,3,6,9], "echec test basique non triée"
    assert fonction_tri([-2,0,2,4,6,8])==[-2,0,2,4,6,8], "echec test basique triée"
```

1.5 – Outils

```
# outils pour permutter #########
def permute(i, j, tab):
    m=tab[i]
    tab[i]=tab[j]
    tab[j]=m
```

1.6 – Tri natif de python

```
def tri_sort(liste_a_trier) :
    # preconditions ###############################################################
    #def estListeEntiers( liste ) :
    #   for elt in liste :
    #           if not isinstance( elt, int) :
    #                   return False
    #   return True
    #assert isinstance( liste_a_trier , list ) , "liste_a_trier n'est pas une liste"
    #assert len(liste_a_trier)>=1, "liste_a_trier est vide"
    #assert estListeEntiers(liste_a_trier), "liste_a_trier n'est pas composée d'entiers"
    ##############################################################################
    liste_a_trier.sort()
    return liste_a_trier
```

1.7 – Tri selection

```
def tri_selection(tab):
    '''
    La fonction tri_selection prend en pramètre :
    * tab : une liste d'entiers non vide à trier
    La fonction tri_selection tri la liste tab dans l'ordre croissant et renvoie :
    * la liste elle-même tab triée dans l'ordre croissant
    '''
    # preconditions ###############################################################
    #def estListeEntiers( liste ) :
    #   for elt in liste :
    #           if not isinstance( elt, int) :
    #                   return False
    #   return True
    #assert isinstance( tab , list ) , "tab n'est pas une liste"
    #assert len(tab)>=1, "tab est vide"
    #assert estListeEntiers(tab), "tab n'est pas composée d'entiers"
    ##############################################################################
    n=len(tab)
    for i in range(n-1):
        mini=i
        for j in range(i+1, n):
            #fonction utilisée par mesure_complexite pour compter
            #le nombre ntb de tours de boucle
            complexite_temps.compteTourBoucle()
            if tab[j]<tab[mini]:
                mini=j
        permute(i, mini, tab)
    return tab
```

1.8 – Tri bulle

```
def tri_bulle(tab):
    '''
    La fonction tri_bulle prend en pramètre :
    * tab : une liste d'entiers non vide à trier
    La fonction tri_bulle tri la liste tab dans l'ordre croissant et renvoie :
    * la liste elle-même tab triée dans l'ordre croissant
    '''
    # preconditions ###############################################################
    '''
    def estListeEntiers( liste ) :
        for elt in liste :
            if not isinstance( elt, int) :
```

```
                return False
        return True
    assert isinstance( tab , list ) , "tab n'est pas une liste"
    assert len(tab)>=1, "tab est vide"
    assert estListeEntiers(tab), "tab n'est pas composée d'entiers"
    '''
    ###############################################################################
    n=len(tab)
    for i in range(n):
        for j in range(n-i-1):
             #fonction utilisée par mesure_complexite pour compter
             #le nombre ntb de tours de boucle
            complexite_temps.compteTourBoucle()
            if tab[j]>tab[j+1]:
                permute(j, j+1, tab)
    return tab
```

1.9 – Tri rapide récursif - fonction de partitionnement

```
def partitionne(tab,debut,fin,pivot):
    '''
    La fonction partitionne prend en pramètre :
    * tab : une liste d'entiers non vide à trier
    * debut : un entier correspondant à l'indice du premier élément de la liste à partir
    duquel on effectue le partitionnement
    * fin : un entier correspondant à l'indice du dernier élément de la liste pour laquelle
    on effectue le partitionnement
    La fonction partitionne la liste tab de l'indice debut à fin autour du pivot,
    et renvoie : un entier correspondant à l'indice du pivot choisi
    '''
    # preconditions ###############################################################
    '''
    def estListeEntiers( liste ) :
        for elt in liste :
            if not isinstance( elt, int) :
                return False
        return True
    assert isinstance( tab , list ) , "tab n'est pas une liste"
    assert len(tab)>=1, "tab est vide"
    assert estListeEntiers(tab), "tab n'est pas composée d'entiers"

    assert isinstance( debut , int ) , "debut n'est pas un entier"
    assert debut>=0, "debut n'est pas >=0"

    assert isinstance( fin , int ) , "fin n'est pas un entier"
    assert fin<len(tab), "fin est strictement supérieur à l'indice du dernier élément de tab"

    assert debut<fin, "debut n'est pas strictement inférieur à fin"

    assert isinstance( pivot , int ) , "pivot n'est pas un entier"
    assert pivot<=fin, "pivot n'est pas inférieur ou égal à fin"
    assert pivot>=debut, "pivot n'est pas supérieur ou égal à debut"
    '''
    ###############################################################################
    permute(pivot, fin, tab)
    j=debut
    for i in range(debut, fin):
        #fonction utilisée par mesure_complexite pour compter
        #le nombre ntb de tours de boucle
        complexite_temps.compteTourBoucle()
        if tab[i]<=tab[fin]:
            permute(i, j, tab)
            j=j+1
    permute(j, fin, tab)
    return j
```

```python
def tri_rapide_recursif(tab, debut, fin):
    '''
    La fonction tri_rapide_recursif prend en pramètre :
    * tab : une liste d'entiers non vide à trier
    * debut : un entier correspondant à l'indice du premier élément de la liste à partir
    duquel on effectue le tri
    * fin : un entier correspondant à l'indice du dernier élément de la liste pour laquelle
    on effectue le tri
    La fonction tri_rapide_recursif tri dans l'ordre croissant la liste tab
    de l'indice debut à fin
    '''
    if debut<fin:
         #fonction utilisée par mesure_complexite pour compter
         #le nombre ntb de tours de boucle
        complexite_temps.compteTourBoucle()
        pivot=fin
        pivot=partitionne(tab, debut, fin, pivot)
        tri_rapide_recursif(tab, debut, pivot-1)
        tri_rapide_recursif(tab, pivot+1, fin)

def tri_rapide(tab):
    '''
    La fonction tri_rapide prend en pramètre :
    * tab : une liste d'entiers non vide à trier
    La fonction tri_rapide tri la liste tab dans l'ordre croissant et renvoie :
    * la liste elle-même tab triée dans l'ordre croissant
    '''
    # preconditions ###########################################################
    '''
    def estListeEntiers( liste ) :
        for elt in liste :
            if not isinstance( elt, int) :
                return False
        return True
    assert isinstance( tab , list ) , "tab n'est pas une liste"
    assert len(tab)>=1, "tab est vide"
    assert estListeEntiers(tab), "tab n'est pas composée d'entiers"
    '''
    ###########################################################################
    n=len(tab)
    tri_rapide_recursif(tab, 0, n-1)
    return tab
```

VI.2 Pour les curieux le tri rapide non récursif

1.11 – Tri rapide itératif - fonction de partitionnement

```
# Partitionner
def partition(tab, start, end):
    """
    On choisit arbitrairement le dernier élément comme pivot
    On le place au bon endroit dans le tableau
    Tous les éléments plus petits à gauche du pivot
    Tous les éléments plus grands à droite du pivot

    tab un tableau, start l'indice de début, end l'indice de fin
    """
    # un indice pour placer le plus petit element au début du tableau
    i = ( start - 1 )
    # arbitrairement le dernier élément du tableau comme pivot
    pivot = tab[end]

    # Pour tous les éléments du tableau
    for j in range(start, end):
        # print("==deb=== j : ", j)
        # print(i)
        # print(tab)
        #si l'élément j du tableau est inferieur ou égal au pivot
        if   tab[j] <= pivot:
            # On incremente l'indice du plus petit élément
            # RAPPEL on était partis de l'indice i=-1 donc maintenant ce sera 0
            # si le premier élément est inférieur au pivot
            i = i + 1
            # On echange les éléments d'indice i et j, au départ avec lui même éventuellement
            tab[i], tab[j] = tab[j], tab[i]
        # print(i)
        # print(tab)
        # print("==fin===")
    # Tous les éléments d'indices entre start et end qui sont inférieurs au pivot sont
    # maintenant à gauche du pivot
    # On échange les éléments d'indices i+1 et end car on ne s'est pas occupé
    # de l'indice end dans la boucle
    tab[i + 1], tab[end] = tab[end], tab[i + 1]
    # i+1 est l'indice du premier élément supérieur au pivot
    return (i + 1)
```

```python
def tri_rapide_non_recursif(tab, start=None, end=None):
    """
    tab un tableau, start l'indice de début, end l'indice de fin
    start et end à None pour initaliser comme les autres fonctions de tri
    """

    # initialiser
    if start is None: start = 0
    if end is None: end = len(tab)-1

    # On crée une pile au maximum sa taille est celle du tableau
    size = end - start + 1 # la taille du tableau
    stack = [0] * (size) # On initialise la pile avec des 0

    # On initialise le haut de la pile
    top = -1

    # On pousse les valeurs initiales start end au début de la pile
    top = top + 1
    stack[top] = start
    top = top + 1
    stack[top] = end

    # On place les éléments du tableau tant que la pile n'est pas vide
    while top >= 0:

        # On récupère end et start de la pile
        end = stack[top]
        top = top - 1
        start = stack[top]
        top = top - 1

        # On place le pivot à sa place définitive dans le tableau trié
        p = partition( tab, start, end )

        # Si il y a des éléments inférieurs au pivot,
        # On pousse leurs indices à gauche de la pile
        if p-1 > start:
            top = top + 1
            stack[top] = start
            top = top + 1
            stack[top] = p - 1

        # Si il y a des éléments supérieurs au pivot,
        # On pousse leurs indices à droite de la pile
        if p + 1 < end:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = end
    return tab
```

VI.3 Le module complexité

1.13 – Module maison - complexité

```
import time
import random
import copy

def generateurListeCsv(tri, n, nomFichier):
    '''
    generateurListeCsv prend en paramètres :
        - tri : une chaine de caractère désignant si la liste est triée
          d'une certaine manière ou non : prend les valeurs "CROISSANT",
          "DECROISSANT" ou "ALEATOIRE";
        - n : un entier correspondant au nombre d'éléments dans la liste;
        - nomfichier : une chaine de caractères correspondant au nom
          de fichier en ".csv" ( ex : "toto.csv" )
    generateurListeCsv génère un fichier csv contenant une liste d'entier
    triée croissante, décroissante ou non triée.
    '''
    fichier=open(nomFichier,"w")
    if tri.lower()=="croissant" :
        i=0
        while i<n :
            if i!= n-1 :
                fichier.write(str(i)+";")
            else :
                fichier.write(str(i))
            i=i+1
    elif tri.lower()=="decroissant" :
        i=n-1
        while i>=0 :
            if i!=0 :
                fichier.write(str(i)+";")
            else :
                fichier.write(str(i))
            i=i-1
    else :
        i=0
        while i<n :
            if i!= n-1 :
                fichier.write(str(random.randint(0,n))+";")
            else :
                fichier.write(str(i))
            i=i+1
    fichier.close()

def genereToutesLesListes(nombre, n):
    '''
    genereToutesLesListes prend en paramètre :
        - nombre : entier >=1 correspondant au nombre de listes aléatoires à générer;
        - n : entier correspondant au nombre dentiers dans les listes à générer
    genereToutesLesListes génère des fichiers .csv contenant des listes d'entiers.
    '''
    generateurListeCsv("croissant",n,"croissant"+str(n)+".csv")
    generateurListeCsv("decroissant",n,"decroissant"+str(n)+".csv")
    for i in range(nombre):
        print("i : "+str(i))
        generateurListeCsv("aleatoire",n,str(i)+"aleatoire"+str(n)+".csv")
```

```python
def restitueListe(nomFichier):
    '''
    retitueListe prend en paramètre :
     - une chaine de caractères correspondant au nom de fichier csv généré
       avec la procédure generateurListeCsv.
    restitueListe renvoie :
     - une liste d'entier
    '''
    fichier=open(nomFichier,"r")
    ligne=fichier.read()
    l=ligne.split(";")
    n=len(l)
    i=0
    while i<n :
        l[i]=int(l[i])
        i=i+1
    fichier.close()
    return l

def mesureTempsExecutionTri(fonctionTri, liste):
    '''
    mesureTempsExecutionTri prend en paramètres :
        - fonctionTri : une fonction de tri prenant en paramètre une liste d'entiers;
        - liste : une liste d'entiers.
    mesureTempsExecutionTri renvoie :
        - un flottant correspondant au temps écoulé pour l'exécution de la fonction.
    '''
    t1=time.time()
    fonctionTri(liste)
    dt=time.time()-t1
    return dt

def mesure_temps(fonctionTri, nMax, nomFichierRapportCsv, nomListeATrierCsv):
    '''
    mesure_temps prend en paramètres :
    - fonctionTri : une fonction de tri prenant en paramètre une liste d'entiers;
    - nMax : un entier >=1, correspondant à la taille de la plus grande liste à tester;
    - nomFichierRapportCsv : une chaine de caractères correpondant au nom du fichier
      csv contenant les données à renvoyer comme résultat;
    - nomListeATrierCsv : une chaine de caractères correpondant au nom du fichier csv
      contenant une liste de données (nombre entiers) à trier, le nombre de données
      à trier de la liste doit être >= nMax.
    mesure_temps affiche le temps mis, et le nombre de tours de boucle effectué par
    fonctionTri pour effectuer son tri pour différentes longueurs des listes à trier,
    et génère un fichier csv contenant les résultat de ces mesures et comptages.
    '''

    fichier=open(nomFichierRapportCsv,"w")
    fichier.write("n;"+nomFichierRapportCsv+" t(s)\n")
    print("####################  "+nomFichierRapportCsv+"  ##############################")

    tableau=[] #liste de listes de temps pour chaque valeur de n
    tabn=[] #liste des valeurs de n

    #mesure des temps d'exécution
    listeAleatoire=restitueListe(nomListeATrierCsv)
    assert len(listeAleatoire)>=nMax, "liste à traiter trop petite ou nMax trop grand"
    tab=[]
    n=1
    while n<=nMax :
        listeATrier=copy.deepcopy(listeAleatoire[:n])
        t=mesureTempsExecutionTri(fonctionTri, listeATrier)
        print("temps pour trier "+str(n)+" entiers d'une liste aléatoire : "+str(t)+" s")
        fichier.write(str(n)+";"+str(t)+"\n")
        n=n*2

    fichier.close()
```

```
68
69  def mesure_complexite(fonctionTri, nMax, nomFichierRapportCsv, nomListeATrierCsv):
70      '''
71      mesure_complexite prend en paramètres :
72      - fonctionTri : une fonction de tri prenant en paramètre une liste d'entiers;
73      - nMax : un entier >=1, correspondant à la taille de la plus grande liste à tester;
74      - nomFichierRapportCsv : une chaine de caractères correpondant au nom du fichier
75        csv contenant les données à renvoyer comme résultat;
76      - nomListeATrierCsv : une chaine de caractères correpondant au nom du fichier csv
77        contenant une liste de données (nombre entiers) à trier, le nombre de données
78        à trier de la liste doit être >= nMax.
79
80      mesure_complexite nécessite l'insertion de la fonction compteTourBoucle() au sein de
81      chaque boucle (dans le code de la fonction testée) de la fonction à testée.
82
83      mesure_complexite affiche le temps mis, et le nombre de tours de boucle effectué par
84      fonctionTri pour effectuer son tri pour différentes longueurs des listes à trier, et
85      génère un fichier csv contenant les résultat de ces mesures et comptages.
86      '''
87
88      fichier=open(nomFichierRapportCsv,"w")
89      fichier.write("n;"+nomFichierRapportCsv+" t(s);"+nomFichierRapportCsv+" opérations\n")
90      print("####################  "+nomFichierRapportCsv+"  ################################")
91
92      tableau=[] #liste de listes de temps pour chaque valeur de n
93      tabn=[] #liste des valeurs de n
94
95      #mesure des temps d'exécution
96      listeAleatoire=restitueListe(nomListeATrierCsv)
97      tab=[]
98      n=1
99      while n<=nMax :
100         listeATrier=copy.deepcopy(listeAleatoire[:n])
101         global spy #espion
102         spy=0     #espion
103         t=mesureTempsExecutionTri(fonctionTri, listeATrier)
104         print("nombre de tours de boucle : "+str(spy))   #espion
105         print("temps pour trier "+str(n)+" entiers d'une liste aléatoire : "+str(t)+" s")
106         fichier.write(str(n)+";"+str(t)+";"+str(spy)+"\n")
107         n=n*2
108
109     fichier.close()
110
111 def compteTourBoucle():
112     '''
113     procédure qui incrémente lorsqu'elle est appelée la variable globale spy déclarée
114     dans le programme d'appel : permet de compter les tours de boucle
115     '''
116     global spy
117     try:
118         spy=spy+1
119     except NameError :
120         pass
```