IC Devoir 5 - Vendredi 17 mai 11h30 à 12h30

Exercice 1

Soit un entier naturel n non nul et une liste t de longueur n dont les termes valent 0 ou 1. Le but de cet exercice est de trouver le nombre maximal de 0 contigus dans t (c'est-à-dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste t1 suivante vaut 4 :

i	0	1	2	3	4	5	6	7
t1[i]	0	1	1	1	0	0	0	1

i	8	9	10	11	12	13	14
t1[i]	0	1	1	0	0	0	0

1. Écrire une fonction nombreZeros(t,i), prenant en paramètres une liste t, de longueur n, et un indice i compris entre 0 et $n-1$.

   - Si t[i] vaut 1 alors la fonction renvoie 0.
   - Sinon, elle renvoie le nombre de zéros consécutifs dans t à partir de t[i] inclus.

Par exemple, les appels nombreZeros(t1,4), nombreZeros(t1,1) et nombreZeros(t1,8) renvoient respectivement les valeurs 3, 0 et 1.

2. Comment obtenir le nombre maximal de zéros contigus à partir du calcul des nombreZeros(t,i) pour $0 \leq i \leq n-1$?

En déduire une fonction nombreZerosMax(t), de paramètre t, renvoyant le nombre maximal de 0 contigus d'une liste t non vide. On utilisera la fonction nombreZeros.

3. Quelle est la complexité de la fonction nombreZerosMax construite à la question précédente ? *(cas le meilleur et cas le pire)*

4. Trouver un moyen simple d'obtenir un algorithme plus performant tout en utilisant la fonction nombreZeros. Donner un script de cette amélioration et estimer sa complexité.

Exercice 2 *Tri par sélection* **Méthode**

➜ Le tri par sélection consiste à :
- pour i variant sur les indices de la liste (de longueur n), en partant de la gauche :
- on recherche un minimum parmi les $n-i$ derniers éléments de la liste
- on le place en position i par un échange d'éléments

1. Écrire le script d'une fonction pmin(L,a,b) donnant la position d'un minimum de la sous-liste L[a:b].
2. Écrire le script d'une procédure tri_selection(L).
3. Définir un invariant de boucle de la boucle principale de pmin.
4. Définir un invariant de boucle de la boucle principale de tri_selection.
5. Étudier la complexité temporelle de tri_selection.
6. Appliquer le tri par sélection à la liste [6,1,2,4,2,1,5].

Tableau à adapter et compléter :

i	pmin	L
X	X	[6,1,2,4,2,1,5]
0	pmin(L,0,6) → ...	
⋮	⋮	⋮

Exercice 3 *Tri par comptage* **Méthode**

➜ Le tri par comptage consiste à :
- sachant que les éléments d'une liste sont dans $[\![0,N]\!]$, on dénombre les éventuels 0, les 1, ... les N.
- on construit une liste comportant autant de 0 suivis d'autant de 1, ... et d'autant de N que la liste de départ.

Soit $N \in \mathbb{N}^*$. On considère que les éléments de la liste L sont dans $[\![0,N]\!]$.

1. Donner le script d'une fonction stat(L,N) qui retourne une liste de $N+1$ éléments contenant à la position k le nombre d'occurrences de k dans L.

stat([1,0,0,2,1],3) retourne [2,2,1,0]

2. Que retournerait l'appel stat([2,1,1,0,1,4],4) ?
3. Donner le script d'une fonction tri_comptage(L,N) qui retourne une liste triée associée à L.
4. Définir un invariant de boucle de la boucle principale de stat.
5. Définir un invariant de boucle de la boucle principale de tri_comptage.
6. Étudier la complexité temporelle de tri_comptage.

Exercice 4 *Déplacement d'un mobile*

Les sommets d'un carré sont numérotés 1, 2, 3, et 4 de telle façon que les côtés du carré relient le sommet 1 au sommet 2, le sommet 2 au sommet 3, le sommet 3 au sommet 4 et le sommet 4 au sommet 1.

Un mobile se déplace aléatoirement sur les sommets de ce carré selon le protocole suivant :
- Au départ, c'est à dire à l'instant 0, le mobile est sur le sommet 1.
- Lorsque le mobile est à un instant donné sur un sommet, il se déplace à l'instant suivant sur l'un quelconque des trois autres sommets, et ceci de façon équiprobable.

Pour tout $n \in \mathbb{N}$, on note X_n la variable aléatoire égale au numéro du sommet sur lequel se situe le mobile à l'instant n. D'après le premier des deux points précédents, on a donc $X_0 = 1$.

Compléter la fonction suivante, ou en proposer une autre qui retourne la liste de n premières positions autres que celle d'origine et le nombre N de fois où il est revenu sur le sommet numéroté 1 au cours de ses n premiers déplacements.

```
1  def mobile(n):
2      p,L,N=1,[],0
3      for i in range(n):
4          r=...
5          if r==p:
6              p=...
7          else:
8              p=...
9          L.append(p)
10         if p==1:
11             ...
12     return L,N
```


Lycée Saint-Exupery – maffre.saintexupery@gmail.com – MPSI

Exercice 5 *Urne de Polya*
On dispose d'une urne contenant au départ 1 boule blanche et 3 boules noires. On dispose également d'une réserve infinie de boules blanches et de boules noires.
Pour tout entier naturel j, on dit que l'urne est dans l'état j lorsqu'elle contient j boules blanches et $(j+2)$ boules noires. Au départ, l'urne est donc dans l'état 1.

On réalise une succession d'épreuves, chaque épreuve se déroulant selon le protocole suivant :
Pour tout entier naturel j, si l'urne est dans l'état j, on extrait une boule au hasard de l'urne.
- Si l'on obtient une boule blanche, alors cette boule n'est pas remise dans l'urne et on enlève de plus une boule noire de l'urne. L'urne est alors dans l'état $(j-1)$.
- Si l'on obtient une boule noire, alors cette boule est remise dans l'urne et on remet en plus une boule blanche et une boule noire dans l'urne. L'urne est alors dans l'état $(j+1)$.

La variable aléatoire X_n donne le nombre de boules blanches présentes dans l'urne après la n-ème épreuve. Compléter le programme suivant ou donner une autre fonction qui simule l'expérience aléatoire décrite et qui retourne X_n.

```
1 def Polya(n):
2     X=...
3     for i in range(...):
4         tirage=rd.randint(...)
5         if tirage<=X:
6             ...
7         else:
8             ...
9     return X
```

Exercice 6
Lister les tris que vous connaissez et pour chacun :
- Donner un rapide descriptif
- Donner sa complexité dans le meilleur des cas avec la description d'une liste correspondant à cette situation.
- Donner sa complexité dans le pire des cas avec la description d'une liste correspondant à cette situation.
- Préciser si le tri est au programme ou pas.

IPT Devoir 5- Proposition de solutions

Solution 1 Longueur maximale d'une série de zéros

1. La fonction nombreZeros :

```
def nombreZeros(t,i):
    z=0
    while i<len(t) and t[i]==0:
        z,i=z+1,i+1
    return z
```

En particulier :

```
--> t=[0,1,1,1,0,0,0,1,0,1,1,0,0,0,0]
--> nombreZeros(t,4)
3
--> nombreZeros(t,1)
0
--> nombreZeros(t,8)
1
```

Avec une boucle for cela donnerait :

```
def nombreZeros(t,i):
    z=0
    for k in range(i,len(t)):
        if t[k]==1:
            break
        else:
            z=z+1
    return z
```

2. Il s'agit de faire le calcul du maximum des valeurs de `nombreZeros(t,i)` pour toutes les valeurs possibles de `i` :

```
def nombreZerosMax(t):
    n=0
    for i in range(len(t)):
        nbz=nombreZeros(t,i)
        if nbz>n:
            n=nbz
    return n
```

Ce qui donne :

```
--> nombreZerosMax(t)
4
```

3. Analyse de la complexité de `nombreZerosMax` : notons n la longueur de la liste `t`
- une boucle parcourant la liste `t`, indexée par i
- pour i fixé, l'appel `nombreZeros(t,i)` requiert entre 1 et $n-i$ comparaisons (et deux affectations)

Dans le meilleurs des cas : $\sum_{i=0}^{n-1} 1 = n$.

Dans le pire des cas :

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

La complexité de `nombreZerosMax` est en $\mathcal{O}(n^2)$.

4. Il suffit d'intégrer au parcours de la liste `t` l'information obtenue à chaque appel de `nombreZeros` : on peut effectuer un saut afin d'ignorer des valeurs ne pouvant pas donner le début d'une liste maximale de zéros.
Sur l'exemple donné, on note `nbZ(t,i)` les appels "*utiles*" :

i	0	1	2	3	4	5	6	7
t1[i]	0	1	1	1	0	0	0	1
nbZ(t,i)	1		0	0	3			

i	8	9	10	11	12	13	14
t1[i]	0	1	1	0	0	0	0
nbZ(t,i)	1		0	4			

```
def nombreZerosMax2(t):
    z=nombreZeros(t,0)
    nb,i=z,z+1
    while i<len(t)-nb:
        z=nombreZeros(t,i)
        i=i+z+1
        if z>nb:
            nb=z
    return nb
```

En particulier :

```
--> nombreZerosMax2(t)
4
```

Pour tester des listes aléatoires de longueur 40, suivre les instruction suivantes :

```
--> import numpy.random as rd
--> t=list(rd.randint(0,2,size=40))
--> t
--> nombreZerosMax2(t)
```

➜ Analyse de complexité de la nouvelle version : les deux boucles imbriquées (celle du parcours de la liste `t` et celle des appels de `nombreZeros`) se complète pour effectuer un simple parcours de `t`. Il y a donc n étapes. A chaque étape, il y a une ou deux comparaisons et une ou quatre affections.
La complexité de `nombreZerosMax2` est en $\mathcal{O}(n)$.

Lycée Saint-Exupery – maffre.saintexupery@gmail.com – MPSI

Solution 2 Tri par sélection

1. Un script de la fonction pmin :

```
def pmin(L,a,b):
    v=L[a]
    p=a
    for i in range(a+1,b):
        if L[i]<v:
            v,p=L[i],i
    return p
```

2. Un script de la fonction tri_selection :

```
def tri_selection(L):
    n=len(L)
    for i in range(n-1):
        p=pmin(L,i,n)
        if p>i:
            L[i],L[p]=L[p],L[i]
```

Ici la liste L est traitée comme une variable globale en PYTHON.

3. Les paramètres de la boucle sont a, b, L, v et p.
Un invariant de la ième étape de pmin est : "v et p contiennent respectivement la valeur et la position du premier minimum des éléments de L entre les positions a et i-1".

4. Un invariant de la ième étape de tri_selection est : "les i plus petits éléments de la liste sont ordonnés sur les i premières positions".

$$\texttt{L[0]} \leq \texttt{L[1]} \leq \cdots \leq \texttt{L[i-1]} \leq \texttt{L[j]} \text{ pour tout } j \geq i$$

5. Soit n la taille de la liste.
Quelle que soit la liste, le nombre de comparaisons est toujours le même. Une boucle pour i variant de 0 à n-2 et pour chaque i un appel de pmin(L,i,n) qui coûte n-i étapes :

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

Le nombre d'affectations est du même ordre.
La complexité du tri par sélection est : $\mathcal{O}(n^2)$.

6. Tri appliqué à L=[6,1,2,4,2,1,5] :

i	pmin	L
X	X	[6,1,2,4,2,1,5]
0	pmin(L,0,6) → 1	[1,6,2,4,2,1,5]
1	pmin(L,1,6) → 5	[1,1,2,4,2,6,5]
2	pmin(L,2,6) → 2	[1,1,2,4,2,6,5]
3	pmin(L,3,6) → 4	[1,1,2,2,4,6,5]
4	pmin(L,4,6) → 4	[1,1,2,2,4,6,5]
5	pmin(L,5,6) → 6	[1,1,2,2,4,5,6]

Solution 3 **Tri par comptage**

1. La fonction stat :

Dénombrement par élément

```
def stat(L,N):
    C=(N+1)*[0]
    for e in L: C[e]=C[e]+1
    return C
```

2. L'appel stat([[2,1,1,0,1,4],4) retourne [1,3,1,0,1].

3. Tri par comptage :

Tri par comptage

```
def tri_comptage(L,N):
    C=stat(L,N)
    T=[]
    for i,e in enumerate(C): T=T+e*[i]
    return T
```

4. Un invariant de la ième étape de tri_comptage est : "tous les éléments inférieur strictement à i, de L, sont ordonnés et constitue la liste T".

5. La fonction stat crée une liste de longueur $N+1$ et parcourt la liste L une fois. Le nombre d'affectation est de $N+1+n$ avec n la longueur de L.
La boucle for reconstitue élément par élément la liste triée et donc il y a n affectation et la longueur de la boucle est N.
Au final, la complexité est en $\mathcal{O}(N+n)$

Solution 4 Déplacement d'un mobile : à chaque saut, l'expérience consiste à choisir au hasard parmi les trois autres sommets. Il s'agit donc de générer 3 valeurs aléatoires avec équiprobabilité d'où l'instruction

r=rd.randint(1,4)

Ainsi, r est une simulation de la loi uniforme sur $[\![1,3]\!]$.
La variable p contient la position actuelle du mobile. Par disjonction de cas, on peut vérifier que l'alternative des lignes 5 à 8 donne bien un des trois autres sommets à l'étape suivante :

- Si p vaut 4 alors r prend la valeur d'une des trois autres sommets.
- Si p vaut 1 alors r peut prendre la valeur 1, 2 ou 3. S'il prend la valeur 1, alors on la remplace par 4. Au final, p prend une valeur parmi {2,3,4} avec équiprobabilité.
- Idem si p prend la valeur 2 ou 3.

La variable N compte le nombre de retour sur le sommet 1.

```
1  def mobile(n):
2      p,L,N=1,[],0
3      for i in range(n):
4          r=rd.randint(1,4)
5          if r==p:
6              p=4
7          else:
8              p=r
9          L.append(p)
10         if p==1:
11             N=N+1
12     return L,N
```

Solution 5 Urne de Polya

```
1  def Polya(n):
2      X=1
3      for i in range(1,n+1):
4          tirage=rd.randint(1,2*X+3)
5          if tirage<=X:
6              X=X-1
7          else:
8              X=X+1
9      return X
```

La ligne 4 traduit le tirage aléatoire dans une urne dans l'état X, c'est-à-dire contenant 2X+2 boules dont X blanches.

Solution 6 Les différents tris :

▶ Tri par sélection **Méthode**

➜ Le tri par sélection consiste à :
- pour i variant sur les indices de la liste (de longueur n), en partant de la gauche :
- on recherche un minimum parmi les $n-i$ derniers éléments de la liste
- on le place en position i par un échange d'éléments
- Dans tous les cas, la complexité est $\mathcal{O}(n^2)$.

- Ce tri est au programme.

▶ Tri par comptage **Méthode**

➜ Le tri par comptage consiste à :
- sachant que les éléments d'une liste sont dans $[\![0,N]\!]$, on dénombre les éventuels 0, les 1, ... les N.
- on construit une liste comportant autant de 0 suivis d'autant de 1, ... et d'autant de N que la liste de départ.
- Dans tous les cas, la complexité est $\mathcal{O}(n+N)$.
- Ce tri est au programme.

▶ Tri par comparaison **Méthode**

➜ Le tri par comparaison consiste à :
- pour chaque élément, on parcoure la liste et détermine sa position dans la liste triée : la position de L[i] est le nombre d'éléments $j \neq i$ tels que :
  - L[j]<=L[i] lorsque $j < i$,
  - L[j]<L[i] lorsque $j > i$,
- enfin, on place chaque élément à sa place dans la liste triée.
- Dans tous les cas, la complexité est $\mathcal{O}(n^2)$.
- Ce tri n'est pas au programme.

▶ Tri par insertion **Méthode**

➜ Le tri par **insertion** consiste à :
- parcourir la liste de gauche à droite ;
- à chaque étape, l'élément considéré, est classé parmi les éléments qui le précèdent (et donc qui sont déjà ordonnés).
- Dans le cas le meilleur, la complexité est $\mathcal{O}(n)$. C'est le cas d'une liste déjà triée.
- Dans le cas le pire, la complexité est $\mathcal{O}(n^2)$. C'est le cas d'une triée dans l'ordre décroissant.
- Ce tri n'est pas au programme.

▶ Tri rapide **Méthode**

➜ Le tri rapide consiste à :
- si la liste contient au plus un élément, alors on retourne la liste ; sinon, on considère le premier élément ;
- on crée deux sous-listes, une contenant les éléments qui lui sont inférieurs et l'autre pour les autres éléments (ceux qui lui sont strictement supérieurs) ;
- on applique le tri aux deux sous-listes obtenues et on concatène le résultat.
- Dans le cas le meilleur, la complexité est $\mathcal{O}(n\ln(n))$. C'est le cas d'une liste où chaque découpe se fait en deux sous listes de tailles similaires.
- Dans le cas le pire, la complexité est $\mathcal{O}(n^2)$. C'est le cas d'une liste triée.
- Ce tri est au programme.

▶ Tri fusion **Méthode**

➜ Le tri fusion consiste à :
- si la liste contient au plus un élément, alors on retourne la liste ; sinon, on la découpe en deux sous-listes de tailles comparables ;
- on applique le tri sur chacune d'elle et on fusion les résultats
- la fusion est une sous fonction qui permet de constituer une liste triée à partir de deux listes déjà triées
- Dans tous les cas, la complexité est $\mathcal{O}(n\ln(n))$.
- Ce tri est au programme.

▶ Tri à bulles **Méthode**

➜ Le tri à bulles consiste à :
- on parcourt la liste de gauche à droite ;
- dès que deux éléments consécutifs sont mal ordonnés, on les échange ;
- on recommence le parcours jusqu'à ce que la liste soit triée.
- Dans le cas le meilleur, la complexité est $\mathcal{O}(n)$. C'est le cas d'une liste déjà triée.
- Dans le cas le pire, la complexité est $\mathcal{O}(n^2)$. C'est le cas d'une liste triée dans l'ordre inverse.
- Ce tri n'est pas au programme.