1 Algorithmes de tri

1.1 Le problème du tri

Un algorithme de tri est un algorithme qui résout pour toutes ses instances un problème de tri :

- **Entrée :** une séquence (liste, tableau) de n objets $(a_1, a_2, \cdots, a_{n-1}, a_n)$. Ces objets peuvent être des entiers, des chaînes de caractères ... mais il faut qu'on puisse les ordonner par une relation d'ordre total $\leqslant$.
- **Sortie :** un réarrangement $(a'_1, a'_2, \cdots, a'_{n-1}, a'_n)$ de la séquence d'entrée telle que $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_{n-1} \leqslant a'_n$. Ce réarrangement peut être effectué sur place par modification du même conteneur (liste, tableau) de la séquence initiale ou par création d'un nouveau conteneur pour la séquence ordonnée.

Exercice 1

1. Trier à la main la liste de nombres :
   `[16, 30, 14, 56, 5, 0, 31, 26, 36, 80, 95, 0, 27, 85, 78, 78, 77, 73, 22, 47]`
   Décrire l'algorithme utilisé.
2. Le lien ci-dessous permet de visionner une video montrant un enfant de 19 mois réalisant un algorithme de tri pour empiler des boîtes cubiques de tailles différentes :
   `http://www.youtube.com/watch?v=Zybl598sK24`

1.2 Importance en informatique

Le tri est historiquement un problème majeur en informatique pour plusieurs raisons :

- on a souvent besoin de trier des données (notes,noms, photos ...)
- les algorithmes de tri sont des sous-programmes indispensables à de nombreuses applications (gestionnaires de fenêtres graphiques ...) ou programmes (compilateurs)
- la diversité des algorithmes de tri qui ont été développés, présente un intérêt pédagogique dans l'apprentissage de l'algorithmique
- on a démontré que la complexité d'un algorithme de tri de n objets possède une borne inférieure (en $n\log_2(n)$) et on connaît des algorithmes de tri asymptotiquement optimaux.

Dans ce chapitre, pour simplifier, on se cantonnera au tri de tableaux de listes d'entiers et sans précision supplémentaire on effectuera toujours un tri dans l'ordre croissant. On parlera indifféremment de liste ou de tableaux, sachant que ce qu'on désigne par tableau d'entier dans les ouvrages d'algorithmiques correspond à une liste en Python.

Exercice 2

1. Pour la découverte et la comparaison de plusieurs algorithmes de tri on pourra consulter les sites :
   - `fr.wikipedia.org/wiki/Algorithme_de_tri`
   - `http://interstices.info/jcms/c_6973/les-algorithmes-de-tri`
   - `http://www.sorting-algorithms.com/`
2. Quelques références bibliographiques :
   - *Manuel ISN* de Gilles Dowek chez Eyrolles, Chapitre 21 page 264.
   - *Algorithmique* de Cormen-Leiserson-Rivest-Stein chez Dunod, Partie 1 Chapitre 2 et Partie 2 Introduction page 139.

1.3 Les fonctions de tri en Python

Soit une liste `t` (les tableaux de Python) d'objets comparables (entiers, caractères ...) , Python propose deux façons de la trier :

- ☞ `sorted(t)` retourne une nouvelle liste triée dans l'ordre croissant ;
- ☞ `t.sort()` trie sur place la liste `t` par ordre croissant.

Dans les deux cas, on peut obtenir un ordre décroissant avec le paramètre optionnel `reverse = True`.
Plus généralement on peut personnaliser le tri avec le paramètre optionnel `key = fonction` : les éléments du tableau seront triés suivant leur image par `fonction`.

```
>>> t = ['b','c','a']
>>> m = sorted(t)
>>> m
['a', 'b', 'c']
>>> t
['b', 'c', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c']
>>> t.sort(reverse=True)
>>> t
['c', 'b', 'a']
>>> from random import randint
>>> alea = [randint(-100, 100) for _ in range(10)]
>>> alea
[-16, -40, 79, -83, 85, -89, -96, 56, 6, 72]
>>> dec = sorted(alea, reverse=True)
>>> dec
[85, 79, 72, 56, 6, -16, -40, -83, -89, -96]
>>> def f(x):
...     return x**2
...
>>> carre = sorted(alea, key=f)
>>> carre
[6, -16, -40, 56, 72, 79, -83, 85, -89, -96]
```

2 Le tri par sélection

2.1 Algorithme

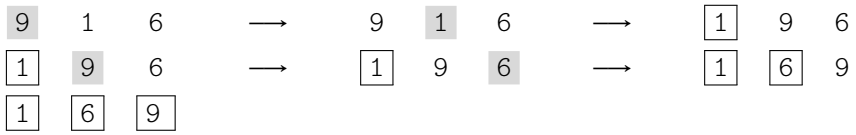
Considérons un joueur de cartes qui tient dans sa main les cartes en désordre de sa donne. On suppose qu'on a défini sur les cartes une relation d'ordre total (Carreau $\leqslant$ Coeur $\leqslant$ Pique $\leqslant$ Trêfle puis par valeur pour des cartes de la même famille). Pour classer ses cartes dans l'ordre croissant de la partie gauche à la droite de sa main, le joueur peut appliquer l'algorithme de tri par sélection :

Etape 1– Il divise mentalement sa main entre partie gauche triée (vide au début) et partie droite non triée.

Etape 2– Il sélectionne la plus petite carte de la partie non triée et la place à la suite des cartes de la partie triée (donc en première position pour la première sélection).

Etape 3– Il recommence l'opération précédente tant qu'il lui au moins deux cartes dans la partie non triée.

Voici un exemple de déroulement du tri par sélection de la liste d'entiers `[9, 1, 6]`.



Exercice 3

Appliquer l'algorithme de tri par sélection à la main pour trier les listes d'entiers :

1. `[72, 29, 39, 59, 17, 54, 77, 79, 21, 6]`
2. `[22, 62, 63, 70, 98, 97, 9, 53, 2, 0]`
3. `[74, 82, 51, 22, 89, 49, 20, 12, 93, 84]`

2.2 Pseudo-code et correction de l'algorithme

L'algorithme de tri par sélection peut se coder sous la forme d'une procédure de paramètre *liste*, qui est la liste d'entiers à trier.
Le tri s'effectue sur place, ainsi la liste passée en paramètres est modifiée et contiendra la liste des valeurs initiales triées après application de la procédure. Celà ne pose pas de problème en Python car les paramètres sont passés par référence et les listes sont des objets mutables.
Le site Interstices propose une page dédiée aux algorithmes de tri (voir exo 2) avec une application illustrant le fonctionnement des algorithmes de tri les plus courants dont le tri par sélection.

L'algorithme imbrique deux boucles Pour :

- la première parcourt la liste entre la première position 0 et l'avant-dernière $n-2$: son compteur i indique la première position de la partie non triée de la liste avant le tour de boucle et celle-ci est mémorisée dans la variable *minimum*.
- la seconde boucle imbriquée dans la première, parcourt la partie non triée de la deuxième position $i+1$ de cette partie à la dernière $n-1$ et recherche par comparaison l'indice du minimum de la partie non triée en stockant l'indice du minimum temporaire dans la variable *minimum*.
- quand on sort de la seconde boucle, on revient dans le tour de boucle de la première avec la position du minimum de la partie non triée. Si l'indice du minimum n'est plus i, on échange alors le premier élément de la partie non triée (position i) avec l'élément minimum de cette partie qui est en position *minimum*. On utilise alors une fonction `echange(p,q)` qui permute les éléments en positions `p` et `q` d'une liste.
- la première boucle continue jusqu'à l'avant dernière position de la liste. A la fin de ce tour de boucle, la partie triée contient les $n-1$ plus petits éléments et la partie non triée réduite à un élément contient nécessairement le plus grand élément de la liste. Ainsi la liste initiale est triée.

La correction de l'algorithme se prouve en montrant que l'algorithme préserve l'invariant de boucle suivant : au début de chaque tour i de la boucle Pour externe, la partie triée (la sous-liste `liste[0,...,i-1]` qui est vide avant la première itération) contient les i plus petits éléments de la liste initiale, triés dans l'ordre croissant. Après le tour de boucle $n-2$ (soit $n-1$ tours de boucles de 0 à $n-2$), la sous-liste `liste[0,...,n-2]` avec les $n-1$ plus petits éléments de la liste est triée et l'élément `liste[n-1]` est forcément l'élément maximum et la liste est triée.

Algorithme de tri par sélection

tri_selection(*liste*)
- n = **longueur**(*liste*)
- **Pour** i **allant de** 0 **à** $n-2$
  - $minimum = liste[i]$
  - $indexminimum = i$
  - **Pour** j **allant de** $i+1$ **à** $n-1$
    - **Si** $liste[j] < minimum$
      - $minimum = liste[j]$
      - $indexminimum = j$
  - **Si** $indexminimum != i$
    - **echanger**($liste[i]$,$liste[indexminimum]$)

2.3 Implémentation

Exercice 4

1. Compléter le programme suivant avec une fonction `echange` puis une fonction `tri_selection(liste)` .

```
from random import randint
import time

def timetest(fonction):
    """exécute la fonction et affiche son temps d'exécution """
    def fonction_modifiee(*args,**kargs):
        tps_avant = time.time()
        fonction(*args,**kargs)
        tps_apres = time.time()
        print("Le temps d'exécution de la fonction est de {:10.3e} s"
              .format(tps_apres-tps_avant))
    return fonction_modifiee

@timetest
def tri_selection(liste):
    """applique l'algorithme de tri par sélection à liste"""
    .....

def test(f,n,liste1,liste2,liste3):
        """fonction de test de la fonction de tri
        sur trois type d'entrées. Oon travaillera sur des copies physiques de
             liste1, liste2, liste3 pour ne pas les modifier """
        l1,l2,l3 = liste1[:],liste2[:],liste3[:]
        print("Test sur une liste de {} entiers aléatoires entre 0 et {}".format
             (n,5*n).center(100,'*'))
        f(l1)
        print("Test sur une liste de {} entiers déjà triés".format(n).center
             (100,'*'))
        f(l2)
        print("Test sur une liste de {} entiers triés dans l'ordre inverse".
             format(n).center(100,'*'))
        f(l3)
```

```
29
30 #programme principal
31 n = int(input('Entrez la taille souhaitée pour les listes d\'entiers: \n'))
32 #initialisation des listes de test
33 liste1,liste2,liste3 = [randint(0,5*n) for i in range(n)],[i for i in range(n)
       ],[n-i for i in range(n)]
34 print('Test du tri par sélection')
35 test(tri_selection,n,liste1,liste2,liste3)
```

2. En utilisant la fonction `test` estimer le temps d'exécution de la fonction `tri_selection` en l'appliquant à des listes d'entiers de trois types (aléatoires, déjà dans l'ordre ou dans l'ordre inverse) pour des tailles 10, 50, 500, 1 000, 10 000.

2.4 Analyse de complexité

Pour analyser la **complexité** (ou efficacité) d'un algorithme on évalue le nombre d'instructions élémentaires effectuées en fonction de la **taille de l'entrée** (ici le nombre n d'entiers à trier). **On peut faire l'hypothèse que chaque ligne du pseudo-code de l'algorithme prend un temps constant à chaque fois qu'elle est exécutée**. De plus pour simplifier, nous considérerons que toutes les instructions (tests, affectations) prennent le même temps. Ce temps d'exécution d'une instruction élémentaire est alors constant (il peut varier selon les machines) c'est pourquoi la complexité de l'algorithme peut se mesurer en nombre d'instructions élémentaires effectués.
Pour l'algorithme de tri par sélection on peut compter les tests de boucles, les affectations de variables, les tests de comparaison et les échanges de variables.
Enfin, la complexité peut varier pour des instances de même taille : pour l'algorithme de tri par sélection, le **meilleur des cas** est celui où la liste est déjà triée et le pire celui où elle est triée dans l'ordre inverse.
En général, on évalue la complexité dans **le pire des cas** et dans **le cas moyen** (liste d'entiers aléatoires).
Soit n la taille de la liste d'entiers à laquelle on applique l'algorithme de tri par sélection :

- **Nombre de tests de comparaisons :**
  La boucle Pour externe contrôle $n-1$ exécutions de la boucle Pour interne, chacune réalise $n-1-i$ comparaisons pour i variant entre 0 et $n-2$. Le nombre de tests de comparaisons dans la boucle interne ne dépend pas de l'instance. C'est le même dans le meilleur des cas (liste triée) ou le pire des cas (liste triée dans l'ordre inverse).
  On peut le calculer :

$$\sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1}1\right)=\sum_{i=0}^{n-2}n-1-i$$

  on fait le changement de variable $k=n-1-i$

$$\sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1}1\right)=\sum_{k=1}^{n-1}k$$

  on applique la formule de la somme des termes consécutifs d'une suite arithmétique

$$\sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1}1\right)=\frac{n(n-1)}{2}$$

  Il faut ajouter les $n-1$ tests de comparaison entre i et $minimum$ à la fin d'un tour de boucle externe.
  Au total il y a donc : $\frac{n(n-1)}{2}+n-1$ tests de comparaisons quelle que soit l'instance.

- **Nombre de tests de boucles :**
  Dans une boucle Tant Que ou Pour, le test d'entrée dans la boucle (n'oublions pas qu'une boucle Pour est une boucle Tant Que avec un test et une affectation qui incrémente le compteur) est exécuté une fois de plus que le nombre de tours de boucles (le dernier test est faux et permet la sortie de boucle).
  Le compteur de la boucle Pour externe varie de 0 à $n-2$, elle s'exécute donc $n-1$ fois et son test n fois.
  A chaque tour i de la boucle externe, le compteur de la boucle Pour interne varie de $i+1$ à $n-1$ donc elle s'exécute $n-1-(i+1)+1 = n-1-i$ fois et le test de boucle s'exécute $n-i$ fois.
  Le test de la boucle interne s'exécute $\sum_{i=0}^{n-2}(n-i) = \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1$ fois.
  Au total il y a donc : $n + \frac{n(n+1)}{2} - 1$ tests de boucles (externe +interne), quelle que soit l'instance.

- **Nombre d'échanges :**
  Dans le pire des cas on réalise $n-1$ échanges « **echanger**($liste[i], liste[minimum]$) » dans la boucle externe (et 0 dans le meilleur des cas si la liste est déjà triée).

- **Nombre d'affectations :**
  Il y a une affectation qu'on peut négliger avant la boucle Pour externe , $n-1$ affectations « $minimum = i$ » dans la boucle externe et au plus $\frac{n(n-1)}{2}$ (ou au mieux 0) affectations « $minimum = j$ » dans la boucle interne.
  Le nombre d'affectations varie donc entre n et $n + \frac{n(n-1)}{2}$.

En additionnant tous les décomptes précédents on observe que dans le meilleur ou le pire des cas la complexité du tri par sélection de n entiers peut s'écrire sous la forme $an^2 + bn + c$ (les constantes sont plus petites pour le meilleur des cas).
Lorsque n devient grand, le facteur en n^2 devient prépondérant, on dit que la complexité du tri par sélection est quadratique.

Exercice 5

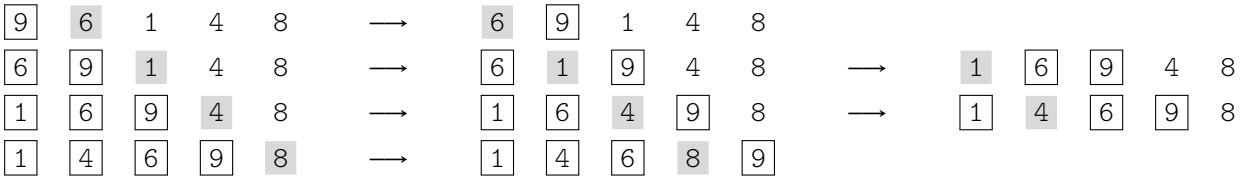
Pour des éléments sur la complexité du tri par sélection on pourra visiter les pages web :

- `http://fr.wikipedia.org/wiki/Tri_par_sélection`
- `http://www.sorting-algorithms.com/selection-sort`

2.5 D'autres algorithmes de tri par comparaison

Exercice 6 *Tri par insertion*

Reprenons l'exemple du joueur qui doit trier les cartes de sa donne. Si les cartes de la donne sont posées en pile sur la table, l'algorithme de tri par insertion consiste en prendre chaque carte dans l'ordre de la donne et à l'insérer à sa place dans la liste des cartes déjà piochées.
Autrement dit si on dispose d'une liste de n entiers dont les k premiers éléments sont triés, en insérant le $k+1$-ème élément à sa place parmi les k premiers, on obtient une liste avec $k+1$ éléments triés. Au départ, le premier élément peut être considéré comme trié donc si on répète cette opération d'insertion $n-1$ fois, on peut faire un *tri sur place* de la liste.
Plusieurs méthodes sont possibles pour insérer le $k+1$-ème élément à sa place, la plus simple est sans doute de procéder par échanges successifs d'éléments adjacents [1] :



1. Faire tourner l'application *Tri par insertion* sur la page dédiée du site Interstices puis écrire en pseudo-code un algorithme de tri par insertion (dans l'ordre croissant) d'une liste d'entiers.

2. Programmer en Python une fonction `tri_insertion` pour compléter le programme `trieleve.py` et tester cette fonction sur des listes d'entiers aléatoires de tailles variables.

   Avec la fonction `test`, comparer les performances des fonctions `tri_insertion` et `tri_selection` sur des listes d'entiers aléatoires de petite (moins de 30) , moyenne (100 ou 500) ou grande taille (1 000, 5 000, 10 000) puis sur des listes d'entiers triées dans l'ordre croissant ou décroissant.

3. Si l'on interrompt l'exécution de l'algorithme du tri par sélection après k étapes, la sous-liste des k premiers éléments est-elle celle de la liste triée finale ? Et pour le tri par insertion ?

4. Évaluer la complexité du tri par insertion dans le pire des cas (liste initiale triée dans l'ordre inverse) et le meilleur des cas (liste déjà triée).

5. Pour la complexité du tri par insertion, on pourra consulter les pages web cités ci-dessus et les pages :

   - `http://www.sorting-algorithms.com/insertion-sort`
   - `http://interstices.info/jcms/c_6973/les-algorithmes-de-tri`

   Le chapitre 2 du livre de référence *Algorithmique* de Cormen donne un calcul détaillé.

6. Pour la comparaison des performances des algorithmes de tri par sélection ou par insertion :
   `http://www.sorting-algorithms.com/`

Exercice 7 *Tri par bulles, énoncé extrait du manuel ISN de Gilles Dowek*

L'algorithme du tri par bulles consiste à trier sur place dans l'ordre croissant une liste d'entiers en ne s'autorisant qu'à échanger deux éléments consécutifs.

> *Tant qu'on a trouvé deux éléments consécutifs dans le désordre lors du dernier parcours de la liste, on effectue un nouveau parcours en permutant deux éléments consécutifs s'ils sont dans le désordre. Ainsi à la fin de chaque parcours, le plus grand élément parmi ceux qui n'étaient pas encore triés est remonté à sa place à la fin de la liste, comme une bulle dans du champagne . . .*

1. Tableau d'Adrien Lalauze

Voici un exemple :

3	5	2	1	8	9	7	3	tableau de départ
3	2	1	5	8	7	3	9	5 et 9 remontent
2	1	3	5	7	3	8	9	3 et 8 remontent
1	2	3	5	3	7	8	9	2 et 7 remontent
1	2	3	3	5	7	8	9	5 remonte

1. Effectuer à la main un tri à bulles de la liste :
`[72, 39, 29, 59]`

2. Faire tourner l'application *Tri par bulles* sur la page dédiée du site Interstices puis écrire en pseudo-code un algorithme de tri par insertion (dans l'ordre croissant) d'un tableau d'entiers.

3. Programmer en Python une fonction `tri_bulle` pour compléter le programme `trieleve.py` et tester cette fonction sur des listes d'entiers aléatoires de tailles variables.
Avec la fonction `test`, comparer les performances des fonctions `tri_bulle`, `tri_insertion` et `tri_selection` sur des listes d'entiers aléatoires de petite (moins de 30) , moyenne (100 ou 500) ou grande taille (1 000, 5 000, 10 000) puis sur des listes d'entiers triées dans l'ordre croissant ou décroissant.

4. Quel est la complexité (en nombre de comparaisons et d'échanges) du tri à bulles sur une liste de n entiers déjà triée ? (pour une implémentation comme celle décrite sur `Wikipedia`, pas celle du site d'Interstices ou dans le livre de Cormen).
Était-ce le cas pour le tri par sélection ?

5. Les listes sur lesquels le tri à bulles est le moins efficace sont celles qui sont triées dans l'ordre décroissant, par exemple : $[n, n-1, \cdots, 2, 1]$.

   a. Si l'on commence par essayer de placer le nombre n à la position correcte, combien de permutations sont nécessaires pour y arriver ? Ensuite, combien de permutations sont nécessaires pour placer $n-1$ au bon endroit ? Et le nombre $n-2$?

   b. Émettre une conjecture sur le nombre total de permutations nécessaires pour trier une liste de taille n rangée initialement en ordre décroissant.

   c. Démontrer cette conjecture (directement avec une formule de sommation classique ou par récurrence).

Exercice 8 *Tri par insertion de Shell*

Le tri de Shell est une amélioration du tri par insertion qui procède ainsi sur une liste L d'entiers :

- **Etape 1** On choisit une suite (p_n) d'entiers positifs appelés pas.
  Nous choisirons classiquement, la suite définie par $\begin{cases} p_0 = 1 \\ p_{n+1} = 3p_n + 1 \end{cases}$.
- **Etape 2** On calcule le plus grand terme p_n qui est inférieur à la taille de notre liste
- **Etape 3** On trie alors par insertion chaque sous-liste d'éléments de L séparés de p_n positions. On applique donc p_n fois (pour départ allant de 0 à $p_n - 1$) une procédure `tri_insertion_pas(liste,depart,pas)` qui trie par insertion une sous-liste de L d'éléments choisis toutes les p_n positions à partir de la position départ.
- **Etape 4** On recommence ensuite avec p_{n-1} et ainsi de suite par une boucle descendante jusqu'à $p_0 = 1$. A cette étape on applique 1 fois la procédure `tri_insertion_pas(liste,0,1)` qui trie par insertion la liste. Mais ce n'est pas la liste initiale, c'est une liste affinée et presque déjà triée par application successives des procédures `tri_insertion_pas(liste,depart,pas)`.

A la dernière étape,on ne trie pas par insertion la liste initiale, mais une liste affinée et presque déjà triée par application successives des procédures `tri_insertion_pas(liste,depart,pas)`. Or le tri par insertion est de complexité linéaire sur les listes presque triées ... Le tri de Shell remplace ainsi les nombreux petits bonds du tri par insertion classique (permutation entre deux éléments successifs qui ne sont pas triés) par de moins nombreux mais plus grands bonds (permutations entre deux éléments non triés séparés d'un pas).

Donnons un exemple de tri de Shell appliqué à [7,4,10,8,6,5,9,3,2,3] :
La liste est de taille 10 donc on commence avec le pas $p_1 = 4$

- On trie d'abord par insertion une première sous-liste de pas 4 :

  [**7**,4,10,8,**6**,5,9,3,**2**,3] $\longrightarrow$ [**6**,4,10,8,**7**,5,9,3,**2**,3] $\longrightarrow$ [**6**,4,10,8,**2**,5,9,3,**7**,3] $\longrightarrow$ [**2**,4,10,8,**6**,5,9,3,**7**,3]

- Puis une seconde sous-liste de pas 4 :

  [2,**4**,10,8,6,**5**,9,3,7,**3**] $\longrightarrow$ [2,**4**,10,8,6,**3**,9,3,7,**5**] $\longrightarrow$ [2,**3**,10,8,6,**4**,9,3,7,**5**]

- Puis une troisième sous-liste de pas 4 :

  [2,3,**10**,8,6,4,**9**,3,7,5] $\longrightarrow$ [2,3,**9**,8,6,4,**10**,3,7,5]

- Puis une quatrième sous-liste de pas 4 :

  [2,3,9,**8**,6,4,10,**3**,7,5] $\longrightarrow$ [2,3,9,**3**,6,4,10,**8**,7,5]

- On a épuisé toutes les sous-listes de pas 4, on passe aux sous listes de pas $p_0 = 1$, il n'y en a qu'une à laquelle on applique un tri par insertion classique :

  [2,3,9,**3**,6,4,10,**8**,7,5] $\longrightarrow \cdots \longrightarrow$ [2,3,3,4,5,6,7,8,9,10]

1. Ecrire en Python la procédure `tri_insertion_pas(liste,depart,pas)` utilisée pour trier par insertion une sous-liste d'éléments de liste espacés de pas et dont le premier élément est en position départ.

2. Ecrire une procédure `tri_shell(liste)`.

3 Le tri par fusion

Pourquoi un autre algorithme de tri ? Parce que les algorithmes de tri par sélection, par insertion ou par bulles sont de complexité quadratique et sont donc lents lorsque le nombre de données à trier (la taille de l'instance) est grand.
L'algorithme de tri par fusion est beaucoup plus rapide pour les instances de grande taille et sa complexité se rapproche de la borne inférieure théorique de complexité des algorithmes de tri.
L'algorithme de tri par fusion se programme naturellement de façon récursive mais pour évaluer sa complexité plus simplement, nous en donnerons d'abord une version itérative.

3.1 Algorithme de tri par fusion version itérative

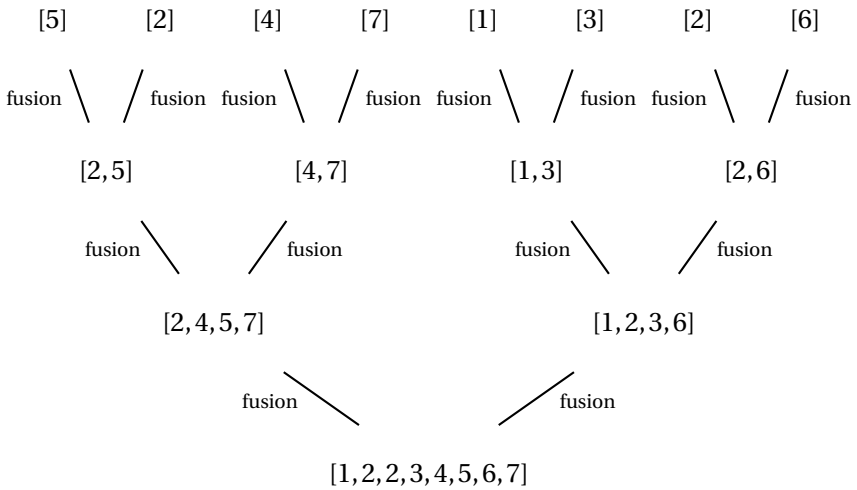
Pour simplifier, on travaillera uniquement sur des listes d'entiers dont la taille est une puissance de 2 du type 2^p. Si la liste est de taille n quelconque on peut toujours la compléter avec des entiers beaucoup plus grands (des *sentinelles*) jusqu'à obtenir une liste de taille 2^p qui est la puissance de 2 immédiatement supérieure. Après application de l'algorithme de tri fusion, les *sentinelles* seront forcément à la fin et il suffira de les supprimer pour retrouver notre liste initiale triée.

A titre d'exemple on veut trier dans l'ordre croissant la liste non triée d'entiers $[5,2,4,7,1,3,2,6]$ de taille 2^3.

- Chaque élément de la liste constitue un segment de la liste de taille 1 qui est ordonné puisqu'il ne comporte qu'un seul élément.
- **Tour de boucle 1 :** on regroupe les segments deux par deux et on **fusionne**, l'une après l'autre, ces quatre paires de segments en des segments ordonnés de taille 2. On obtient alors quatre segments ordonnés de taille 2.
- **Tour de boucle 2 :** On recommence la fusion des quatre segments ordonnés de taille 2, regroupés deux par deux, en deux segments ordonnés de taille 4.
- **Tour de boucle 3 :** On recommence la fusion des deux segments ordonnés de taille 4, regroupés deux par deux, en un segment ordonné de taille 8.
- On arrête alors la fusion puisqu'on a obtenu un unique segment ordonné de taille $8 = 2^3$. C'est la liste initiale triée.

On remarque qu'on a effectué 3 tours de boucles car la taille de l'instance à trier était $8 = 2^3$. On illustre ci-dessous l'exemple traité avec un arbre.

Si la liste initiale est de taille 2^p on effectue p tours de boucle. Au tour de boucle i (avec i variant entre 0 et $p-1$) la fonction de fusion est appliquée à 2^{p-i} segments ordonnés qu'elle fusionne en 2^{p-i-1} segments ordonnés. Au dernier tour d'indice $p-1$, la fonction de fusion est appliquée à $2^{p-(p-1)} = 2$ segments ordonnés qu'elle fusionne en $2^{p-(p-1)-1} = 1$ segment ordonné qui est la liste initiale triée.



Exercice 9

Appliquer à la main l'algorithme de tri par fusion aux listes d'entiers suivantes :

1. [17,4,18,16,6,5,2,13]
2. [11,6,6,12,17,10,16,20]
3. [5,1,14,18,14,1,7,4]
4. [7,10,7,16,6,5,17,15]

3.2 Implémentation de la fonction fusion

Exercice 10

En appliquant à la main l'algorithme de tri par fusion, on remarque que le coeur de l'algorithme est la fonction de fusion qui fusionne deux segments ordonnés de taille 2^q en un segment ordonné deux fois plus grand, de taille 2^{q+1}.

1. On donne ci-dessous un programme en Python qui fusionne deux listes triées de taille 8 `segmentA` et `segmentB`, en une liste `segmentC` de taille 16. Les deux segments A et B sont d'abord juxtaposés pour former une liste de taille 16.
   Expliquer le rôle des affectations des lignes 11 et 12 puis celui des instructions du bloc de la boucle `for` entre les lignes 14 et 19.

```
1  from random import randint
2  segmentA = [randint(0,20) for i in range(8)]
3  segmentA.sort()
4  print("Segment A: ",segmentA)
5  segmentB = [randint(0,20) for i in range(8)]
6  segmentB.sort()
7  print("Segment B: ",segmentB)
8  liste = segmentA+segmentB
9  print("Juxtaposition des sements A et B : ",liste)
10 segmentC = []
11 x = 0
12 y = 8
13 for i in range(0,16):
14     if (x<=7 and y<=15 and liste[x]<=liste[y]) or (y==16):
15         segmentC.append(liste[x])
16         x = x+1
17     else:
18         segmentC.append(liste[y])
19         y = y+1
20 print("SegmentC fusion des segments A et B : ",segmentC)
```

fusionexemplecours.py

2. Ecrire en Python une fonction `fusion(liste,p,q,r)` dont les paramètres sont : une `liste` d'entiers, l'indice `p` du premier élément d'un segment ordonné A de `liste`, l'indice `q` du dernier élément du segment A (`A=liste[p,..,q]`) et l'indice `r` du dernier élément d'un segment ordonné B de `liste` contigu à A (`B=liste[q+1,..,r]`). La fonction `fusion` retourne un segment ordonné de `liste` obtenu par fusion puis tri des éléments des segments A et B.

3.3 Implémentation de l'algorithme de tri par fusion

Exercice 11

1. Ecrire une fonction `estpuissance2(n)` qui retourne l'exposant p tel $n = 2^p$ ou None si n n'est pas une puissance de 2.
2. Compléter le programme suivant avec une procédure `tri_fusion(liste)` qui effectue un tri sur place d'une liste d'entiers de taille 2^p avec l'algorithme de tri par fusion version itérative.

```
from random import*

def estpuissance2(n):
    """Teste si un entier n est une puissance de 2
    et retourne None ou son exposant p tel que n=2**p"""
    .....

def fusion(liste,p,q,r):
    """fonction fusion qui fusionne en un seul segment ordonné,
    deux segments ordonnés de liste, le premier allant de la
    position p à la position q, le second de q+1 à r"""
    segment = []
    x = p
    y = q+1
    for i in range(0,r-p+1):
        if (x<=q and y<=r and liste[x]<=liste[y]) or (y==r+1):
            segment.append(liste[x])
            x = x+1
        else:
            segment.append(liste[y])
            y = y+1
    return segment

def tri_fusion(liste):
    """algorithme de tri par fusion (version itérative)
    d'une liste de taille 2^n"""
    .......

#programme principal (tests)
taille = int(input('Entrez la taille de la liste : \n'))
liste = [randint(1,101) for i in range(taille)]
print("liste initiale non triée : \n",liste)
tri_fusion(liste)
print('La liste triée : \n',liste)
```

tri_fusion.py

3.4 Complexité de l'algorithme de tri par fusion

On donne ci-dessous un code possible pour la procédure `tri_fusion` qui implémente un algorithme de tri par fusion itératif d'une liste d'entiers (elle utilise la fonction `fusion` donnée dans l'exercice 10).
Les commentaire précèdent les instructions commentées.

```
1  def tri_fusion(tab):
2      """tri par fusion (version itérative)
3      d'un tableau tab de taille 2^n.
4      Retourne None si la taille n'est pas une puissance de 2"""
5      #longueur du tableau
6      n = len(tab)
7      expo = estpuissance2(n)
8      #expoest le log binaire de n
9      #si expo est None (evalué à False) on retourne None
10     if not expo:
11         return None
12     #sinon n est une puissance de 2
13     #on paut faire le tri fusion
14     #initialisation de la taille des segments
15     #on boucle sur la longueur des segments
16     #tant que taillesegment<n
17     #ce qui fera expo=log2(n) tours
18     taillesegment = 1
19     while taillesegment<n:
20         #boucle interne sur les segments de longueur taillesegment
21         #on fusionne 2 par 2 les segments
22         #de longueur taillesegment qui sont consecutifs
23         #initialisation des index p de debut du premier segment
24         #q de fin du premier segment
25         #et r de fin du second segment
26         p,q,r = 0,taillesegment-1,2*taillesegment-1
27         #tant que l'index de fin du second segment
28         #n'est pas au bout du tableau
29         while r<=n-1:
30             #la partie de tab comprise entre les indices
31             #p et r est fusionnée
32             tab[p:r+1] = fusion(tab,p,q,r)
33             #mise à jour des index des 2 segments consécutifs
34             #on fait un saut d'index de longueur 2*taillesegment
35             #car on passe aux 2 segments consécutifs suivants
36             p,q,r = p+2*taillesegment,q+2*taillesegment,r+2*taillesegment
37         #tous les segments de longueur taillesegment ont été fusionnés
38         #on incrémente (fois 2) taillesegment
39         taillesegment *= 2
```

On va évaluer grossièrement la complexité du tri par fusion appliqué à une instance (liste à trier) de taille n. Pour plus de détails on pourra se référer à l'ouvrage de référence *Algorithmique* de Cormen Chapitre 2 pages 31 à 34.
Considérons les imbrications des principaux blocs d'instructions :

- le bloc d'instructions de la boucle `while` externe (ligne 19) est exécuté $\log_2(n)$ fois car son test porte sur s qui est initialisée à $s = 1$ et multiplié par 2 tant qu'il ne dépasse pas n c'est-à-dire que le nombre de tours de boucle est le logarithme binaire de n.

- le bloc d'instructions de la boucle `while` interne (ligne 29) est exécuté $\frac{n}{2s}$ fois car son test porte sur r qui est initialisé à $2s-1$ et qui augmente de $2s$ tant qu'il ne dépasse pas n.
- dans le bloc d'instructions de la boucle `while` interne, la fonction `fusion` (ligne 32) contient une boucle `for` dont le bloc d'instructions (voir code de la fonction `fusion` dans l'exercice 10) est exécuté $r-p+1=2s$ fois.

Le principal bloc d'instructions de l'algorithme est le bloc d'instructions de la fonction `fusion`. Si on déplie les imbrications successives décrites ci-dessus, il est exécuté environ : $\log_2(n)\times\frac{n}{2s}\times 2s=\log_2(n)\times n$ fois.
En additionnant les différents coûts en instructions, on peut dire que lorsque n devient grand, la complexité du tri par fusion en fonction de la taille n de l'instance est de l'ordre de $\log_2(n)\times n$**.**

Exercice 12

Rajouter les fonctions `fusion` et `tri_fusion` dans le module `trieleve.py`.

1. Avec la fonction `test`, comparer les performances des fonctions `tri_bulle`, `tri_insertion`, `tri_selection` et `tri_fusion` sur des listes d'entiers aléatoires de petite (2^5), moyenne (2^9) ou grande taille (2^{10}, 2^{12}) puis sur des listes d'entiers triées dans l'ordre croissant ou décroissant. Attention notre algorithme de tri par fusion ne fonctionne que pour des tailles du type 2^p !!!
2. Les rapports de performances entre les algorithmes de tri obtenus expérimentalement correspondent-t-ils aux rapports théoriquement attendus entre les algorithmes de complexité en n^2 et l'algorithme de tri fusion de complexité en $\log_2(n)\times n$.

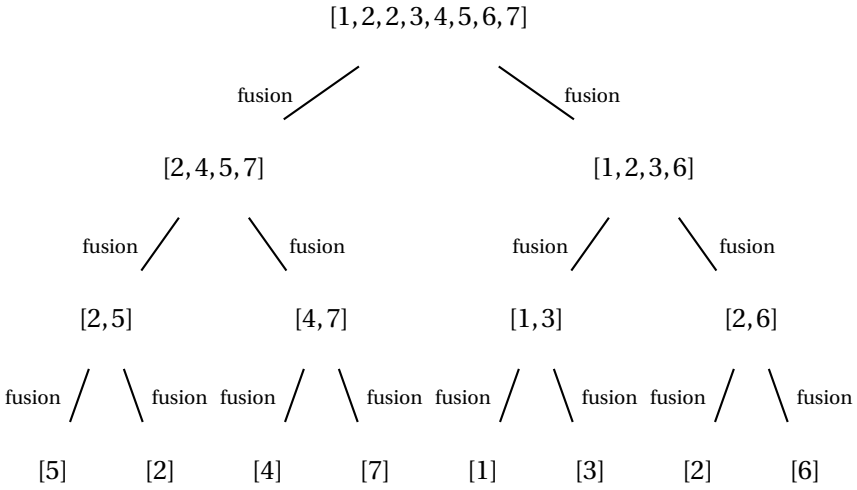
3.5 Algorithme de tri par fusion version récursive

3.5.1 Pseudo-code et implémentation

Le tri par fusion d'une liste de n entiers peut se programmer de façon récursive en s'appliquant au segment de liste situé entre deux positions p et r :

- si le segment est de taille 1, il est trié et on le retourne ;
- sinon on le divise en deux sous-segments entre les positions p et $q=\lfloor(p+r)/2\rfloor$ et $q+1$ et r :
  - on applique récursivement l'algorithme de tri par fusion au premier puis au second sous-segment ;
  - puis on fusionne les deux sous-segments triés.

Si on reprend l'exemple du tri par fusion de la liste $[5,2,4,7,1,3,2,6]$, pour visualiser le déroulement de l'algorithme récursif, il suffit d'inverser l'arbre :



Algorithme de tri par fusion récursif

tri_fusionrec(*liste,p,r*)
  Si $p < r$
    $q = \lfloor (p+r)/2 \rfloor$
    tri_fusionrec(*liste,p,q*)
    tri_fusionrec(*liste,q+1,r*)
    fusionrec(*liste,p,q,r*)

On a besoin d'une procédure **fusionrec(liste,p,q,r)** dont les paramètres sont : une `liste` d'entiers, l'indice `p` du premier élément d'un segment ordonné A de `liste`, l'indice `q` du dernier élément du segment A (`A=liste[p,..,q]`) et l'indice `r` du dernier élément d'un segment ordonné B de `liste` contigu à A (`B=liste[q+1,..,r]`). La fonction `fusionrec` remplace les segments contigus A et B dans liste par leur fusion en un segment ordonné. Elle effectue un traitement similaire à celui de la fonction `fusion` utilisée dans le tri par fusion version itérative, en incluant la recopie de la fusion des segments fusionnés A et B dans la liste initiale.

Exercice 13

1. Programmer en Python les fonctions `fusionrec` et `tri_fusionrec`.
2. Rajouter ces fonctions dans le module `trieleve.py`.
   Avec la fonction `test`, comparer les performances des fonctions `tri_selection` et `tri_fusionrec` sur des listes d'entiers aléatoires de petite (30), moyenne (100 ou 500) ou grande taille (1000, 5000, 10000) puis sur des listes d'entiers triées dans l'ordre croissant ou décroissant.
   Comme `tri_fusionrec` prend trois paramètres alors que les autres fonctions de tri en prennent trois, on testera avec `test` une fonction enveloppe qui prend `liste` pour seul paramètre :

```
@timetest
def tri_fusionrec_enveloppe(liste):
    """fonction enveloppe pour l'algorithme de tri fusion
    version récursive"""
    p = 0
    r = len(liste)-1
    def tri_fusionrec(liste,p,r):
        .....
    return tri_fusionrec(liste,p,r)
```

3.5.2 L'approche Diviser-pour-Régner

L'algorithme de tri par fusion présente une approche **Diviser-pour-Régner** commune à de nombreux algorithmes récursifs (comme le tri rapide). Une telle approche se décline en trois étapes à chaque étape de la récursivité :

- **Diviser** : le problème en plusieurs sous-problèmes qui constituent des instances de taille inférieure du même problème.
- **Régner** : sur les sous-problèmes en les résolvant de manière récursive (prévoir une condition d'arrêt lorsque la taille de l'instance est minimale)
- **Combiner** : les solutions des sous-problèmes pour reconstruire la solution du problème originel.

Références utiles :

- `http://fr.wikipedia.org/wiki/Diviser_pour_régner_(informatique)`
- *Algorithmique* de Cormen-Leiserson-Rivest-Stein Chapitre 2 pages 26 à 35.

Exercice 14 *Le tri rapide, un autre algorithme Diviser-pour-Régner*

Le *tri rapide* (ou *quicksort*) est un des algorithmes de tri les plus performants : sa complexité dans le cas moyen (liste d'entiers aléatoires) est proportionnelle de à $n\log_2(n)$ comme pour le tri fusion mais avec un facteur constant plus petit. En revanche dans le pire des cas, sa complexité est quadratique comme pour les tris par sélection ou par insertion.
Algorithme du tri rapide :

- **Diviser :** on choisit un entier au hasard dans la liste (par exemple le premier) et on l'appelle pivot et on place les éléments de la liste plus petits que le pivot à sa gauche et les éléments plus grands à sa droite (on peut inclure le pivot comme premier élément du segment à droite) ;
- **Régner :** on applique récursivement l'algorithme aux deux segments de liste de part et d'autre du pivot, on s'arrête lorsqu'on atteint des segments de taille 1 qui sont donc déjà triés ;
- **Combiner :** les deux segments étant triés par appels récursifs de `tri_rapide`, il n'y a rien à faire.

1. Appliquer à la main l'algorithme de tri rapide aux listes suivantes :

a. [17,4,18,16,6,5,2,13]
b. [11,6,6,12,17,10,16,20]
c. [5,1,14,18,14,1,7,4]
d. [7,10,7,16,6,5,17,15]

2. Ecrire une fonction récursive `tri_rapide(liste)` qui trie une liste d'entiers avec l'algorithme de tri rapide.

3. Rajouter la fonction `tri_rapide` dans le module `trieleve.py` et comparer ses performances avec celles des autres algorithmes de tri grâce à la fonction `test`.
Que se passe-t-il pour une liste déjà triée de taille 1 000 ?
En Python, la profondeur du nombre d'appels récursifs successifs est limité par sécurité pour éviter une saturation de la pile. Pour accéder cette valeur ou la modifier il faut utiliser les fonctions `sys.getrecursionlimit()` et `sys.setrecursionlimit()` du module `sys` (la valeur maximale dépendra de la machine).

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(10000)
>>> sys.getrecursionlimit()
10000
```

Expliquer ce qu'il se passe par défaut pour le tri rapide d'une liste de 1 000 entiers déjà triés.

4. Faire une recherche internet sur la complexité du tri rapide.

Table des matières