TP – tri de valeurs dans une liste

Copiez le répertoire du TP et collez-le dans votre espace personnel. Lancez le logiciel Spider.

I. Introduction

On se propose dans ce TP de coder et évaluer les performances de différents algorithmes de tri.
Pour cela, nous pourrons éprouver les algorithmes avec les liste T_ord et T_alea, fournies en début du script, qui sont deux listes d'entiers, triée pour la première, et aléatoire pour la seconde.

II. Tri Insertion

On rappelle l'algorithme du tri insertion :

Pour i de 0 à n-1 faire
  $x \leftarrow T[i]$
  j ← 0
  Tant que $j <= i - 1 \wedge T[j] < x$ # ∧ => &
    $j \leftarrow j + 1$
  Fin Tant que
  Pour k de i-1 à j-1 par pas de -1 faire # (j-1 exclu !)
    $T[k+1] \leftarrow T[k]$
  Fin Pour
  $T[j] \leftarrow x$
Fin Pour

1) Dans ***tri_insertion.py*** codez la fonction appelée « tri_insertion » réalisant le tri insertion.
Testez votre fonction avec les listes fournies.

Remarque : pour réaliser une boucle for en décrémentant, il suffit d'utiliser ***range*** à bon escient, exemple :

```
>>> range(6,3,-1)
[6, 5, 4]
```

La fonction time.time() donne la date à la microseconde près. On souhaite utiliser cette fonction pour évaluer la rapidité du tri.

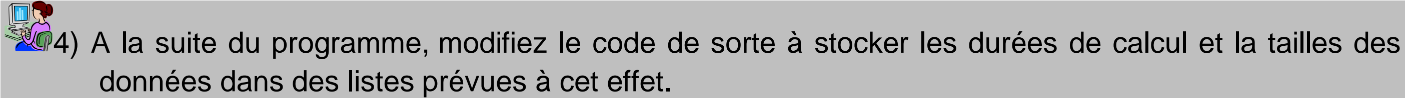
2) A la suite du programme, mettez en place les éléments permettant d'afficher la durée du tri et la taille de la liste traitée

On souhaite vérifier expérimentalement l'ordre de grandeur de la complexité en temps de cet algorithme. Pour cela, on souhaite réaliser le tri sur des tableaux de plus en plus grands, et à chaque fois, évaluer le temps et la tailles des données traitées.

3) A la suite du programme, mettre en place une boucle permettant à chaque itération de :
- augmenter la taille de la liste (en multipliant la liste par 2^i par exemple, ou en produisant une nouvelle liste)
- exécuter le tri insertion sur la nouvelle liste
- évaluer la durée du tri.

Testez votre fonction.

On souhaite maintenant tracer la courbe de la fonction durée=f(taille).



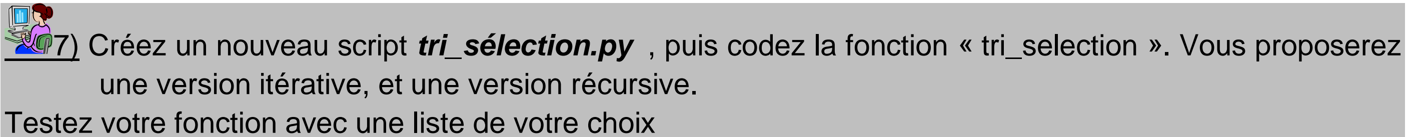
4) A la suite du programme, modifiez le code de sorte à stocker les durées de calcul et la tailles des données dans des listes prévues à cet effet.

5) A la suite du programme, décommentez la ligne permettant de réaliser l'affichage de la fonction durée=f(taille).

6) l'allure de la courbe correspond-elle à l'évolution théorique du temps de calcul vue en cours ?

III. Tri par sélection

Le principe : on prend a plus petite valeur de toutes, et on la met en premier, ensuite en prend la plus petite valeur parmi les valeurs restantes, et on la place en second, et ainsi de suite jusqu'à ce que toutes les valeurs aient été placées correctement.

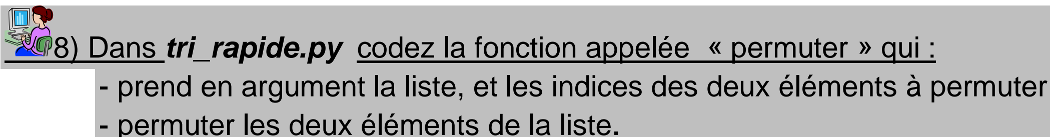


7) Créez un nouveau script ***tri_sélection.py*** , puis codez la fonction « tri_selection ». Vous proposerez une version itérative, et une version récursive.
Testez votre fonction avec une liste de votre choix

IV. Tri rapide

1. Fonction permuter

Le tri rapide nécessite de permuter des éléments dans la liste à trier.



8) Dans ***tri_rapide.py*** codez la fonction appelée « permuter » qui :
- prend en argument la liste, et les indices des deux éléments à permuter
- permuter les deux éléments de la liste.

Testez votre fonction avec une liste de votre choix

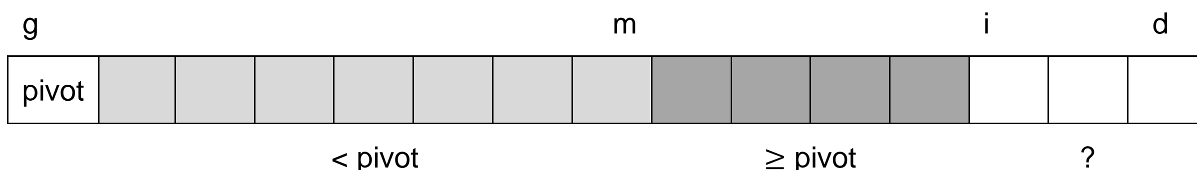
2. Fonction partition

Le succès de l'algorithme de tri rapide repose sur cette opération.

On souhaite organiser le tableau autour d'une valeur 'pivot', de sorte à ce que les éléments à gauche soient plus petits, et les éléments à droite plus grands

Le principe :

- On compare les éléments du tableau T à un élément particulier appelé « pivot »
- Il s'agit d'un travail itératif, à chaque itération, on évalue un nouvel élément du tableau.
- à la i-eme itération, le tableau est sous la forme suivante :



Pivot : est choisi arbitrairement. Ici le premier élément du tableau

g et d : indice de début et fin de tableau (d exclu)

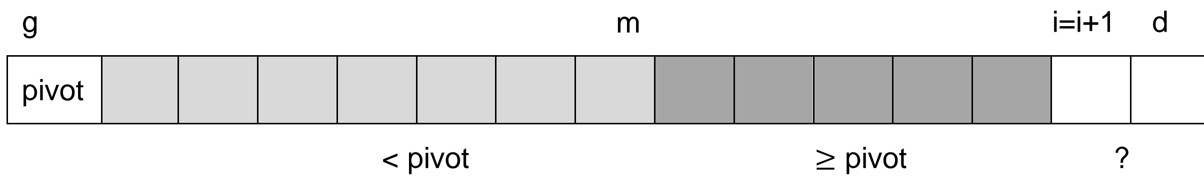
m : indice du dernier élément des valeurs < pivot

i : indice de l'élément à évaluer à la i-eme itération

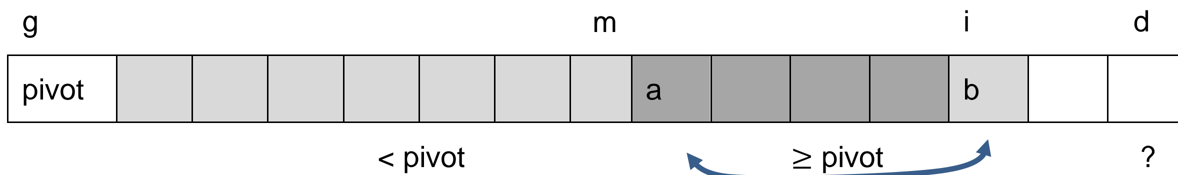
? : éléments qui n'ont pas encore été évalués

se présentent alors deux cas de figure :

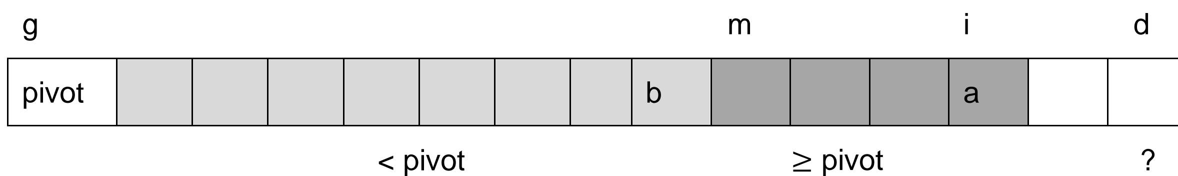
- soit **T[i]** ≥ **pivot** : alors on veut que T[i] fasse partie des ''≥ pivot '' en gris foncé -> on incrémente i, et on passe au suivant



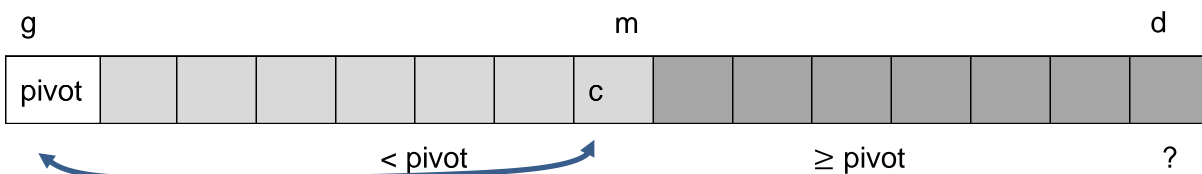
- soit **T[i]** < **pivot** : alors on veut que T[i] fasse partie des ''< pivot'' en gris clair

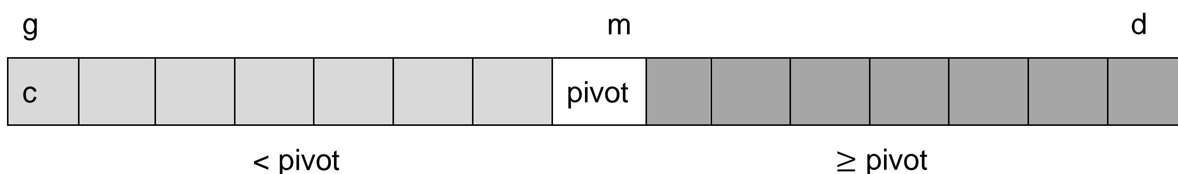


Il suffit alors de permuter T[i] et T[m+1]
Et enfin actualiser m en l'incrémentant



- Enfin on permute T[g] et T[m]





9) remémorez-vous ci-dessous une écriture de l'algorithme de partition

Algorithme 4 partition

Données : T : un tableau de valeurs numériques [0..n-1]
Résultat : le tableau T « partitionné » avec le pivot à sa place définitive, l'indice de la place du pivot

```
partition(T, g, d) :

    pivot  = T [ g ]
    m = g

    Pour i de g à d :
        Si T[ i ] < pivot :
            permuter( T , m+1 , i )
            m=m+1
    Fin Pour
    permuter(T, m , g )
    Renvoyer m
```

10) Dans ***tri_rapide.py*** **,** codez la fonction « partition »

3. *Fonction tri rapide*

L'algorithme de tri rapide est un algorithme récursif. Il est constitué du cas trivial et du cas récursif.
Il s'agit donc de déterminer :

- ce qui constitue le cas récursif :
  Tant qu'il y a plus qu'un élément dans le tableau :
  Segmenter le tableau et obtenir l'indice du pivot m, puis appel récursif de la fonction tri sur chacun des sous tableaux obtenus, c'est-à-dire sur T[g , m-1] et T[m+1 , d]

- ce qui constitue le cas trivial :
  Lorsque le tableau ne contient plus qu'un élément, il est trié. On ne fait rien.

En pseudo-code :

Fonction tri_rapide :
  Si tableau plus long que 1 :
    - m = partition(T, g, d)
    - tri_rapide du sous tableau de gauche
    - tri_rapide du sous tableau de droite

Algorithme 3 tri_rapide

Données : T : un tableau de valeurs numériques [0..n-1]
Résultat : le tableau T, trié par ordre croissant

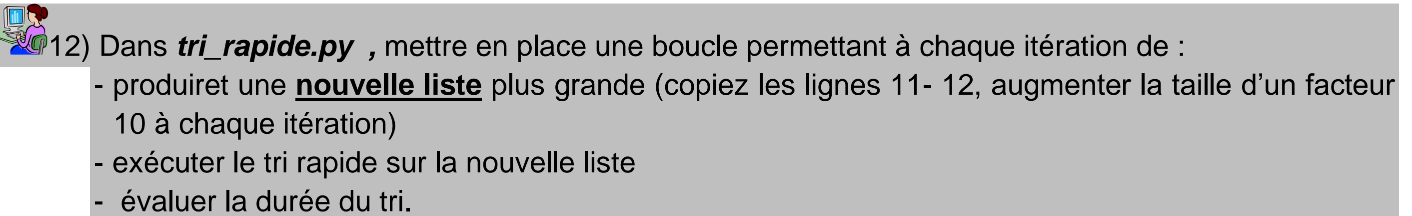
```
tri_rapide(T, g, d) :
    si g < d :
        m = partition(T, g, d)
        tri_rapide(T, g, m-1)
        tri_rapide(T, m+1, d)
    Fin Si
```

➢ *Mise en œuvre*

11) Dans ***tri_rapide.py*** **,** codez la fonction « tri_rapide » . testez votre fonction avec la liste tri_alea.

➢ ***Complexité pour des données désordonnées***

Comme pour le tri insertion, on souhaite vérifier expérimentalement l'ordre de grandeur de la complexité en temps de cet algorithme. Pour cela, on souhaite réaliser le tri sur des tableaux de plus en plus grands, et à chaque fois, évaluer le temps et la tailles des données traitées.

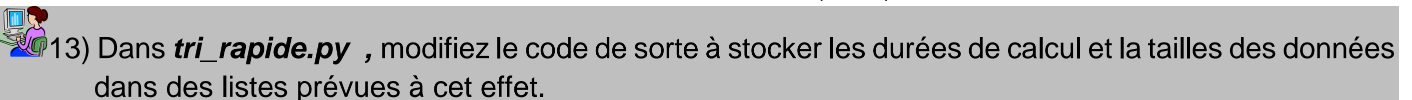

12) Dans ***tri_rapide.py*** **,** mettre en place une boucle permettant à chaque itération de :

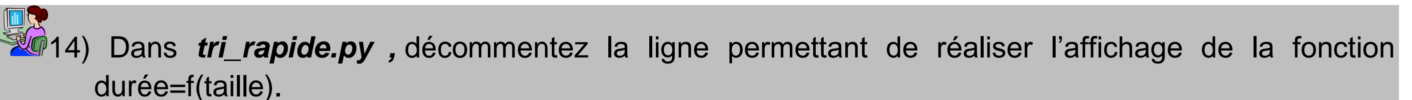
- produiret une **nouvelle liste** plus grande (copiez les lignes 11- 12, augmenter la taille d'un facteur 10 à chaque itération)
- exécuter le tri rapide sur la nouvelle liste
- évaluer la durée du tri.

REMARQUE : **on se limitera à des listes ne dépassant pas 10^6 valeurs**.

Testez votre fonction.

On souhaite maintenant tracer la courbe de la fonction durée=f(taille).

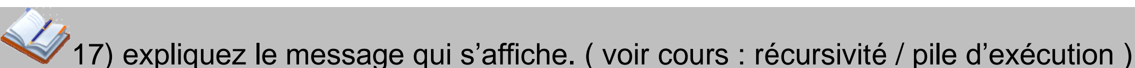

13) Dans ***tri_rapide.py*** **,** modifiez le code de sorte à stocker les durées de calcul et la tailles des données dans des listes prévues à cet effet.

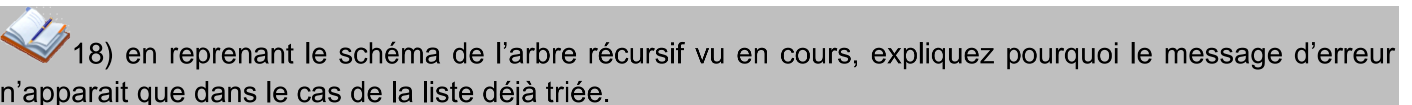

14) Dans ***tri_rapide.py*** **,** décommentez la ligne permettant de réaliser l'affichage de la fonction durée=f(taille).

15) l'allure de la courbe correspond-elle à l'évolution théorique du temps de calcul vue en cours ?

➢ ***Complexité pour des données triées***

16) Dans ***tri_rapide.py*** **,** ajoutez le code permettant d'effectuer la même analyse de la complexité du tri rapide, mais en triant la liste T_ord déjà triée


17) expliquez le message qui s'affiche. (voir cours : récursivité / pile d'exécution)


18) en reprenant le schéma de l'arbre récursif vu en cours, expliquez pourquoi le message d'erreur n'apparait que dans le cas de la liste déjà triée.

Pour éviter le comportement observé précédemment, on propose de limiter le caractère récursif, en introduisant une boucle :

```
def tri_rapide2(T,g,d):
    while g<d:
        m=partition(T,g,d)
        tri_rapide2(T,g,m-1)
        g=m+1
```

19) utilisez le code proposé ci-dessus pour réaliser l'étude de complexité pour des données désordonnées puis triées

20) en vous inspirant de la ligne de code suivante :

```
plt.plot(taille, temps,'r',tailleord, tempsord,'b')
```

Affichez l'allure de l'évolution du temps de calcul dans le cas où les données sont ordonnées ou désordonnées.

15) Expliquez les courbes affichées.

4. Tri fusion

On se propose ici de coder cet algorithme de tri et de l'expérimenter.

a. Mise en œuvre

- ***Fusion***

11) Dans ***tri_fusion.py*** , codez la fonction « fusion» . Testez votre fonction avec une liste de votre choix.

- ***Tri fusion***

11) Dans ***tri_fusion.py*** , codez la fonction « tri_fusion » . Testez votre fonction avec la liste tri_alea.

- ***Complexité pour des données désordonnées***

Comme pour le tri insertion, on souhaite vérifier expérimentalement l'ordre de grandeur de la complexité en temps de cet algorithme. Pour cela, on souhaite réaliser le tri sur des tableaux de plus en plus grands, et à chaque fois, évaluer le temps et la tailles des données traitées.

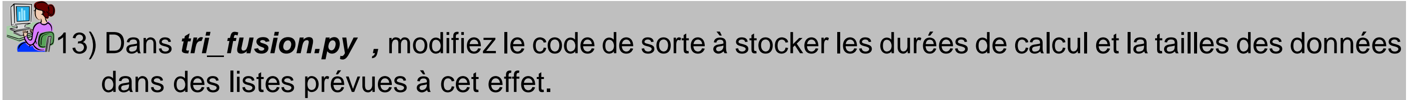
12) Dans ***tri_fusion.py*** , mettre en place une boucle permettant à chaque itération de :

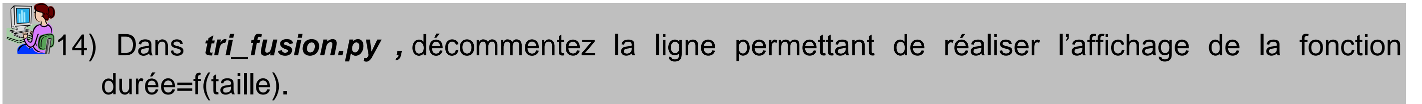
- produisant une nouvelle liste plus grande (augmenter la taille d'un facteur 10 à chaque itération
- exécuter le tri rapide sur la nouvelle liste
- évaluer la durée du tri.

REMARQUE : **on se limitera à des listes ne dépassant pas 10^6 valeurs.**

Testez votre fonction.

On souhaite maintenant tracer la courbe de la fonction durée=f(taille).


13) Dans ***tri_fusion.py*** **,** modifiez le code de sorte à stocker les durées de calcul et la tailles des données dans des listes prévues à cet effet.


14) Dans ***tri_fusion.py*** **,** décommentez la ligne permettant de réaliser l'affichage de la fonction durée=f(taille).

15) l'allure de la courbe correspond-elle à l'évolution théorique du temps de calcul vue en cours ?