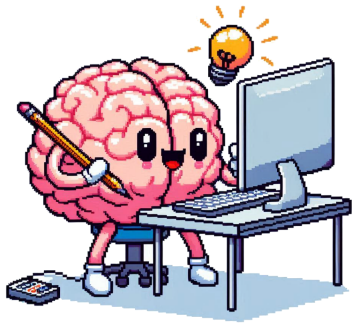
Informatique PT. 2024 - 2025

Mathématiques & Informatique - F. Gaunard
http://frederic.gaunard.com
Lycée Voltaire, Paris 11e.



Devoir surveillé n°1

Solution

Partie I : Introduction

Les étudiants d'une classe de CPGE ont pris l'habitude de se rendre en salle d'études pour travailler. Chaque étudiant ne peut aller dans cette salle qu'une seule fois par jour, mais peut y rester autant de temps qu'il le désire. La période de travail d'un étudiant est donc caractérisée par deux quantités : son instant d'arrivée et son instant de départ.

Dans ce sujet un instant de la journée est représenté par un unique entier égal au nombre de secondes s'étant écoulées depuis minuit. Par exemple, si un étudiant arrive en salle d'études à l'instant 45296 , cela signifie qu'il est arrivé à 12 h 34 m 56 s. En effet :

$$45296 = 12 \times 3600 + 34 \times 60 + 56.$$

Les instants d'arrivées et de départs des différents étudiants sont stockés dans un dictionnaire dont les clés sont des chaînes de caractères (les noms des étudiants) et dont les valeurs sont des couples d'entiers `(deb, fin)` où `deb` est l'instant d'arrivée de l'étudiant et `fin` est son instant de départ. Un tel dictionnaire sera appelé un *planigramme*.

Par exemple le dictionnaire `plani_0` est un *planigramme* :

```
plani_0 = {
'Yasmine' : (47704,51650), 'Sophie' : (50400,54000), 'Emile' : (48104,49545),
'Garance': (48104,52050), 'Esteban' : (48104,49500), 'Thomas' : (49200,50400)
}
```

Après division euclidienne par 3600 puis 60, on obtient les horaires suivants pour ce dictionnaire :

- ✗ Yasmine : arrivée à 13h15m04s et départ à 14h20m50s
- ✗ Sophie : arrivée à 14h00m00s et départ à 15h00m00s
- ✗ Emile : arrivée à 13h21m44s et départ à 14h27m30s
- ✗ Garance : arrivée à 13h21m44s et départ à 14h27m30s
- ✗ Esteban : arrivée à 13h21m44s et départ à 13h45m00s
- ✗ Thomas : arrivée à 13h40m00s et départ à 14h00m00s.

1. a. Donnez une des lignes affichée par le programme suivant :

```
for c in plani_0.keys() :
  deb,fin=plani_0[c]
  print (c," est arrivé(e) à l'instant " ,deb, " et parti(e) à l'instant ",fin)
```

Solution. On obtient par exemple la ligne :

```
Yasmine  est arrivé(e) à l'instant  47704  et parti(e) à l'instant  51650
```

□

b. Qu'affiche le programme suivant?

```
print('Esteban' in plani_0.keys())
print(48104 in plani_0.values())
print((48104, 49500) in plani_0.values())
```

Solution. La commande `in` teste l'appartenance d'une clé à un dictionnaire, donc le code renverra successivement `True`, puis `False` et enfin `True`. □

c. Qu'affiche le programme suivant?

```
print([c for c in plani_0 if plani_0[c][0]== plani_0['Emile'][0]])
```

Solution. Ce code renvoie la liste des noms des étudiants qui sont arrivés en même temps qu'Émile. En l'occurrence `['Emile', 'Garance', 'Esteban']`. □

Les horaires présents dans le planigramme étant renseignés manuellement, on souhaite vérifier la cohérence des heures d'arrivée et de départ. Pour que le planigramme soit cohérent, il faut que :

- ✗ les instants d'arrivée et de départ soient positifs ou nuls;
- ✗ l'instant d'arrivée de chaque étudiant soit inférieur ou égal à son instant de départ;
- ✗ chaque étudiant soit parti de la salle d'études au plus tard à 23h59m59s.

2. a. Écrire une fonction `est_coherent(plani)` qui indique si le planigramme donné en entrée est cohérent (en renvoyant `True` or `False`).

Solution. Sans difficulté.

```
def est_coherent(plani):
    heure_max=23*3600+59*60+59
    for c in plani :
        deb,fin = plani[c]
        if deb<0 or fin<0:
            return False
        if deb>fin :
            return False
        if fin>heure_max :
            return False
    return True
```

□

b. Quelle est la complexité de votre fonction `est_coherent`? On exprimera le résultat en fonction de la taille n du dictionnaire, et on justifiera sa réponse.

Solution. Puisque l'on parcourt une seule fois le dictionnaire, la complexité de la fonction est en $O(n)$ où n est la taille du dictionnaire. □

La manière dont sont représentés les instants dans un planigramme n'est pas très lisible pour un humain. On souhaite donc construire un dictionnaire dans lequel chaque instant `inst` est donné par un triplet d'entiers `(h,m,s)` où `h` est le nombre d'heures, `m` le nombre de minutes et `s` le nombre de secondes qui se sont écoulées depuis minuit. Par exemple, à partir du planigramme `plani_0`, on obtient le dictionnaire :

```
{'Yasmine' : ((13,15,4),(14,20,5)), 'Sophie' : ((14,00,00),(15,00,00)),
'Emile' : ((13,21,44),(13,45,45)), 'Garance': ((13,21,44),(14,27,30)),
'Esteban' : ((13,21,44),(13,45,00)), 'Thomas' : ((13,40,00),(14,00,00))}
```

3. Écrire une fonction `convertir(plani)` qui renvoie un dictionnaire dans lequel les instants sont donnés sous le format heures-minutes-secondes, comme dans l'exemple ci-dessus. Votre fonction ne doit pas modifier le dictionnaire `plani` donné en entrée.
On rappelle que `n//p` renvoie le quotient dans la division euclidienne de `n` par `p`, et `n%p` renvoie le reste dans cette même division.

Solution. On commence par écrire une fonction `conversion` qui transforme un nombre de secondes (écoulées depuis minuit) en une heure.

```
def conversion(n):
    h=n//3600
    n=n%3600
    m=n//60
    s=n%60
    return (h,m,s)

def convertir(plani):
    plani1={}
    for c in plani.keys() :
        deb,fin = plani[c]
        plani1[c]=(conversion(deb),conversion(fin))
    return plani1
```

□

Partie II : Étudiant.e ayant le plus travaillé

On souhaite déterminer le ou les étudiant.e.s ayant passé le plus de temps dans la salle d'études. Par exemple, pour le planigramme `plani_0`, les étudiant.e.s ayant étudié le plus longtemps sont Yasmine et Garance puisqu'elles sont restées 3946 secondes en salle d'études, contre 3600 secondes pour Sophie, 1441 secondes pour Émile, 1396 secondes pour Esteban et 1200 secondes pour Thomas.

4. Écrire une fonction `plus_long(plani)` qui prend en entrée un planigramme et renvoie la liste du (de la) ou des étudiant.e.s qui ont passé le plus de temps en salle d'études.
Par exemple, à partir du dictionnaire `plani_0`, votre fonction doit renvoyer `["Yasmine", "Garance"]` (l'ordre des éléments de la liste n'a pas d'importance). Si le dictionnaire est vide, votre fonction renverra la liste vide.

Solution. On adapte la méthode des candidats au cas qui nous occupe.

```
def plus_long(plani):
    Liste_nom=[]
    t_max=0
    for c in plani.keys() :
        deb,fin = plani[c]
        if fin-deb>t_max:
            Liste_nom=[c]
            t_max=fin-deb
        elif fin-deb==t_max:
            Liste_nom.append(c)
    return Liste_nom
```

□

Les étudiants se sont rendus compte que le temps passé en salle d'études n'est pas le seul critère pour déterminer si le travail a été fructueux : l'entraide entre étudiants est un facteur prédominant dans la réussite en classe prépa. Deux étudiant.e.s ont pu s'entraider s'ils ont passé au moins une seconde ensemble dans la salle d'études. On appelle *score d'entraide* d'un.e étudiant.e le nombre d'autres étudiant.e.s avec lesquels il ou elle a pu s'entraider. Par exemple, avec le planigramme `plani_0`, le score d'entraide de Sophie est 2 car elle était dans la salle d'études en même temps que Yasmine et Garance.

5. Écrire une fonction `score_entraide(plani)` qui renvoie un dictionnaire `d_SE` ayant les mêmes clés que la planigramme `plani`, et tel que pour toute clé `c`, l'entier `d_SE[c]` est le score d'entraide de l'étudiant.e `c`.
Le planigramme donné en entrée ne doit pas être modifié par votre fonction.
Par exemple pour le planigramme `plani_0`, le dictionnaire `d_SE` vaut:

```
{"Yasmine": 5, "Sophie": 2, "Emile": 4, "Garance": 5, "Esteban": 4, "Thomas": 4}
```

Solution. Il y a deux parcours imbriqués du dictionnaire, puisque chaque étudiant doit être comparé à tous les autres. Attention à la gestion des cas d'égalité : si un étudiant arrive exactement au moment où un autre part, on ne considérera pas qu'ils se sont entraidés.

```
def score_entraide(plani):
    d_SE={}
    for c in plani.keys() :
        deb,fin = plani[c]
        score=0
        for c2 in plani.keys() :
            deb2,fin2 = plani[c2]
            if c2!=c and deb2<fin and deb<fin2:
                score=score+1
        d_SE[c]=score
    return d_SE
```

À la place de la condition `c2!=c and deb2<fin and deb<fin2`, on aurait pu utiiser `deb<=deb2<fin or deb<fin2<=fin or deb2<=deb<=fin<=fin2 or deb<=deb2<=fin2<=fin` conviendrait également. □

Partie III : Fierté d'un.e étudiant.e

Un.e étudiant.e aime souvent bien montrer aux autres qu'il ou elle travaille plus qu'eux. Il ou elle est particulièrement fier.e lorsqu'il ou elle arrive en salle d'études avant un.e autre étudiant.e et qu'il ou elle en repart plus tard. On appelle *score de fierté* d'un.e étudiant.e E_1 le nombre d'étudiant.e.s $E_2 \neq E_1$ tels que :

- ✗ E_1 est arrivé avant ou en même temps que E_2,
- ✗ et E_1 est parti après ou en même temps que E_2.

Par exemple, le niveau de fierté de Yasmine est de 3 grâce à Émile, Esteban et Thomas qui sont tous les trois arrivés après elle et partis avant elle. Notre but est de déterminer l'étudiant ayant la plus grande fierté. Pour cela, il sera utile d'avoir une fonction capable de trier une liste d'entiers.

Étant donnée une liste `L`, on souhaite créer une liste triée `T` ayant les mêmes éléments que `L`. La liste `T` ne contiendra pas non plus d'élément en double.
Ainsi, si `L` contient deux fois le même élément, il n'apparaitra qu'une seule fois dans `T`. Pour cela, on va utiliser le *tri par insertion* dont voici le principe :

- ✗ Initialement, la liste `T` est vide.
- ✗ Pour chaque élément `e` de `L` :
  - ◆ On définit une variable `i` de la manière suivante :
    - ✷ si `T` est vide, on pose `i=0`;
    - ✷ si `e` apparaît dans `T`, on pose `i=None`;
    - ✷ sinon, si tous les éléments de `T` sont strictement inférieurs à `e`, on pose `i=len(T)`;
    - ✷ sinon, `i` est le plus petit indice tel que `T[i]>e`.
  - ◆ Si `i` vaut `None` alors on ne fait rien.
  - ◆ Sinon, on insère `e` à l'indice `i` dans `T`.

6. a. Écrire une fonction `insertion(T, e, i)` qui insère l'élément `e` dans `T` à l'indice `i`.
Votre fonction renverra la liste ainsi obtenue et ne devra pas modifier la liste initiale T. Si la condition $0 \leqslant i \leqslant \text{len}(T)$ n'est pas respectée, votre fonction déclenchera une erreur.

Solution. On peut soit faire un code à la main, soit utiliser une concaténation.
L'instruction `assert` de `Python` est une aide au débogage qui teste une condition. Si la condition est vraie, cela ne fait rien et votre programme continue simplement à s'exécuter. Mais si la condition d'assertion est fausse, elle lève une exception `AssertionError`.
Ici, c'est demandé, donc on le fait.

```
def insertion(T,e,i):
    assert 0<=i and i <=len(T)
    return T[0:i]+[e]+T[i:len(T)]
```

ou bien, sans concaténation,

```
def insertion(T,e,i):
    assert 0<=i and i <=len(T)
    T2=[ ]
    for k in range(i):
        T2.append(T[k])
    T2.append(e)
    for k in range(i,len(T)):
        T2.append(T[k])
    return T2
```

□

b. Quelle est la complexité de la fonction insertion? Justifier.

Solution. La fonction `insertion` est en $O(p)$, où p est la longueur de la liste `T`, puisqu'on parcourt une seule fois cette liste. □

7. Écrire une fonction `indice_insertion` qui prend en entrée une liste triée d'entiers `T`, ainsi qu'un entier `e`, et renvoie l'entier `i` comme défini dans la description du tri par insertion. On utilisera une recherche dichotomique, puisque `T` est triée.

Solution. Le plus simple est d'utiliser un code récursif. La recherche dichotomique garantit une complexité logarithmique en $O(\ln(\texttt{len(T)}))$.

```
def indice_insertion(T,e):
    n=len(T)
    if n==0:
        return 0
    if e>T[n-1]:
        return n
    else:
        p=n//2
        if e==T[p]:
            return None
        elif e<T[p]:
            return indice_insertion(T[0:p],e)
        else:
            i=indice_insertion(T[p+1:n],e)
            if i==None:
                return None
            else :
                return (p+1)+i
```

□

8. a. À l'aide des questions précédentes, écrire une fonction `tri_insertion(L)` qui renvoie la liste triée `T` obtenue en appliquant un tri par insertion sur `L`.

Solution. Tout est mis en place pour que ça s'écrive tout seul.

```
def tri_insertion(L):
    T=[ ]
    for e in L:
        i=indice_insertion(T,e)
        if i!=None:
            T=insertion(T,e,i)
    return T
```

□

b. Quelle est la complexité de la fonction `tri_insertion`? Justifier. Comment pourrait-on améliorer cette fonction ? On ne demande pas de donner un code plus performant.

Solution. On obtient une complexité en $O(p^2)$, où p est la longueur de la liste `L`. Cela provient du fait qu'on parcours `L` dans `tri_insertion`, puis du fait que `insertion` a une complexité en $O(p)$. On s'est donc fatigué à écrire une code de complexité logarithmique pour `indice_insertion` en pure perte... Pour gagner du temps dans l'insertion, le mieux serait d'utiliser un algorithme de tri en place, au lieu de recopier la liste à chaque étape (on renvoie au **Cours d'informatique n°1**). □

On souhaite maintenant déterminer le score de fierté maximal parmi les étudiant.e.s. Étant donné un.e étudiant.e E, on note $d(E)$ l'instant auquel E est arrivé.e en salle d'études et $f(E)$ l'instant auquel E est parti.e de cette salle. Les valeurs du planigramme sont donc tous les couples $(d(E), f(E))$. Pour tout étudiant.e E_1, on note :

- ✗ $m_1(E_1)$ le nombre d'étudiant.es $E_2 \neq E_1$ tels que $d(E_2) < d(E_1)$;
- ✗ $m_2(E_1)$ le nombre d'étudiant.es $E_2 \neq E_1$ tels que $f(E_2) > f(E_1)$.

Dans un premier temps, on va calculer les quantités $m_1(E)$ et $m_2(E)$ pour chaque étudiant.e E. Pour cela, on commence par créer un dictionnaire `dict_deb` dont les clés sont tous les instants d'arrivées des étudiant.e.s. Pour toute clé `inst`, la valeur `dict_deb[inst]` est une liste contenant tou.te.s les étudiant.e.s qui sont arrivé.e.s à l'instant `inst`. Par exemple à partir du planigramme `plani_0`, on obtient:

```
dict_deb0 = {47704: ['Yasmine'], 50400: ['Sophie'],
48104: ['Emile', 'Garance', 'Esteban'], 49200: ['Thomas'] }
```

9. Écrire une fonction `make_dict_deb(plani)` qui prend en entrée un planigramme et renvoie le dictionnaire `dict_deb`.

Solution.

```
def make_dict_deb(plani):
    dict_deb={}
    for c in plani.keys() :
        deb=plani[c][0]
        if not deb in dict_deb:
            dict_deb[deb]=[c]
        else:
            dict_deb[deb].append(c)
    return dict_deb
```

□

On note maintenant `T_deb` la liste triée contenant toutes les clés de `dict_deb`. Par exemple, pour le dictionnaire `dict_deb0`, on a : `T_deb0=[47704,48104,49200,50400]`.

10. a. Écrire une fonction

`make_dict_m1_aux(dict_deb, T_deb)`

qui prend en entrée le dictionnaire `dict_deb` ainsi que la liste triée `T_deb` et qui renvoie un dictionnaire `dict_m1` tel que :

- ✗ les clés de `dict_m1` sont les noms des étudiant.e.s;
- ✗ à chaque étudiant.e E, la valeur associée à E dans `dict_m1` est $m_1(E)$.

Solution.

```
def make_dict_m1_aux(dict_deb,T_deb):
    dict_m1={}
    m1=0
    for deb in T_deb :
        for c in dict_deb[deb]:
            dict_m1[c] = m1
        m1=m1+len(dict_deb[deb])
    return dict_m1
```

□

b. En déduire une fonction `make_dict_m1(plani)` qui prend en entrée un planigramme et renvoie le dictionnaire `dict_m1` décrit dans la question précédente.

Solution.

```
def make_dict_m1(plani) :
    dict_deb=make_dict_deb(plani)
    L_deb=[deb for deb in dict_deb.keys()]
    T_deb=tri_insertion(L_deb)
    return make_dict_m1_aux(dict_deb,T_deb)
```

□

En suivant le même principe que dans la Question **10.** on peut obtenir une fonction

`make_dict_m2(plani)`

qui prend en entrée un planigramme et renvoie un dictionnaire `dict_m2` tel que :

- ✗ les clés de `dict_m2` sont les noms des étudiant.e.s;
- ✗ à chaque étudiant.e E, la valeur associée à E dans `dict_m2` est $m_2(E)$.

Dans la suite, on suppose que la fonction `make_dict_m2` a été écrite. Vous pouvez donc l'utiliser sans la réécrire.

Pour tout étudiant E_1, on note :

- ✗ $n_0(E_1)$ le nombre d'étudiant.e.s $E_2 \neq E_1$ tels que $d(E_2) < d(E_1)$ et $f(E_2) > f(E_1)$;
- ✗ $n_1(E_1)$ le nombre d'étudiant.e.s $E_2 \neq E_1$ tels que $d(E_2) < d(E_1)$ et $f(E_2) \leqslant f(E_1)$;
- ✗ $n_2(E_1)$ le nombre d'étudiant.e.s $E_2 \neq E_1$ tels que $d(E_2) \geqslant d(E_1)$ et $f(E_2) > f(E_1)$;
- ✗ $n_3(E_1)$ le nombre d'étudiants $E_2 \neq E_1$ tels que $d(E_2) \geqslant d(E_1)$ et $f(E_2) \leqslant f(E_1)$, donc $n_3(E_1)$ est le score de fierté de E_1;
- ✗ $n(E_1) = n_0(E_1) + n_1(E_1) + n_2(E_1) + n_3(E_1)$.

11. **a**. Soit E un.e étudiant.e. Exprimer $m_1(E)$ et $m_2(E)$ en fonction des $n_i(E)$.

Solution. On a clairement $m_1(E) = n_0(E) + n_1(E)$ et $m_0(E) = n_0(E) + n_2(E)$. □

b. Expliquer comment calculer $n(E)$ à partir du planigramme. Votre méthode devra s'exécuter en temps constant.

Solution. $n(E)$ est égal au nombre d'étudiants différents de E, il se calcule donc avec la commande `len(plani)-1`. □

c. Pour chaque étudiant E, on calcule $\Delta(E) = n(E) - m_1(E) - m_2(E)$. Soit M le maximum des $\Delta(E)$. Montrer que M est le score de fierté maximal parmi les étudiant.e.s.

Solution. Pour tout étudiant E, on voit que $n_3(E)$ est le score de fierté de E. Alors :

$$\begin{aligned}\Delta(E) &= n(E) - m_1(E) - m_2(E)\\ &= n_0(E) + n_1(E) + n_2(E) + n_3(E) - (n_0(E) + n_1(E)) - (n_0(E) + n_2(E))\\ &= n_3(E) - n_0(E)\\ &\leq n_3(E)\end{aligned}$$

On voit ainsi que le score de fierté $n_3(E)$ de chaque étudiant majore $\Delta(E)$.
Soit maintenant E_0 un étudiant dont le niveau de fierté $n_3(E_0)$ est maximal, et montrons que $n_0(E_0) = 0$. En effet, si $n_0(E_0) > 0$ alors il existe un étudiant E_1 qui est arrivé avant et parti après E_0 en salle d'études : on a donc $n_3(E_1) > n_3(E_0)$ ce qui contredit la *maximalité* de $n_3(E_0)$.
En résumé, pour tout étudiant E, on a $\Delta(E) \leq n_3(E)$ et $\Delta(E_0) = n_3(E_0)$. On en conclut que $M = \Delta(E_0)$ est bien le score de fierté maximal parmi les étudiants. □

12. À l'aide du résultat de la Question **11.c.** et des fonctions précédentes, écrire une fonction

`score_fierte_max (plani)`

qui renvoie le score de fierté maximal parmi les étudiant.e.s.

Solution. Il suffit de construire le dictionnaire contenant toutes les valeurs de $\Delta(E)$ (et dont les clés sont les noms des étudiants), puis de calculer le maximum de celui-ci.

```
def max_dict(d):
    maxi=0
    for c in d:
        if maxi<d[c]:
            maxi=d[c]
    return maxi
```

```
def score_fierte_max ( plani ) :
    dict_delta={}
    n=len(plani)-1
    dict_m1=make_dict_m1(plani)
    dict_m2=make_dict_m2(plani)
    for c in plani :
        dict_delta[c]=n-dict_m1[c]-dict_m2[c]
    return max_dict(dict_delta)
```

□