TP5

1 Le tri par insertion dichotomique

1.1 Tri par insertion : le principe

On considère une liste L d'éléments comparables que l'on trie selon le protocole suivant. On classe au fur et à mesure de l'avancée dans le tableau : si le tableau n'a qu'un seul élément il est déjà trié. Sinon on prend le deuxième et on le compare au premier pour les mettre dans le bon ordre. Si on suppose les i premiers éléments déjà classés on prend le $(i+1)$-ème que l'on compare, par exemple en descendant à partir de la position i, jusqu'à trouver, ou pas, un plus petit que lui et l'insérer alors à la bonne place.

1.2 Une implémentation Python du tri par insertion

Cela peut s'écrire, de manière impérative, en Python pour une liste L donnée :

```
def tri_insertion(L):
    for i in range(1,len(L)):
        u = L[i]
        j = i
        while j > 0 and L[j-1] > u:
            L[j] = L[j-1]
            j = j-1
        L[j] = u
```

Ici, le tri se fait en place, ce qui est positif du point de vue de l'utilisation de la mémoire. On peut apporter des améliorations pour limiter le nombre de comparaisons ou d'échanges, mais cela ne change pas le principe d'insertion : il y a deux boucles imbriquées, ici une boucle globale en for et à l'intérieur une boucle en while.

1. Entrer le code dans un script et le tester.
2. Écrire une fonction `liste_aleatoire` d'arguments le couple d'entiers naturels (N, n), créant de manière aléatoire une liste de longueur n d'entiers compris entre 0 et N.
3. À l'aide de la fonction `time` du module `time`, donner la durée d'exécution de la fonction `tri_insertion` pour quelques listes aléatoires de grande taille. On pourra commencer par $N = 10^7$ et n dans $\{100, 1000, 10\,000\}$.

On prouvera plus tard la correction du programme et on étudiera sa complexité.

1.3 L'insertion dichotomique

En profitant du fait qu'au moment de chaque insertion, le début de liste est déjà trié, on peut procéder à une *insertion dichotomique* selon le principe qui suit. On suppose que l'on dispose d'une liste L de longueur n triée dans l'ordre croissant et d'un élément a comparable que l'on veut ajouter à la liste en l'insérant de manière telle qu'en sortie la nouvelle liste soit encore triée dans l'ordre croissant. On considère l'élément m du milieu de la liste (il faudra choisir lequel on prend lorsque n est pair). Si $a = m$ alors on peut insérer a à gauche ou à droite de m (il faudra choisir), si $a < m$ alors on peut recommencer en insérant a dans la sous-liste extraite de L en prenant les premiers éléments

de L jusqu'à m (exclu ou pas) et si $a > m$, alors on l'insère dans la sous-liste de L extraite à droite de m.

Donner une implémentation en Python du tri par insertion dichotomique. (on pourra se lancer seul ou s'appuyer sur ce qui suit)

Avec des outils mathématiques un peu sophistiqués on montre que la complexité dans le pire des cas du tri avec insertion dichotomique est en $n \ln(n)$.

On peut commencer par écrire une fonction qui cherche, par dichotomie, le bon indice d'insertion d'un élément e dans une liste déjà triée L (on en donne ici une version itérative). Là encore, tester, comprendre avant de l'utiliser.

```
def cpd(L,e):
"""détermine l'indice d'insertion d'un nouvel élément dans une liste déjà triée"""
    a, b = 0, len(L) - 1
    while a <= b:
        m = (a+b) // 2
        f = L[m]
        if e == f:
            return m
        elif e > f:
            a = m + 1
        else:
            ...
    return a
```

et il ne reste plus qu'à écrire une fonction qui implémente le tri par insertion dichotomique.

```
def tri_insert_dicho(liste):
    Ltriee = [liste[0]]
    for i in range(1, len(liste)):
        ...
    return Ltriee
```

La fonction donnée ici ne fait plus un tri en place mais il est possible de faire le tri dichotomique en place.

2 Le problème du sac à dos

Il s'agit d'un paradigme d'optimisation linéaire où l'approche gloutonne donne de bons résultats. On dispose de n objets qui ont chacun un poids p_i et une valeur v_i pour i dans $[\![1, n]\!]$ et d'un sac qui peut supporter un poids maximal P. Quels objets faut-il prendre pour optimiser le remplissage du sac à dos du point de vue de la valeur transportée ?

Pour un n petit, la force brute s'impose mais pour un n de l'ordre de quelques dizaines, cela n'est déjà plus possible. On adopte donc une stratégie gloutonne, mais il y a ici plusieurs possibilités. On pourrait prendre en premier les objets de poids minimaux pour augmenter le nombre d'objets et ainsi espérer augmenter la valeur transportée. On pourrait prendre en premier les objets de valeurs maximales avec le même objectif. On pourrait enfin classer les objets par le rapport $r_i = \frac{v_i}{p_i}$ et prendre en premier les objets qui ont les r_i les plus grands. Nous allons explorer ces trois stratégies gloutonnes.

Pour l'écriture des fonctions, on suppose que l'on dispose d'une liste $L = \big[[1, p1, v1], ..., [n, pn, vn]\big]$ de longueur n et d'un poids maximal P.

Pour une exploration numérique ou faire un test, on prendra

$$L = \big[[1, 14, 126], [2, 2, 32], [3, 5, 20], [4, 1, 5], [5, 6, 18], [6, 8, 80]\big] \text{ et } P = 15.$$

On aura besoin de classer nos objets par poids croissants, puis par valeurs décroissantes et enfin par rapports r_i décroissants. Nous avons déjà vu des algorithmes de tri et nous en verrons d'autres encore plus tard. Pour l'instant, comme ce n'est pas le sujet ici, on utilise la fonction `sorted` de Python qui permet de trier une liste en précisant la clé du tri selon la syntaxe suivante.

`sorted(L, key = lambda x : x[i])` qui trie la liste des x suivant l'ordre croissant des $x[i]$ et

`sorted(L, key = lambda x : x[i], reverse = True)` qui fait la même chose dans l'ordre décroissant.

Ainsi pour la liste L donnée ci-dessus, l'instruction `T = sorted(L, key = lambda x : x[1])` crée une variable `T` qui est la liste L classée par poids croissants et l'instruction `U = sorted(L, key = lambda x : x[2], reverse = True)` crée une variable `U` qui est la liste L classée par valeurs décroissantes.

1. On donne ici la version gloutonne pour les poids `sac_a_dos_poids` qui à partir d'une liste d'objets, avec poids et valeurs, $L = \big[[1, p1, v1], ..., [n, pn, v_n]\big]$ et d'un poids maximal P retourne un remplissage en précisant les numéros des objets retenus, la valeur et le poids ainsi obtenus.

```
def sac_a_dos_poids(L, P):
    """La liste L est supposée classée selon les poids croissants"""
    n, liste, poids, valeur, i = len(L), [], 0, 0, 0
    while i < n and poids <= P:
        if poids + L[i][1] <= P:
            liste.append(L[i][0])
            poids += L[i][1]
            valeur += L[i][2]
        i += 1
    return liste, valeur, poids
print(sac_a_dos_poids(T, 15))
```

   Quel résultat obtient-on pour l'exemple numérique proposé ci-dessus ?

2. Implémenter une fonction Python `sac_a_dos_valeurs` qui procède de même mais en adoptant la stratégie gloutonne des valeurs décroissantes.

   Qu'obtient-on pour l'exemple numérique proposé ?

3. Implémenter une fonction Python `sac_a_dos_rapports` qui procède de même mais en adoptant la stratégie gloutonne des rapports décroissants.

4. Comme ici, la liste n'est que de longueur 6, on peut mettre en œuvre la force brute. Déterminer ainsi, dans le cas particulier proposé ci-dessus, la valeur maximale que l'on peut transporter et comparer aux trois valeurs précédemment obtenue.

3 Glouton ou pas glouton ?

Peut-on proposer un algorithme glouton pertinent pour l'exemple introductif vu en cours, à savoir : pour une liste `L` de nombres de longueur `n`, la maximisation, pour toutes les permutations `T=[T[0],...,T[n-1]]` de `L`, de la fonction (longueur du chemin, du parcours de `L` dans l'ordre représenté par `T`)

$$f(T) = \sum_{i=1}^{n-1} \big|T[i] - T[i-1]\big| \, ?$$