Chapitre 37

Informatique Python

Optimal Sup-Spé
Groupe Ipesup ▪ Le n°1 en Sup-Spé

Prépas Scientifiques - Concours 2019

Correction des exercices

Sommaire

Algorithmes classiques

Exercice 1 - Maximum, moyenne, écart type ★

1) Pour la fonction `maximum` :

```
1  def maximum(l):
2      m = l[0]
3      for x in l:
4          if x > m:
5              m = x
6      return m
```

Remarque

Si l'on ne fait pas trop attention, ligne 2, on pourrait écrire `m = 0` au lieu de `m = l[0]`. Le comportement de notre fonction serait alors faux pour une liste en entrée ne contenant que des nombres négatifs. Voyez-vous pourquoi ? En prenant `m = l[0]` nous sommes sûr du bon fonctionnement de notre code – la liste est non vide – puisque le maximum de la liste appartient à la liste. Faites bien attention à cette remarque car ce genre de questions arrivera typiquement en tout début de sujet de concours lorsque le candidat n'est potentiellement pas encore parfaitement concentré et où ce genre de petites erreurs sont faciles à commettre.

2) Pour la fonction `moyenne` :

```
1  def moyenne(l):
2      somme = 0
3      for x in l:
4          somme += x
5      return somme/len(l)
```

Rappel de cours

À la ligne 4, l'opérateur `+=` est un raccourci pour `somme = somme + x`.

3) Tout d'abord, quelques rappels sur l'écart type.

Rappel de cours

L'écart type est la racine carrée de la variance. La variance d'une liste est la moyenne des carrés des écarts à la moyenne de cette liste : $V(X) = E((X - E(X))^2)$. En développant cette expression on trouve également : $V(X) = E(X^2) - E(X)^2$. Cela donne deux manière différentes de calculer l'écart type. Dans cette correction nous utilisons la première expression, utilisez la deuxième pour vous entraîner ! Enfin, en Python on peut prendre la racine carré d'un nombre `x` avec la commande `x**0.5` qui calcule $x^{\frac{1}{2}} = \sqrt{x}$.

Pour la fonction `ecart_type`, deux solutions à complexité différente (voir question 4) :

— Solution en $O(n^2)$:

```
def ecart_type1(l):
    somme = 0
    for x in l:
        somme += (x - moyenne(l))**2
    return (somme/len(l))**0.5
```

— Solution en $O(n)$:

```
def ecart_type2(l):
    somme = 0
    m = moyenne(l)
    for x in l:
        somme += (x - m)**2
    return (somme/len(l))**0.5
```

Enfin, plus élégant, avec une compréhension de liste, toujours en $O(n)$:

```
def ecart_type3(l):
    m = moyenne(l)
    return moyenne([ (x-m)**2 for x in l ])**0.5
```

Pour les concours on choisira `ecart_type2` ou `ecart_type3` qui ont la meilleure complexité.

Point méthode

Dans ces trois fonctions nous n'avons pas utilisé la syntaxe `for i in range(len(l))` car nous n'avons pas eu besoin des indices mais simplement des éléments auxquels nous pouvons accéder directement avec `for x in l`. Noter que par convention, les programmeurs préfèrent appeler les indices `i,j,k` et les éléments plutôt `x,e`. Respecter cette convention vous permettra d'éviter tout malentendu avec vos correcteurs.

Remarque

La fonction `maximum` est déjà présente dans Python sous le nom `max`. Les fonctions `moyenne` et `ecart_type` existent dans la bibliothèque `numpy` sous les noms `mean` et `std` ("std" pour "strandard deviation").

4) Il est important de bien savoir rédiger les questions de complexité. Tout d'abord, un rappel :

Rappel de cours

En informatique, la complexité d'une fonction est le nombre d'opérations élémentaires qu'un ordinateur doit effectuer afin de la calculer. La notion d'opération élémentaire n'est pas universelle, mais dans le cadre des concours, l'on conviendra que les opérations élémentaires en Python sont les affectations, les opérations arithmétiques et certaines fonctionnalités internes de Python (l'indexation de liste par exemple : `l[0]`.). De manière générale, utilisez votre bon sens ! Par exemple, la fonction `moyenne` effectue :

— Une affectation (ligne 2).
— Une affectation et une opération arithmétiques (`+`) par élément de la liste `l` (lignes 3 et 4).
— Une opération arithmétique (`/`)

En tout, si n est la taille de la liste, la fonction moyenne effectue $1 + n(1 + 1) + 1 = 2n + 2$ opérations élémentaires, elle a donc pour complexité $2n + 2$. Cependant, étant donné qu'il est fastidieux de compter aussi précisément ce nombre d'opérations, on se contentera de dire que la fonction `moyenne` a une complexité en $O(n)$. La notion de O – grand O – permettant d'omettre les facteurs multiplicatifs (ici 2) et les termes asymptotiquement négligeables (ici +2). Bien que paraissant abstraite, la notion de complexité a une implication très **concrète** : si une fonction `g` a une complexité supérieure à une fonction `f`, elle mettra, asymptotiquement, plus de temps à être calculée.

Le point méthode suivant donne la rédaction type idéale pour les questions de complexité.

Point méthode

Pour la fonction `moyenne` on rédige : "La fonction moyenne parcourt chaque élément de la liste en effectuant à chaque fois uniquement des opérations élémentaires. Ainsi, la complexité de la fonction moyenne est en $O(n)$ avec n la taille de la liste.". Utilisez cette rédaction en début de concours, vous pouvez être plus concis par la suite.

Dans la suite de ce corrigé, nous adoptons une rédaction très concise des questions de complexité, charge à vous de les développer.

— La complexité de la fonction `maximum` est en $O(n)$.
— La complexité de la fonction `moyenne` est en $O(n)$.
— La complexité de la fonction `ecart_type1` est en $O(n^2)$ car chaque tour de boucle fait appel à la fonction `moyenne` qui a une complexité linéaire (c'est-à-dire $O(n)$). Ainsi on obtient *in fine* $O(n * n) = O(n^2)$. Notre fonction est donc inutilement peu efficace ! Dans `ecart_type2`, on calcule la moyenne une fois pour toute et on obtient donc une complexité en $O(n)$. Déjà avec des listes de taille 1000 vous pouvez expériementer une différence de temps sensible entre l'éxecution de la fonction `ecart_type1` et `ecart_type2` qui fait de l'ordre de 10^6 opérations élémentaires au lieu de 1000.

Exercice 2 - Recherche naïve ★

1) Pour la fonction `est_dedans` :

```
1  def est_dedans(l, e):
2      for x in l:
3          if x == e:
4              return True
5      return False
```

Remarque

En allant trop vite, on pourrait écrire la fonction `est_dedans_FAUSSE` ci-dessous. Cette fonction ne répond pas à la question car elle effectuera systématiquement uniquement un tour de boucle, regardant si le premier élément est égal à `e` ou pas. Comprenez-vous l'erreur ? Garder à l'esprit qu'une fonction "meurt" et en particulier interrompt toutes ses boucles au moment où elle rencontre un `return`. Enfin, il n'y a pas lieu d'écrire un `else` dans la fonction `est_dedans` car, s'il elle arrive "vivante" à la ligne 5 cela veut dire que l'élément `e` n'est pas dans la liste `l` et on peut donc renvoyer `False`.

```
def est_dedans_FAUSSE(l, e):
    for x in l:
        if x == e:
            return True
        else:
            return False
```

2) La complexité de la fonction `est_dedans` est en $O(n)$.

Exercice 3 - Recherche dichotomique ★★

1) Pour la fonction `dicho_rec_1` :

```
1  def dicho_rec_1(l,e):
2      if len(l) == 0:
3          return False
4
5      indice_moitie = len(l)//2
6      valeur_moitie = l[indice_moitie]
7
8      if valeur_moitie == e:
9          return True
10
11     if valeur_moitie > e:
12         return dicho_rec_1(l[:indice_moitie], e)
13
14     return dicho_rec_1(l[indice_moitie+1:], e)
```

Rappel de cours

Les slices en Python suivent la convention de Python pour les indices : "début inclus, fin exclue". Cela veut dire que le slice `liste[i:j]`, en supposant `i` $<$ `len(liste)` et $i < j$, contiendra `liste[i]` mais pas `liste[j]`. En fait, le slice `liste[i:j]` contiendra tous les éléments de la liste depuis l'indice `i` jusqu'au dernier indice `k` (inclus) tel que `k` $< j$ et `k` $<$ `len(liste(l))`.

Il est important d'avoir ce point de cours à l'esprit pour `dicho_rec_1` car cela nous permet de s'assurer que cette fonction termine dans tous les cas (pas d'appel récursif infini). En effet, son cas de base (ligne 2) est le cas `len(l) == 0`. Il faut donc s'assurer que, quelque soit la liste `l` de départ, on se ramenera au cas `len(l) == 0` après un certain nombre d'appels récursifs. Cela est garantit par les lignes 12 et 14. En effet, dans chacun des deux cas, le slice que l'on effectue exclut la valeur se situant à `indice_moitie`. Donc à chaque appel récursif, on réduit la taille de la liste en d'argument d'au moins un. Ainsi l'on est sûr un jour d'atteindre le cas `len(l) == 0`.

Remarque

Lorsqu'on écrit une dichotomie, s'assurer qu'elle n'engendre pas d'éxecution infinie est déjà un très bon signe sur la validité de son code. Comme on vient de le montrer, c'est le cas pour `dicho_rec_1`.

Le code de `dicho_rec_1` a le mérite d'être conceptuellement simple : il est aisé de l'écrire sans commettre d'erreur puisqu'on y fait assez clairement ce qu'une dichotomie doit faire. Cependant cette clarté a un coût en complexité qui est caché dans l'utilisation des slices Python (ligne 12 et 14). En effet la complexité d'un slice d'une liste de taille n est en $O(n)$. Ainsi, si notre dichotomie effectue k appels récursifs, la complexité de `dicho_rec_1` est en $O(kn)$ avec n la taille de la liste de départ. Ce n'est pas satisfaisant car comme on va le voir ci-dessous, on peut se passer de ce coût et avoir une complexité en $O(k)$ en se passant des slices.

2) La philosophie de `dicho_rec_2` est de ne jamais modifier la liste `l` mais de faire évoluer deux variables `begin` et `end` qui représentent la sous-liste que l'on considère à chaque appel récursif. On prend la même convention que Python, `begin` inclus et `end` exclue :

```
1  def dicho_rec_2(l,e,begin,end):
2      if end-begin == 0:
3          return False
4
5      indice_moitie = (end+begin)//2
6      valeur_moitie = l[indice_moitie]
7
8      if valeur_moitie == e:
9          return True
10
11     if valeur_moitie > e:
12         return dicho_rec_2(l, e, begin, indice_moitie)
13
14     return dicho_rec_2(l, e, indice_moitie+1, end)
```

Pour utiliser cette fonction, il faudra l'appeler sur les arguments `l,e,0,len(l)`.

On obtient `dicho_rec_2` en adaptant `dicho_rec_1` à la contrainte de ne jamais modifier `l`. Il faut cependant faire particulièrement attention aux lignes 2 et 5. En effet, à ligne 2 on ne peut plus utiliser `len(l)` étant donné que celle-ci ne changera jamais. Il faut se convaincre que `end-begin` correspond à la taille du segment considéré lors de l'appel récursif – on peut par exemple remarquer qu'au premier appel on a `end-begin` = `len(l)` -0 = `len(l)`. Enfin, à la ligne 5 on ne peut plus utiliser `len(l)//2` pour la même raison : `len(l)` ne change jamais. Ici on a la valeur du milieu entre deux nombres : c'est la moyenne.

On peut s'assurer que `dicho_rec_2` termine de la même manière que pour `dicho_rec_1`, on a tout fait pour en adaptant légèrement `dicho_rec_1` pour coder `dicho_rec_2` : `end-begin` diminue strictement à chaque appel (convention début inclus fin exclue).

3) Une fois que l'on a codé `dicho_rec_2`, on peut l'adapter aisément en une implémentation itérative :

```
1  def dicho_iter(l,e):
2      begin = 0
3      end = len(l)
4
5      while end-begin != 0:
6
7          indice_moitie = (end+begin)//2
8          valeur_moitie = l[indice_moitie]
9
10         if valeur_moitie == e:
11             return True
12
13         if valeur_moitie > e:
14             end = indice_moitie
15         else:
16             begin = indice_moitie+1
17
18     return False
```

Une seule subtilité peut être, la ligne 18. C'est ici très similaire au code de la "Recherche naïve" (voir exercice). Si notre programme arrive vivant à la ligne 18, l'on est sûr qu'il n'a jamais trouvé l'élément `e` et que donc l'élément `e` n'est pas dans la liste `l`.

Ce détour par la récursivité nous permet, en augmentant graduellement la difficulté du problème, d'aboutir en un code itératif de la dichotomie dans lequel on peut avoir confiance.

4) Comme on l'a brièvement abordé dans la correction de la question 1, la complexité de la fonction `dicho_rec_1` est en $O(kn)$ avec n la taille de la liste en entrée et k la complexité de la fonction `dicho_rec_2`. D'autre part, la fonction `dicho_iter` consistant uniquement en un jeu de ré-écriture de la fonction `dicho_rec_2`, elles ont la même complexité. Il nous reste donc uniquement à déterminer la complexité de la fonction `dicho_rec_2`.

Supposons, que l'on appelle `dicho_rec_2` sur une liste `l` en entrée de taille $n = 2^m$. On veut prouver par récurrence que la fonction `dicho_rec_2` va se rappeler récursivement elle même (c'est-à-dire atteindre les lignes 12 ou 14) au plus m fois.

— Si $m = 1$: ou bien `l == [e]` auquel cas la fonction ne se rappelle pas elle même et renvoie `True`. Ou bien `l` $\neq$ `[e]` et la fonction se rappelle une fois sur `[]` et renvoie `False`. Dans les deux cas, elle se rappelle au plus 1 fois.
— On suppose vrai pour m. Si on appelle la fonction sur une liste de taille $m+1$, trois cas possible :
  — Ligne 8 : la fonction ne se rappelle jamais elle-même et renvoie `True`.
  — Ligne 12 : la fonction se rappelle sur une sous-liste de taille 2^m. Par hypothèse de récurrence, depuis cette sous-liste, la fonction se rappellera au plus m fois. Au total, depuis la liste d'origine, la fonction se rappelera donc au plus $m+1$ fois.
  — Ligne 14 : la fonction se rappelle sur une sous-liste de taille $2^m - 1$. Depuis cette sous-liste, la fonction ne peut pas se rappeler plus de fois que sur une sous-liste de taille 2^m. Par le même raisonnement qu'au point précédent, la fonction se rappelle au total au plus $m+1$ fois.

Dans tous les cas, la fonction se rappelle au plus $m+1$ fois et on a le résultat.
Comme la fonction n'effectue que des opérations élémentaires (il n'y a plus de slices), sa complexité est donc en $O(m)$ dans les cas où $n = 2^m$ avec n la taille de la liste. Noter que si $n = 2^m$, $m = O(\log(n))$, on a même $m = \log(n)$ si l'on convient, comme souvent en informatique, que $\log = \log_2$.
Ainsi, quand la liste en entrée a une taille n en puissance de 2, on a montré que la complexité de `dicho_rec_2` est en $O(log(n))$. Vous pouvez adapter la preuve ci-dessus au cas général et montrer que la complexité de la fonction `dicho_rec_2` est toujours en $O(log(n))$.

Remarque

Les complexités en $O(log(n))$ paraissent étrange au premier abord. Pour les comprendre il faut se demander : "quelle est la complexité de ma fonction sur une entrée de taille $n = 2^m$". Si la réponse est $O(m)$ alors cela veut dire que notre algorithme requiert $O(log(n))$ étapes pour s'exécuter. Les algorithmes en $O(log(n))$ sont très performants. Ils sont exponentiellement plus rapides que les algorithmes en $O(n)$. Pour se faire une idée, 2^{240} est considéré comme plus grand que le nombre d'atomes dans l'univers. Un algorithme en $O(log(n))$ répondra en de l'ordre de 240 étapes à la question posée sur une entrée de taille 2^{240} alors qu'un algorithme en $O(n)$ répondra en de l'ordre de 2^{240} étapes ce qui revient à se poser une question plus de fois qu'il n'y a d'atomes dans l'univers.

Remarque

La preuve de la complexité de `dicho_rec_2` cache une intuition simple : on ne peut pas découper un nombre entier en deux plus de fois que son logarithme en base 2.

Pour conclure, se souvenir que la complexité optimale d'une dichotomie est $O(log(n))$, ici :
— La fonction `dicho_rec_1` a une complexité en $O(n\log(n))$ avec n la taille de la liste.
— La fonction `dicho_rec_2` a une complexité en $O(\log(n))$ avec n la taille de la liste.
— La fonction `dicho_iter` a une complexité en $O(\log(n))$ avec n la taille de la liste.
Pour les concours, `dicho_rec_2` ou `dicho_iter` sont très satisfaisantes.

Exercice 4 - Médiane d'une liste ★

1) La fonction `mediane` peut s'implémenter de la manière suivante :

```
def mediane(liste):
    taille_liste = len(liste)
    liste_triee = sorted(liste)

    if taille_liste%2 == 1:
        return liste_triee[ taille_liste//2 ]

    return ( liste_triee[ taille_liste//2 ] + liste_triee[ taille_liste//2-1 ] ) / 2
```

Remarque

Ici il est important d'utiliser `sorted` au lieu de `liste.sort()`. En effet, la fonction `liste.sort()` est "en place" : elle modifie directement `liste` or l'on ne veut pas modifier la liste `liste` qui a été donnée par l'utilisateur sinon celui-ci pourrait être impacté dans ses usages futurs de la liste `liste`.

2) Notre fonction `mediane` fait appel à la fonction Python `sorted` dont la complexité est en $O(nlog(n))$ (complexité optimale pour un tri). Toutes nos autres opérations sont élémentaires donc la complexité de la fonction `mediane` est en $O(nlog(n))$.

3) **Bonus :** Non, notre fonction n'a pas une complexité optimale car il existe un algorithme sophistiqué qui calcule la médiane en temps linéaire. Voir : `https://fr.wikipedia.org/wiki/Médiane_des_médianes`

Types et structures de données

Exercice 5 - Manipulation des entiers ★

1) `est_diviseur` :

```
def est_diviseur(a, b):
    return b%a == 0
```

2) `est_premier` :

— Solution en $O(n)$:

```
def est_premier1(n):
    for i in range(2, n):
        if est_diviseur(i, n):
            return False
    return True
```

— Solution en $O(\sqrt{(n)})$:

```
def est_premier2(n):
    # O(sqrt(n))
    for i in range(2, int(n**0.5)):
        # O(1)
        if est_diviseur(i, n):
            return False
    return True
```

3) `premiers_inferieurs` :

```
def premiers_inferieurs(N):
    l = []
    for i in range(2, N+1):
        if est_premier1(i):
            l.append(i)
    return l
```

4) `diviseurs_premiers` :

```
def diviseurs_premiers(N):
    l = []
    for i in range(2, N+1):
        if est_diviseur(i, N) and est_premier1(i):
            l.append(i)
    return l
```

5) `binaire` :

```
def binaire(n):
    l = []
    while n != 0:
        l.append(n%2)
        n //= 2
    return l
```

6) decimal :

```
def decimal(decompo):
    n = 0
    for i in range(len(decompo)):
        n += decompo[i]*2**i
    return n
```

Exercice 6 - Manipulation des ensembles ★★

1) ensemble_vide :

```
def ensemble_vide():
    return []
```

2) est_vide :

```
def est_vide(ens):
    return ens == []
```

3) est_dans :

```
def est_dedans(ens, e):
    for i in range(len(ens)):
        if e == ens[i]:
            return True
    return False
```

4) est_inclus :

```
def est_inclus(ens1, ens2):
    for i in ens1:
        if not(est_dedans(ens2, i)):
            return False
    return True
```

5) sont_egaux :

```
def sont_egaux1(ens1, ens2):
    for i in ens1:
        if not(est_dedans(ens2, i)):
            return False
    for i in ens2:
        if not(est_dedans(ens1, i)):
            return False
    return True
```

Ou bien :

```
def sont_egaux2(ens1, ens2):
    if len(ens1) != len(ens2):
        return False

    for i in ens1:
        if not(est_dedans(ens2, i)):
            return False

    return True
```

6) cardinal :

```
def cardinal(ens):
    return len(ens)
```

7) ajoute :

```
def ajoute(ens, a):
    if not(est_dedans(ens, a)):
        ens.append(a)
```

8) ajoute_liste :

```
def ajoute_liste(ens, l):
    for i in l:
        ajoute(ens, i)
```

9) union :

```
def union1(ens1, ens2):
    ens = []
    ajoute_liste(ens, ens1)
    ajoute_liste(ens, ens2)
    return ens
```

Ou bien :

```
def union2(ens1, ens2):
    ajoute_liste(ens1, ens2)
    return ens1
```

10) inter :

```
def inter(ens1, ens2):
    ens = []
    for i in ens1:
        if est_dedans(ens2):
            ens.append(i)
    return ens
```

11) produit :

```
def produit(ens1, ens2):
    ens = []
    for i in ens1:
        for j in ens2:
            ens.append((i, j))
    return ens
```

12) diff :

```
def diff(ens1, ens2):
    ens = []
    for i in ens1:
        if not(est_dedans(ens2, i)):
            ens.append(i)
    return ens
```

13) est_partition :

```
def est_partition(X, partition):
    uni = []
    for partie in partition:
        uni = union1(uni, partie)
    return sont_egaux1(uni, X)
```

Remarque

Python possède un type `set` qui implémente la plupart des fonctions de cet exercice. Exemple : `my_set = {1, 2, 3}`. Pour en savoir plus, voir la documentation officielle Python pour le type `set`.

Exercice 7 - Manipulation des piles ★★★

Le but de cet exercice est d'implémenter une structure de donnée **FILO** ("First In Last Out"), c'est-à-dire une pile, puis d'implémenter des opérations sur ces piles en utilisant uniquement l'interface de cette structure abstraite. L'interface d'une pile est définie ici comme étant les fonctions `pile_vide`, `empiler`, `est_vide` et `depiler`. La force de cette démarche est que, quelque soit la manière dont on implémente cette interface, si l'on respecte le contrat demandé par l'énoncé, toutes les fonctions que l'on demande de coder par la suite auront le même comportement. Dans les questions 2,3 etc... il s'agit "d'oublier" la manière dont on a représenté nos piles et de n'utiliser uniquement les fonctions de l'interface que l'on a définie.

1) En Python, on peut implémenter les piles grâce aux listes :

— On utilise la liste vide pour représenter la pile vide :

```
def pile_vide():
    return []
```

— On utilise la routine `append` pour empiler :

```
def empiler(pile, a):
    pile.append(a)
    return
```

— On code la fonction `est_vide` en évitant une directe comparaison de liste :

```
def est_vide(pile):
    return len(pile) == 0
```

— On dépile grâce à la routine `pop` :

```
def depiler(pile):
    if est_vide(pile):
        return None
    return pile.pop()
```

Remarque

Les fonctions `empiler` et `depiler` sont "en place", c'est à dire qu'elles modifient directement la pile donnée en entrée par l'utilisateur. C'est ce que demande l'énoncé. C'est ce qui fera la difficulté des fonctions que l'on demande d'implémenter ensuite puisqu'on précise qu'elles ne **doivent pas** modifier la pile donnée en entrée par l'utilisateur.

2) Pour copier une pile sans modifier la pile donnée par l'utilisateur il va nous falloir une pile de reconstruction afin de pouvoir reconstruire la pile de l'utilisateur :