

Cahier de vacances - Les algorithmes à maîtriser en fin de sup

On reprend un à un les points cités par le programme...

1/. Rechercher un élément dans une liste

Ecrire une fonction **cherche**(**L**,**elt**) qui renvoie un booléen indiquant si l'élément **elt** est présent ou non dans **L**, sans utiliser le test **in**.

Donner la complexité en temps dans le pire des cas.

Variante : qui renvoie la position de l'élément (ou *la liste de toutes les positions de l'élément*)

2/. Faire la somme des éléments d'une liste (sans sum)

Ecrire une fonction **somme**(**L** : [**int**]) ->**int** qui renvoie la somme des éléments de la liste **L**.

Donner la complexité en temps de cette fonction.

3/. Compter le nombre d'éléments d'une liste qui vérifient un critère donné

Ecrire une fonction **compter (L) ->int** qui renvoie le nombre d'éléments positifs de la liste L ; on supposera que L ne contient que des flottants.

Donner la complexité en temps de cette fonction.

4/. Rechercher le maximum minimum dans une liste

Ecrire une fonction **maxi (L)** qui renvoie le maximum des éléments de L, ainsi qu'une de ses positions dans la liste.

Donner la complexité en temps dans le pire des cas.

Variante : qui renvoie **toutes** les positions du maximum

5/. Comptage des éléments d'un tableau à l'aide d'un dictionnaire

Ecrire une fonction **compte(L)** -> **dict** qui renvoie un dictionnaire indiquant le nombre d'occurrences de chaque élément de L.

Donner la complexité en temps dans le pire des cas.

6/. Recherche d'un facteur dans un texte

Ecrire une fonction **cherche_mot(T, mot)** qui renvoie l'indice de la première occurrence du mot dans le texte si le mot y est, et `False` ou `None` sinon.

Variante : qui renvoie les indices de *toutes* les occurrences

Donner la complexité en temps dans le pire des cas pour la première version.

Version sans slicings :

Version avec slicings :

7/. Algorithmes de tri quadratiques (en $O(\text{len}(L)^2)$)

a/. Ecrire une fonction **tri_bulles(L) -> None** qui trie en place la liste L ; on supposera que L est une liste de valeurs numériques.

Donner la complexité de ce tri dans le meilleur et dans le pire des cas.

a/. Ecrire une fonction **tri_insertion(L) -> None** qui trie en place la liste L avec un tri par insertion (le plus classique aux écrits !!!) ; on supposera que L est une liste de valeurs numériques.

Donner la complexité de ce tri dans le meilleur et dans le pire des cas, ainsi que la complexité moyenne.

8/. Recherche dichotomique dans un tableau trié

Ecrire une fonction `cherche_dicho(L, a)` qui recherche si l'élément `a` est présent dans la liste `L`. On supposera la liste `L` triée dans l'ordre croissant. Si l'élément `a` est présent, on renverra sa position dans la liste, sinon on renverra `False`.

Donner la complexité en temps de cet algorithme. Justifier brièvement ; on pourra supposer que la liste `L` contient 2^p éléments (plus simple à montrer).

Quelques corrections

1/. Rechercher un élément dans une liste

```
def cherche(L, elt) -> bool :  
  
    """L : liste de valeurs (de tous types), elt : élément à chercher  
    -> booléen indiquant si elt est présent dans L """  
  
    for k in L :  
        if k == elt :           # élément trouvé !  
            return True  
    return False               # élément absent de la liste
```

Complexité : Parcours de liste en $O(\text{len}(L))$ et le corps de boucle (test) est en temps constant $O(1)$ (indépendant de $\text{len}(L)$), donc complexité en $O(\text{len}(L))$.

```
def cherche_positions(L,elt) -> [int] :  
  
    """L : liste d'éléments (de tous types), elt : élément à chercher  
    -> liste des positions de cet élément dans L, et liste vide si  
    élément absent de L """  
  
    pos = []  
    for k in range(len(L)) :  
        if L[k] == elt :       # élément trouvé !  
            pos.append(k)  
    return pos
```

Complexité :

Affectation, test et ajout avec append : complexité en $O(1)$.

Le coût est donné par le nombre d'itérations de la boucle donc $O(\text{len}(L))$.

2/. Faire la somme des éléments d'une liste (sans sum)

```
def somme(L :[int]) -> int :  
    """L : liste de valeurs numériques  
    -> somme des éléments de L """  
  
    S = 0  
    for k in L :  
        S = S + k  
    return S
```

Complexité : Soit $n = \text{len}(L)$.

Une bouclé `for` réalisée n fois, le reste étant en temps constant (indépendant de $\text{len}(L)$).

Donc **complexité linéaire** $O(n)$.

3/. Compter le nombre d'éléments d'une liste qui vérifient un critère donné

```
def compteur(L)->int :  
    """L : liste de valeurs (ici nombres)  
    -> nombre de valeurs positives dans L (par exemple) """  
  
    cpt = 0  
    for k in L :  
        if k >= 0 :  
            cpt = cpt + 1  
    return cpt
```

Complexité : soit $n = \text{len}(L)$

Une boucle `for` réalisée n fois, le reste étant en temps constant (affectations, comparaison, somme). Complexité linéaire en la longueur de la liste, donc $O(n)$.

4/. Rechercher le maximum minimum dans une liste

```
def maxi(L) :  
    """L : liste de valeurs  
    -> maximum des éléments et la position d'une occurrence du max """  
  
    M = L[0]          # premier terme de la liste  
                     # éventuellement float('inf') dans certains exos  
    pos = 0  
    for k in range(len(L)) :  
        if L[k] > M :  
            M = L[k]      # nouveau maximum temporaire trouvé  
            pos = k       # indice mémorisé  
    return M, pos
```

Complexité : comparaisons, affectations s'exécutent en temps constant donc $O(1)$ (c'est-à-dire indépendant de $\text{len}(L)$)

Le coût vient du nombre de fois où la boucle est exécutée, soit $\text{len}(L)$ fois.

Complexité en $O(\text{len}(L))$

```
def maxi2(L) -> [int] :  
    """L : liste de valeurs comparables  
    -> liste des positions du maximum dans L, en un seul parcours """  
  
    pos = []  
    M = L[0]  
    for k in range(len(L)) :  
        if L[k] > M :  
            M = L[k]          # nouveau maximum trouvé et mémorisé  
            pos = [k]         # on écrase les anciennes positions, on recommence  
        elif L[k] == M :  
            pos.append(k)     # autre apparition du maximum temporaire  
                               # position mémorisée  
    return M, pos
```

Complexité : La boucle for est exécutée $\text{len}(L)$ fois et le corps de boucle est en temps constant (affectations, comparaisons, ajout dans une liste, donc opérations indépendantes de $\text{len}(L)$).

Complexité en $O(\text{len}(L))$.

5/. Comptage des éléments d'un tableau à l'aide d'un dictionnaire

```
def comptage(L : list) -> dict :  
    """L : liste de valeurs (pas forcément comparables)  
    -> dictionnaire ayant pour clés les éléments distincts de L et pour valeurs  
    le nb d'occurrences de chacune de ces valeurs  
  
    Rappel : aucun ordre dans un dictionnaire !!! """  
    d = {k : 0 for k in L}    # aucun risque de doublon sur les clés (structure d'ensemble)  
    # remplissage du dictionnaire :  
    for k in L :  
        d[k] = d[k] + 1      # valeur k rencontrée une fois de plus  
    return d
```

Complexité :

- Remplissage **initial** du dictionnaire par compréhension : $O(\text{len}(L))$
- Compléter le dictionnaire en $O(\text{len}(L))$ car un parcours de L et une affectation en $O(1)$

Donc complexité = $O(n) + O(n) = 2 O(n) = O(n)$ avec $n = \text{len}(L)$.

6/. Recherche d'un facteur dans un texte

Version sans slicings :

```
def cherche_mot(T ,mot) -> [int] :  
  
    """ T : texte ou liste de valeurs  
    mot : texte ou liste de valeurs  
    -> liste des positions (initiales) de toutes les occurrences de mot dans T """  
  
    assert len(T) >= len(mot)    # sinon aucun intérêt de faire la recherche  
  
    pos = []  
    for k in range(len(T) - len(mot) + 1) : # on teste les positions de départ possibles  
  
        # recherche du mot à partir de la position k  
        cpt = 0 # nb de caractères communs  
        for i in range(len(mot)) : # sur tout la taille du petit mot  
            if T[k+i] == mot[i] : # caractères égaux  
                cpt = cpt + 1 # un caractère commun en plus  
  
        if cpt == len(mot) : # mot trouvé !  
            pos.append(k) # ajout de la position de la 1ère lettre du mot  
                           # dans le texte  
  
    return pos
```

Complexité dans le pire des cas :

On note $n = \text{len}(T)$ et $m = \text{len}(\text{mot})$.

Boucle externe réalisée $n - m$ fois.

Pour chacune de ces fois, une boucle réalisée m fois et des opérations à coût constant
Une comparaison de `cpt` et un ajout en temps constant

Complexité globale : $O(n - m) \times (O(m) + O(1)) = O(n - m) \times O(m)$

Donc complexité en $O((n - m) \times m)$.

Version avec slicings :

```
def cherche_mot2(T, mot) -> [int] :  
  
    """T : texte ou liste de valeurs  
    mot : texte ou liste de valeurs  
    -> liste des positions de départ de toutes les occurrences de mot dans T"""  
  
    assert len(T) >= len(mot)  
  
    pos = []  
    for k in range(len(T) - len(mot) + 1) :  
        if T[k : k+len(mot)] == mot :  
            pos.append(k)  
    return pos
```

7/. Algorithmes de tri quadratiques (en $O(\text{len}(L)^2)$)

a/.

```
def tri_bulles(L)-> None :  
    """L : liste de valeurs comparables  
    -> None ;  
    tri de la liste dans l'ordre croissant ; la liste L sera modifiée  
    en place """  
  
    for i in range(len(L)-1) :  
        for k in range(len(L)-i-1) :  
            if L[k+1] < L[k] :  
                L[k+1],L[k] = L[k], L[k+1]      # pas dans le bon ordre  
                                                # on les permute  
  
# tests  
from randint import randint  
L_test = [randint(-10,10) for k in range(10)]  
print(L_test)                                # liste d'entiers pseudi-aléatoires  
tri_bulles(L_test)                            # on trie la liste  
print(L_test)                                # on vérifie ...
```

Complexité : notons $n = \text{len}(L)$

- Dans le pire des cas, la complexité est quadratique = $O(n^2)$:

En effet, dans le pire des cas, la liste est initialement triée dans l'ordre décroissant ; chaque paire d'éléments consécutifs est dans le mauvais ordre, donc chaque comparaison entraîne une permutation.

- La boucle externe s'exécute $n - 1$ fois.
- Pour chaque itération i , la boucle interne s'exécute $n - i - 1$ fois
- Le nombre total de comparaisons est donc :

$$\sum_{i=0}^{n-2} (n - i - 1) = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} \in O(n^2)$$

- Dans le meilleur des cas, la complexité est quadratique , $O(n^2)$.

Le meilleur des cas est lorsque la liste est déjà triée ; la version ci-dessus n'est pas optimisée ; on pourrait écrire une version qui détecte quand le tri est terminé (un parcours sans la moindre modification), et on gagnerait alors en calculs. Ici, si la liste est déjà triée, toutes les comparaisons seront effectuées, sans faire les permutations. Donc on effectue autant de comparaisons que dans le pire des cas. Donc complexité en $O(n^2)$.

b/. Tri par insertion

```
def tri_insertion(L) -> None :  
    """L : liste de valeurs comparables  
    -> None ; tri en place de la liste L par ordre croissant """  
    for k in range(1, len(L)) :  
        x = L[k]                # élément à placer  
        j = k  
  
        # on cherche la place de x parmi le début de la liste, triée.  
        while j > 0 and x < L[j-1] : # tant que pas au début de la liste, et  
            # que la future place de x n'est pas trouvée  
            L[j] = L[j-1]          # on décale le terme d'avant  
            j = j-1                # on décale vers la gauche le compteur  
            # qui cherche la future place de x  
        L[j] = x                  # on place x dans la position libérée
```

Complexité : notons $n = \text{len}(L)$

- Dans le pire des cas, la complexité est quadratique = $O(n^2)$:

Pire des cas quand la liste est initialement triée dans l'ordre décroissant ; donc chaque nouvel élément doit être comparé à tous ceux d'avant et inséré au début.

- A chaque itération il y a k comparaisons et k décalages.
- Le nombre total d'opérations est donc :

$$\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} \in O(n^2)$$

- Dans le meilleur des cas, la complexité est linéaire , $O(n)$.

Le meilleur cas se produit lorsque la liste est déjà triée. Dans ce cas :

- Condition $x < L[j-1]$ toujours fausse, donc le `while` ne s'exécute jamais.
- Il y a donc exactement $n - 1$ comparaisons, et aucun décalage.

Donc une complexité en $O(n)$.

- Complexité moyenne : $O(n^2)$

8/. Recherche dichotomique dans un tableau trié

```
def cherche_dicho(L,a) :  
    """L : liste triée dans l'ordre croissant, a: élément à chercher  
    -> position de a dans L si a est dans L, et False sinon """  
  
    d,f = 0, len(L)-1          # dichotomie sur les positions  
    while d <= f :             # tant que les compteurs ne se sont pas rejoints  
        m = (d+f)//2           # indice médian  
        if L[m] == a :         # a trouvé à la position m  
            return m  
        elif L[m] < a :        # on garde la partie droite  
            d = m+1  
        else :                 # on garde la partie gauche  
            f = m-1  
    return False               # élément a pas dans L
```

Complexité :

- **Dans le pire des cas : $O(\log(n))$** avec $n = \text{len}(L) = 2^p$

En effet, si $n = 2^p$; à chaque itération, la recherche conserve la moitié de la liste précédente.

Taille initiale : 2^p

Taille après une itération : 2^{p-1}

Taille après 2 itérations : 2^{p-2}

...

Taille après k itérations : 2^{p-k}

Arrêt de la boucle quand il n'y a plus qu'un seul élément, donc quand $2^{p-k} = 1$ ce qui donne $p = k$, donc après maximum p itérations.

Or, si $n = 2^p$, alors $p = \log_2(n)$.

Donc dans le pire des cas, la recherche effectue $\log_2(n)$ comparaisons.

Donc complexité en $O(\log(n))$.

- **Complexité moyenne : $O(\log(n))$**
- **Complexité dans le meilleur des cas : $O(1)$** , donc indépendante de la taille de la liste.

C'est le cas où l'élément cherché est à la position médiane donc trouvé du premier coup.

C'est un cas très favorable qui ne nécessite qu'une seule comparaison.