Exercices - Capes 2017 - première épreuve / option informatique : corrigé

Avertissement : Ceci n'est pas une correction *in extenso* du problème de capes. Il s'agit plutôt d'une lecture personnelle des questions, avec des indications, des idées de preuve, des mises en garde d'erreurs à éviter. Ce n'est surtout pas une correction modèle à reproduire... Pour signaler toute erreur, merci d'écrire à devgeolabo@gmail.com

Le premier problème aborde la résolution d'une grille de Sudoku. Il s'agit vraiment d'un problème de programmation sous Python, où on demande de corriger, modifier, créer des fonctions en vue de la résolution d'un problème de Sudoku. Le second problème propose l'étude de deux algorithmes pour déterminer l'enveloppe convexe d'un nombre fini de points. Il comporte des thématiques plus évoluées, comme des calculs de complexité ou des preuves de terminaison d'algorithme. Il est à mon avis nettement plus difficile que le premier ! Je m'interroge aussi sur la pertinence de faire écrire des fonctions plutôt complexes sans pouvoir les tester sur un ordinateur. Pour plusieurs questions, j'ai été très content de travailler avec mon ordinateur pour pouvoir me corriger !

Premier problème

Partie A : Résultats préliminaires

1. Si une grille de Sudoku est complète, alors la somme de chacune des lignes, chacune des colonnes, chacun des carrés de taille 3x3 est égale à la somme de tous les entiers de 1 à 9, c'est-à-dire 45. La réciproque est fausse : la grille constituée uniquement de 5 vérifie la propriété que la somme de chacune des lignes, chacune des colonnes et chacun des carrés est 45. Ce n'est pas une grille complète de Sudoku !
2. Il suffit de vérifier que chacun des chiffres de 1 à 9 est dans la ligne (si tous y sont, la ligne est complète, si l'un n'y est pas, la ligne n'est pas complète). Ceci donne par exemple le code suivant :

```
def ligne_complete(L,i):
    ligne_teste=L[i]
    for n in range(1,10):
        if n not in ligne_teste:
            return False
    return True
```

3. On teste toutes les lignes, toutes les colonnes et tous les carrés pour savoir s'ils sont complets. On obtient :

```
def complet(L):
    for i in range(0,9):
        if ( not(ligne_complete(L,i)) or not(colonne_complete(L,i))
               or not (carre_complet(L,i)) ):
            return False
    return True
```

Partie A : Fonctions annexes

4. Je trouve la question assez mal formulée. Si la ligne comporte deux fois le même chiffre, s'attend-on à ce que l'on renvoie une liste comportant une seule fois ou deux fois ce chiffre ? Une solution possible est

```
def ligne(L,i):
    chiffre=[]
    for j in range(0,9):
        if ( (L[i][j] not in chiffre) and (L[i][j]!=0) ):
            chiffre.append(L[i][j])
    return chiffre
```

   Si on ne prend pas garde aux chiffres éventuellement redondants, on pourra se passer de la première partie du test. Il faut bien faire attention ici à ce que la boucle se fait pour les entiers j de 0 à 8.

5. La case en haut à gauche d'un carré 3x3 a pour coordonnées $(3m, 3n)$, avec $0 \leq m \leq 2$ et $0 \leq n \leq 2$. La case en haut à gauche du carré contenant (i, j) a pour coordonnées $(3m, 3n)$ si et seulement si $0 \leq i - 3m \leq 2$ et $0 \leq j - 3n \leq 2$. Autrement dit, $i - 3m$ est le reste dans la division euclidienne de i par 3, et $j - 3n$ est le quotient dans la division euclidienne de j par 3. Le réel m est donc le quotient dans la division euclidienne de i par 3, c'est-à-dire que $m = \left[\frac{i}{3}\right]$ et donc $3m = 3 \times \left[\frac{i}{3}\right]$. De même, $3n = 3 \times \left[\frac{j}{3}\right]$.

6. On complète comme à la question 4. Remarquez que par rapport à l'énoncé, j'ai changé le nom des variables apparaissant dans les boucles. C'est une très mauvaise pratique d'algorithmique et programmation d'utiliser dans une même fonction une variable avec des significations différentes.

```
def carre(L,i,j):
    icoin=3*(i//3)
    jcoin=3*(j//3)
    chiffre=[]
    for k in range(icoin,icoin+3):
        for l in range(jcoin,jcoin+3):
            if ( (L[k][l] not in chiffre) and (L[k][l]!=0) ):
                chiffre.append(L[k][l])
    return chiffre
```

7. Il suffit de concaténer les listes obtenues précédemment.

```
def conflit(L,i,j):
    return ligne(L,i)+colonne(L,j)+carre(L,i,j)
```

   Je n'ai pas vraiment compris ce que voulait dire l'énoncé par "On ne prendra pas en compte la valeur de $L[i][j]$".

8. Là encore, l'énoncé commet une erreur ! On ne peut pas utiliser sous Python une variable qui porte le même nom qu'une fonction ! L'auteur du sujet a-t-il posé son sujet en travaillant sous Python en même temps ? ? ? ? J'ai donc ajouté un s à la variable conflits. Pour le reste, c'est presque la même question que précédemment !

```
def chiffres_ok(L,i,j):
    ok=[]
    conflits=conflit(L,i,j)
    for k in range(1,10):
        if k not in conflits:
            ok.append(k)
    return oks
```

Partie B : Algorithme naïf

9. Il suffit de compter le nombre d'éléments de la liste donnée par `chiffres_ok`.

```
def nb_possibles(L,i,j):
    return len(chiffres_ok(L,i,j))
```

10. Il est dit dans le rapport de l'épreuve que le jury a été particulièrement attentif à cette question. La fonction proposée comporte au moins 3 erreurs :
    — Les test d'égalité se font avec == et non avec =
    — Le premier élément d'une liste est le terme d'indice [0]
    — La variable booléenne changement n'est pas mise à jour en cas de changement dans la grille.

    Le code correct devient

```
def un_tour(L):
    changement=False
    for i in range(9):
        for j in range(9):
            if (L[i][j]==0):
                if (nb_possibles(L,i,j)==1):
                    L[i][j]=chiffres_ok(L,i,j)[0]
                    changement=True
    return changement
```

1. On répète la fonction `un_tour` jusqu'à ce qu'il n'y ait plus de changements.

```
def complete(L):
    continuer=un_tour(L)
    while continuer:
        continuer=un_tour(L)
    return complet(L)
```

Partie C : Backtracking

12. Si on a comme entrée (i, j), on doit renvoyer $(i, j+1)$ sauf si $j = 8$. Dans ce cas, on renvoie $(i+1, 0)$. Ceci donne

```
def case_suivante(pos):
    if (pos[1]==8):
```

```
        return [pos[0]+1,0]
    else:
        return [pos[0],pos[1]+1]
```

13. Les commentaires permettent de comprendre le code ajouté.

```
def backtracking(L,pos):
    if (pos==[9,0]):
                # on a resolu le sudoku!
        return True
    i,j=pos[0],pos[1]
    if L[i][j]!=0:
        # la case est deja remplie
        # on passe a la suivante
        return backtracking(L,case_suivante(pos));
    for k in chiffres_ok(L,i,j):
        # on essaie tous les chiffres possibles pour la case
        L[i][j] = k
        if (backtracking(L,case_suivante(pos))):
            return True
    L[i][j]=0
        # si on est arrive ici, c'est que ca ne fonctionne pas.
        # On reinitialise la grille
        # Et on retourne faux.
    return False
```

14. La fonction `backtracking` est appelée au plus 9 fois pour chaque case vide. Comme il y a $81-p$ cases vides, la fonction `backtracking` est appelée au plus 9^{81-p} fois.
15. La fonction `solution_sudoku(L)` renvoie `True` dès que la grille L admet au moins une solution. Si elle en admet plusieurs, par exemple si on part de la grille vide, elle retournera a fortiori `True`.
16. Question incompréhensible ! Dans la version distribuée le jour du concours, la question portait sur le nombre d'appels à la **fonction** `pos`, fonction qui n'existe pas. Dans la version diffusée sur le site du concours, la question porte sur le nombre d'appels à la **variable** `pos`, mais est-ce vraiment cela qui prend du temps ? Je pense plutôt qu'il fallait comprendre le nombre d'appels à la fonction `backtracking`. Il y a plusieurs stratégies possibles pour limiter le nombre d'appels. D'abord, on peut d'abord faire appel à l'algorithme naïf pour remplir les cases où une seule solution est possible. On peut aussi, plutôt que d'utiliser l'ordre lexicographique, trier les cases par nombre de chiffres possibles, et commencer par les cases avec le plus petit nombre possible. Mais comme il faudrait faire ceci à chaque appel de la fonction `backtracking`, difficile de savoir si on gagnerait réellement du temps. Pour info, l'exécution sur ma machine de `solution_sudoku` aux grilles proposées par l'énoncé est quasi-immédiate.

Pour info, voici un code possible pour les autres fonctions qui n'étaient pas demandées par l'énoncé :

```
def colonne_complete(L,i):
    colonne_testee=[]
    for j in range(0,9):
        colonne_testee.append(L[j][i])
    for n in range(1,10):
        if n not in colonne_testee:
            return False
        return True

def carre_complet(L,i):
    coiny=3*(i//3)
    coinx=3*(i-coiny)
    carre_teste=[]
    for k in range(0,3):
        for j in range(0,3):
            carre_teste.append(L[coiny+k][coinx+j])
    for n in range(1,10):
        if n not in carre_teste:
            return False
        return True
```

Deuxième problème

Partie A : préliminaires

1.
```
def distance(P,Q):
    return sqrt((P[0]-Q[0])**2+(P[1]-Q[1])**2)
```

2.a. La fonction `distance` est exécutée une fois au début de la fonction `distance_minimale`, puis à chaque fois que l'on est dans la double boucle `while`. Lorsque $i = 0$, la boucle sur j est exécutée n fois. Puis, comme j n'est pas réinitialisée à 0, la boucle en j n'est plus jamais exécutée. On fait donc appel $n + 1$ fois à la fonction `distance`.

2.b. Lors de l'entrée dans la double boucle, on a $i = 0$ et $j = 0$. Ainsi, on calcule `distance(L[0],L[0])` qui est nulle, et on ne peut pas faire moins. Ainsi, la fonction `distance_minimale` renvoie toujours 0.

2.c Voici une correction possible, qui tient compte des remarques précédentes, et qui utilise une boucle Pour plutôt qu'une boucle Tant Que puisqu'on connait toujours les bornes d'arrêt. En particulier, on ne teste la distance entre deux points $L[i]$ et $L[j]$ que si $j < i$.

```
def distance_minimale(L):
    n=len(L)
    minimum=distance(L[0],L[1])
    for i in range(n):
        for j in range(i):
                a=distance(L[i],L[j])
```

```
                if (a<minimum):
                    minimum=a
    return minimum
```

3. On procède comme précédemment, mais on n'a plus besoin d'initialiser avec une distance entre deux points. Il suffit de l'initialiser avec une valeur négative, puisque dès la première itération de la boucle, on aura une distance supérieure. Ceci donne

```
def distance_maximale(L):
    n=len(L)
    maximum=-1
    for i in range(n):
        for j in range(i):
            a=distance(L[i],L[j])
            if (a>maximum):
                maximum=a
                imax=i
                jmax=j
    return (maximum,jmax,imax)
```

Pour $i=0$, la fonction `distance` est appelée 0 fois. Pour $i=1$, la fonction `distance` est appelée 1 fois. Pour $i=2$, la fonction `distance` est appelée 2 fois. Plus généralement, pour $i=k$, j prend toutes les valeurs de 0 à $k-1$ et la fonction `distance` est appelée k fois. Finalement, la fonction `distance` est appelée $1+2+\cdots+n-1=\frac{n(n-1)}{2}$ fois.

4.
```
def point_abs_min(L):
    n=len(L)
    absmin=L[0][0]
    imin=0
    for i in range(1,n):
        if (L[i][0]<absmin):
            absmin=L[i][0]
            imin=i
    return imin
```

La boucle étant appelée $n-1$ fois, et puisqu'on ne fait appel à l'intérieur de cette boucle qu'à des opérations élémentaires, la complexité temporelle de la fonction est en $O(n)$ (il n'est pas précisé quelle est la complexité temporelle de la fonction `len` mais il est raisonnable qu'elle ne peut pas être pire que $O(n)$). Selon le rapport du jury, environ 31% des candidats ont traité correctement cette question, 41% ont fourni une réponse incomplète ou incorrecte, et 27% n'ont pas traité la question.

5. En utilisant les propriétés du déterminant, on a

$$\begin{aligned}\det(\overrightarrow{QR},\overrightarrow{QP}) &= \det(\overrightarrow{QP}+\overrightarrow{PR},\overrightarrow{QP})\\ &= \det(\overrightarrow{QP},\overrightarrow{QP})+\det(\overrightarrow{PR},\overrightarrow{QP})\\ &= -\det(\overrightarrow{PR},\overrightarrow{PQ})\\ &= \det(\overrightarrow{PQ},\overrightarrow{PR}).\end{aligned}$$

Ainsi, (Q, R, P) est aussi en sens direct. En revanche, puisque

$$\det(\overrightarrow{PR}, \overrightarrow{PQ}) \quad = \quad -\det(\overrightarrow{PQ}, \overrightarrow{PR})$$

le triplet (P, R, Q) est en sens indirect.

6.
```
def orientation(P,Q,R):
    a=Q[0]-P[0]
    b=Q[1]-P[1]
    c=R[0]-P[0]
    d=R[1]-P[1]
    det=a*d-b*c
    if (det>0):
        return 1
    elif (det==0):
        return 0
    else:
        return -1
```

7.a A chaque itération de la boucle, on parcourt la liste entre l'indice i et le dernier indice (en partant du dernier), et on déplace vers le début un élément qui est plus grand que l'élément qui le précède immédiatement. On obtient donc
- Après $i = 0$, on obtient `L=[1,5,2,3,4]`
- Après $i = 1$, on obtient `L=[1,2,5,3,4]`
- Après $i = 2$, on obtient `L=[1,2,5,3,4]`
- Après $i = 3$, on obtient `L=[1,2,3,4,5]`
- Après $i = 4$, on obtient `L=[1,2,3,4,5]`

7.b. Démontrons cette propriété par récurrence finie sur k.
Initialisation : la propriété $\mathcal{P}(0)$ est clairement vraie.
Hérédité : soit $k \in \{0, \dots, n-1\}$ telle que la propriété $\mathcal{P}(k)$ est vraie, et prouvons que $\mathcal{P}(k+1)$ est vraie. Alors la $k+1$-ième itération laisse les k premiers éléments de L invariants. De plus, elle change les $n-k$ derniers éléments de sorte de mettre en position $k+1$ le plus petit de ces éléments. Alors, les $k+1$ premiers éléments sont bien triés, puisque les k premiers l'étaient déjà, et qu'on a mis en position $k+1$ un élément qui était plus grand qu'eux. De plus, puisque le $(k+1)$ième élément est plus petit que les suivants, les $k+1$ premiers éléments sont inférieurs aux $n-(k+1)$ derniers. Ainsi, $\mathcal{P}(k+1)$ est vraie.
Conclusion : Par le principe de récurrence (finie), $\mathcal{P}(n)$ est vraie, et donc les n premiers éléments de la liste, c'est-à-dire tous les éléments de la liste, sont triés en ordre croissant.

7.c. Quelle que soit la configuration initiale, les instructions à l'intérieur des deux boucles sont exécutées $\frac{n(n-1)}{2}$ fois (comme à la question 3.). Comme il n'y a qu'un nombre fini d'instructions élémentaires exécutées à chaque itération, la complexité de la fonction, dans le meilleur comme dans le pire des cas, est en $O(n^2)$.

8.a. On parcourt les deux listes en même temps afin de déterminer un premier rangement. On s'arrête lorsqu'une des deux listes est épuisée. On ajoute alors les termes qui restent de l'autre liste.

```
def fusion(L1,L2):
    n1=len(L1)
    n2=len(L2)
    i,j=0,0
    L=[]
    while ( (i<n1) and (j<n2)):
        if (L1[i]<L2[j]):
            L.append(L1[i])
            i=i+1
        else:
            L.append(L2[j])
            j=j+1
    # On ajoute encore les termes de la liste non epuisee
    for i in range(i,n1):
        L.append(L1[i])
    for j in range(j,n2):
        L.append(L2[j])
    return L
```

8.b. On prouve par récurrence forte sur n que la fonction `tri_fusion` se termine pour toute liste de taille n.
Initialisation : La fonction se termine pour une liste de taille 0 ou 1, en retournant exactement cette liste.
Hérédité : Soit $n \geq 1$. Supposons que la fonction `tri_fusion(L)` s'arrête pour toute liste de taille inférieure ou égale à n, et prouvons qu'elle s'arrête également pour toute liste de taille $n+1$. Soit L une telle liste et $m = (n+1)//2$. Alors, puisque $n+1 \geq 2$, on a $1 \leq m \leq n$. Lors de l'appel de `tri_fusion(L)`, on fait alors successivement appel à
— `tri_fusion(L1)`, où $L1$ est une liste de taille $m \leq n$. Par hypothèse de récurrence, cette fonction se termine.
— `tri_fusion(L2)`, où $L2$ est une liste de taille $(n+1) - m \leq n$. Par hypothèse de récurrence, cette fonction se termine.
— `fusion(L1,L2)` : cette fonction se termine.
Donc la fonction `tri_fusion(L)` se termine, et la propriété est prouvée au rang $n+1$.
Conclusion : Par le principe de récurrence forte, la fonction `tri_fusion(L)` se termine toujours, quelle que soit la longueur de la liste L.

8.c. On peut d'abord remarquer que les complexités dans le meilleur et dans le pire des cas de la fonction `tri_fusion` sont identiques : en effet, un appel à `tri_fusion` avec une liste de longueur 2^p entraîne deux appels à la fonction `tri_fusion` avec des listes de longueur 2^{p-1}, et ainsi de suite, et ce peu importe la liste de départ de longueur 2^p.
Notons ensuite C_p le nombre d'opérations élémentaires nécessaires à la fonction `tri_fusion` pour trier une liste de taille 2^p. Si $p = 0$, alors $C_0 = 2$ (affectation d'une valeur à n, puis test). Pour $p \geq 1$, on a

$$C_p \leq 3 + 2C_{p-1} + a2^p.$$

où la constante a vient l'appel à la fonction `fusion`. Remarquons que, pour $p \geq 1$, on a

$3 \leq 2 \times 2^p$. En posant $b = a + 2$, on a donc

$$C_p \leq 2C_{p-1} + b2^p.$$

Prouvons alors par récurrence sur $p \geq 0$ que, pour tout $p \geq 0$,

$$C_p \leq 2^p C_0 + bp2^p.$$

La propriété est vraie pour $p = 0$. Soit $p \geq 1$ et supposons que la propriété est vraie au rang $p - 1$, c'est-à-dire que

$$C_{p-1} \leq 2^{p-1} C_0 + b(p-1)2^{p-1}.$$

Alors

$$C_p \leq 2^p C_0 + b(p-1)2^p + b2^p = 2^p + bp2^p.$$

Ainsi, la propriété est démontrée au rang p et par le principe de récurrence, on a bien démontré que pour tout $p \geq 0$, on a

$$C_p \leq 2^p C_0 + bp2^p.$$

Ainsi, la complexité de la fonction `tri_fusion` pour une liste de taille $n = 2^p$ est $O(p2^p)$, c'est-à-dire $O(n \log n)$.

Partie B : Enveloppe convexe d'un nuage de points

9.a. Les lignes 16 à 20 permettent de déterminer, i et j étant fixés, si les triplets $(L[i], L[j], L[k])$ ont tous la même orientation, pour $k \neq i, j$, c'est-à-dire si $[L[i]L[j]]$ est un côté de l'enveloppe convexe.

9.b. La fonction consiste en trois boucles imbriquées, chaque boucle étant parcourue n fois. Comme les autres opérations ont toutes une complexité en $O(1)$, la complexité temporelle de cette fonction est en $O(n^3)$.

9.c. Le script trace une ligne brisée entre les différents sommets de l'enveloppe convexe. Mais ces sommets ne sont pas forcément dans l'ordre qu'il faudrait, et on obtient pas nécessairement l'enveloppe convexe voulue. Imaginer par exemple un parallélogramme $ABCD$ pour lesquels on tracerait les segments $[AC]$, puis $[CD]$, puis $[DB]$.

10. On détermine l'ordre en prenant une demi-droite d'origine P_6 et en la faisant dans le sens trigonométrique, en commençant par le point le plus petit. On voit ici facilement que c'est P_0, car tous les triplets (P_6, P_0, P_i) sont tous en sens direct. On obtient finalement :

$$P_0 \succ_{P_6} P_4 \succ_{P_6} P_1 \succ_{P_6} P_7 \succ_{P_6} P_2 \succ_{P_6} P_5 \succ_{P_6} P_8 \succ_{P_6} P_3$$

11.

```
def prochain_sommet(L,i):
        if (i==0):
                jmin=1
        else:
                jmin=0
        for j in range(len(L)):
                if orientation(L[i],L[j],L[jmin])>0:
                        jmin=j
        return jmin
```

La complexité est bien en $O(n)$ puisqu'on parcourt n fois une boucle composée d'instructions élémentaires.

12.
```
def jarvis2(L):
        i=point_abs_min(L)
        suivant=prochain_sommet(L,i)
        Enveloppe=[i]
        while (suivant!=i):
                Enveloppe.append(suivant)
                suivant=prochain_sommet(L,suivant)
        return Enveloppe
```

Remarquer la place de `Enveloppe.append(suivant)` et l'initialisation `Enveloppe=[i]` afin d'éviter de répéter deux fois le sommet i dans la liste des sommets retournés.

13. On exécute N fois une boucle, et les instructions à l'intérieur de cette boucle ont une complexité en $O(n)$. Ainsi, la complexité de `jarvis2` est en $O(n \times N)$.
14. Laissé au lecteur !