NOM : **Prénom :** **Classe :**

16 oct. 2018 # DS d'informatique I2 — Promo 122

A. Piles

Exercice 1 Parenthésage (5/30 points)

Nous allons utiliser la structure de pile pour savoir si une chaîne de caractères est bien parenthésée. Une chaîne est bien parenthésée si l'une des trois conditions suivantes est vérifiée :

- elle ne contient aucune parenthèse ;
- elle est de la forme "(*mot*)" ou *mot* est une chaîne de caractères bien parenthésée ;
- elle est la concaténation de 2 chaînes de caractères bien parenthésées.

Coder en Python une fonction (sans utiliser les piles) **parenthesage(chaine)** permettant de savoir si la chaîne de caractères **chaine** est bien parenthésée. Dans le cas où la chaine de caractères est bien parenthésée, la fonction retourne *True* ; dans le cas contraire, elle retourne *False*. Pour cela, vous pouvez parcourir la chaîne de caractères de gauche à droite, incrémenter un compteur à chaque fois que l'on rencontre une parenthèse ouvrante et décrémenter ce même compteur à chaque parenthèse fermante et vérifier que ...

Recoder la fonction précédente en remplaçant le compteur par une pile.

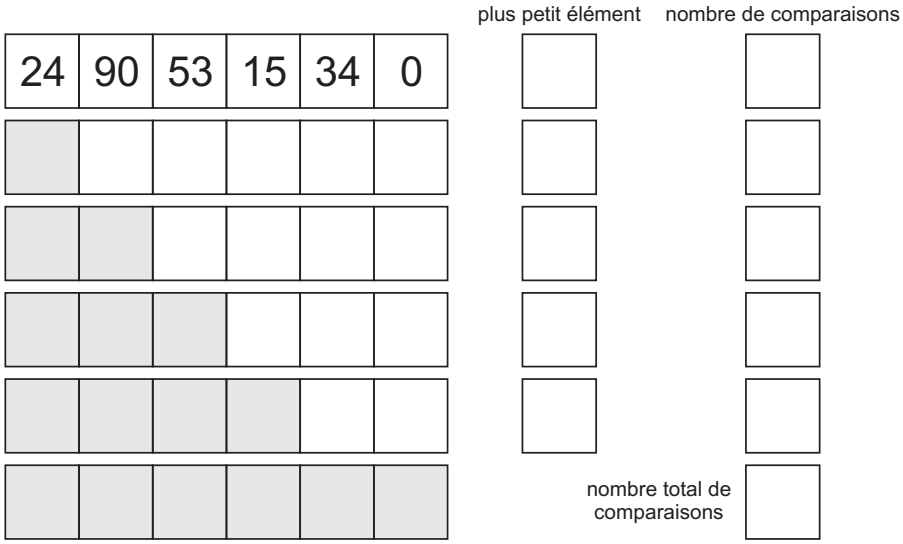
B. Algorithmes de tri

Exercice 2 Tri par sélection (7,5/30 points)

Soit L une liste de n éléments, le principe du tri par sélection est le suivant :

- rechercher le plus petit élément de la liste L et l'échanger avec le premier élément de la liste L ;
- rechercher le deuxième plus petit élément de la liste L et l'échanger avec le deuxième élément de la liste L ;
- continuer de cette façon jusqu'à ce que la liste L soit entièrement triée.

Effectuer l'algorithme « à la main » sur la liste $[24, 90, 53, 15, 34, 0]$ en complétant la figure ci-dessous. Compter le nombre de comparaisons réalisées.



Pour une liste de n éléments, combien y a-t-il de comparaisons ? En déduire la complexité du tri par sélection.

Coder en Python la fonction **indice_min(L)** qui retourne l'**indice** du plus petit élément de la liste L.

Rappel : L'instruction $L[i:j]$ retourne une liste contenant les éléments d'indice i à $j-1$ de la liste L.

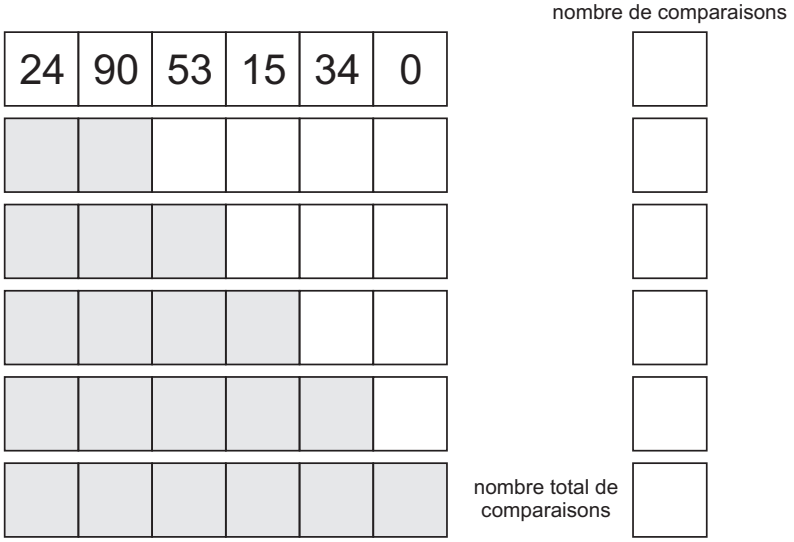
Coder en Python la fonction **tri_selection(L)** qui utilise l'algorithme du tri par sélection pour trier la liste L en utilisant **obligatoirement** la fonction **indice_min(L)**.

Exercice 3 Tri par insertion (7/30 points)

Le tri par insertion est le tri le plus connu. C'est celui que les gens utilisent intuitivement quand ils doivent trier une liste d'objets, par exemple quand on joue aux cartes. Soit L une liste de n éléments, le principe du tri par sélection est le suivant :

- on commence par trier les deux premiers éléments de la liste L l'un par rapport à l'autre ;
- puis on trie le troisième élément de la liste L par rapport aux deux premiers en l'insérant parmi les deux premiers éléments ;
- et ainsi de suite jusqu'à ce que la liste L soit entièrement triée.

Effectuer l'algorithme « à la main » sur la liste $[24, 90, 53, 15, 34, 0]$ en complétant la figure ci-dessous. Compter le nombre de comparaisons réalisées.



Pour quelle type de listes a-t-on le minimum de comparaisons ? Pour une liste de n éléments, combien y a-t-il de comparaisons au minimum ? En déduire la complexité minimale du tri par insertion.

Pour quelle type de listes a-t-on le maximum de comparaisons ? Pour une liste de n termes, combien y a-t-il de comparaisons au maximum ? En déduire la complexité maximale du tri par insertion.

Rappels :

- L'instruction $L.insert(i, e)$ permet d'insérer l'élément e dans la liste L à la position d'indice i.
- L'instruction *del* $L[i]$ permet de supprimer de la liste L l'élément d'indice i.

Coder en Python la fonction **tri_insertion(L)** qui utilise l'algorithme du tri par insertion pour trier la liste L.

C. Récursivité

Exercice 4 Suite de Fibonacci (2/30 points)
On considère la suite de Fibonacci (U_n) suivante :

$$U_0 = 1 \quad , \quad U_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, U_{n+1} = U_n + U_{n-1}$$

Coder en Python la fonction **récursive fibonacci(n)** qui retourne le terme U_n de la suite de Fibonacci.

D. Complexité

Exercice 5 Recherche d'un élément d'une liste (8,5/30 points)
Nous allons comparer plusieurs algorithmes permettant de déterminer l'**indice** d'un élément e d'une liste **triée** L de n **entiers**.

Dans un premier temps, on recherche de manière naïve l'indice de l'élément e de la liste L :

```
1 def recherche_indice1(L,e):
2     i=0
3     while(i<len(L)):
4         if(L[i]==e):
5             return i
6         else:
7             i=i+1
```

Expliquer **clairement** (pas de pseudo-code) comment la fonction **recherche_indice1(L,e)** détermine l'indice de l'élément e de la liste L.

Déterminer le nombre de comparaisons d'éléments réalisées pour une liste de n éléments dans le pire des cas (lorsque e est le dernier élément de la liste L). En déduire la complexité maximale de cette fonction.

On souhaite améliorer la complexité à l'aide d'une fonction récursive :

```
def recherche_indice2(L,e):
    indice=len(L)-1
    if(L[indice]==e):
        return indice
    else:
        L1=L[0:len(L)-1]
        indice=recherche_indice2(L1,e)
        return indice
```

Remarque : L'instruction $L[i:j]$ retourne une liste contenant les éléments d'indice i à $j-1$ de la liste L.

Expliquer **clairement** (pas de pseudo-code) comment la fonction **recherche_indice2(L,e)** détermine l'indice de l'élément e de la liste L.

Combien de fois la fonction **recherche_indice2(L,e)** est-elle appelée pour une liste de n éléments dans le pire des cas (lorsque e est le premier élément de la liste L). Déterminer le nombre de comparaisons d'éléments réalisées **à chaque appel** de la fonction **recherche_indice2(L,e)**. En déduire la complexité maximale de cette fonction.

Comme la complexité de la fonction **recherche_indice2(L,e)** est identique à celle de la fonction **recherche_indice1(L,e)**, on utilise la dichotomie pour rechercher l'indice de l'élément e :

```
def recherche_indice3(L,e):
    indice=len(L)//2
    if(L[indice]==e):
        return indice
    else:
        if(L[indice]>e):
            L1=L[0:len(L)//2]
            indice=recherche_indice3(L1,e)
        else:
            L2=L[len(L)//2:len(L)]
            indice=indice+recherche_indice3(L2,e)
        return indice
```

Expliquer **clairement** (pas de pseudo-code) comment la fonction **recherche_indice3(L,e)** détermine l'indice de l'élément e de la liste L.

On suppose que le nombre d'éléments contenus dans la liste L est paire, soit : $n = 2^k$.

Pour n=1, 2, 4 puis 8 (soit k=0, 1, 2 puis 3) compter le nombre d'appels total à la fonction **recherche_indice3(L,e)** dans le pire des cas (lorsque e est le premier ou le dernier élément de la liste L).

Soit i le nombre maximum de fois que la liste L à été divisée par 2, exprimer i en fonction de k. En déduire le nombre d'appels à la fonction **recherche_indice3(L,e)** en fonction de k puis en fonction de n (rappel : $n = 2^k$).

Déterminer le nombre maximum de comparaisons d'éléments réalisées **à chaque appel** de la fonction **recherche_indice3(L,e)**. En déduire la complexité de cette fonction.