Lycée César Baggio — PTSI — 2021/2022
Devoir de contrôle d'informatique
Durée 2h

- Sans document, sans calculatrice ou autre dispositif électronique.
- Lorsque l'on demande d'écrire un code `Python`, on prendra soin de numéroter les lignes et de faire apparaître très clairement l'indentation.

Exercice 1. Déterminer la valeur retournée par chacun des appels demandés.

```
1 def f(x):
2     if x % 2 == 0:
3         return 3
4     if x % 4 == 0:
5         return 5
6     else:
7         return 7
```

1. `f(3)`
2. `f(6)`
3. `f(8)`
4. `f(1)`

Exercice 2. Déterminer la valeur retournée par chacun des appels demandés.

```
1 def f(x):
2     if x % 2 == 0:
3         a = 3
4     if x % 4 == 0:
5         a = 5
6     else:
7         a = 7
8     return a
```

1. `f(3)`
2. `f(6)`
3. `f(8)`
4. `f(1)`

Exercice 3. Déterminer la valeur retournée par chacun des appels demandés.

```
1 def f(x):
2     if x % 2 == 0:
3         a = 3
4     elif x % 4 == 0:
5         a = 5
6     else:
7         a = 7
8     return a
```

1. `f(3)`
2. `f(6)`
3. `f(8)`
4. `f(1)`

Exercice 4. Expliquer pourquoi les instructions suivantes renvoient des messages d'erreur.

1.

```
def f(x):
    a = 0
    if x % 2 == 0:
        a = 0
    else x % 4 == 0:
        a = 1
    return a
```

2.

```
def g(x):
    a = 0
    if x % 2 = 0:
        a = 0
    else:
        a = 1
    return a
```

Exercice 5. On donne la fonction `lapin`

```
def lapin(n):
    a = 0
    b = n + 1
    while a + 1 < b:
        x = (a + b) // 2
        if x * x <= n:
            a = x
        else:
            b = x
        print(a, b, x)
    return b - 1
```

1. Donner les valeurs affichées par l'instruction `print` lors de l'appel `lapin(52)`.

2. Quelle valeur retourne la fonction lors de l'appel `lapin(52)` ?

3. Donner les valeurs affichées par l'instruction `print` lors de l'appel `lapin(0)`.

4. Quelle valeur retourne la fonction lors de l'appel `lapin(0)` ?

Exercice 6. Si x et y sont deux nombres réels quelconques, on définit une fonction prenant comme variables x et y de la façon suivante

$$f(x,y) = \begin{cases} x^2 + y - 3 & \text{si } x > y \\ x^2 - y^2 + 12 & \text{si } x < y \\ x - 1 & \text{si } x = y. \end{cases}$$

1. Écrire une fonction `f` prenant comme paramètres deux flottants («nombres à virgule») `x` et `y` et calculant $f(x,y)$.
2. On définit une suite $\left(u_n\right)_{n\in\mathbb{N}}$ par $u_0 = 0$ et la relation

$$\forall n \in \mathbb{N}, u_{n+1} = f\left(u_n^2, u_n\right).$$

   Écrire une fonction `carotte` ayant pour argument un entier naturel `n` et que retourne la valeur de u_n.

Exercice 7.

1. Écrire une fonction `nb_chiffres` ayant pour arguments deux entiers `n` et `b` avec $n \geq 1$ et $b \geq 2$ et qui retourne l'entier p tel que

$$b^{p-1} \leq n < b^p.$$

   On utilisera uniquement les opérations sur les entiers sans import de bibliothèque.
2. Écrire un ensemble de tests fonctionnels pour votre fonction `nb_chiffres`.

Exercice 8. Écrire une fonction `seconddegre` avec trois paramètres `a`, `b`, `c`, et qui renvoie la liste contenant les racines réelles du trinôme du second degré

$$aX^2 + bX + c$$

- Préconditions : `a`, `b`, `c` sont trois flottants et `a` est non nul.

On obtiendra par exemple

Python 3 Shell

```
>>> seconddegre(1, -5, 6)
[2.0, 3.0]
>>> seconddegre(1, -2, 1)
[1.0]
>>> seconddegre(1, 1, 1)
[]
```

Exercice 9. On donne trois nombre a, b, c et on souhaite les ordonner dans l'ordre croissant. Voici donc mon programme:

```
1  def tri(a, b, c):
2      """
3      Retourne la liste triée contenant les trois nombres a, b, c
4      """
5      if a > b:
6          x = a
7          a = b
8          b = x
9      if b > c:
10         x = b
11         b = c
12         c = x
13     if a > c:
14         x = a
15         a = c
16         c = x
17
18     return [a, b, c]
```

Je l'essaie avec $a = 3$, $b = 1$ et $c = 2$.

Python 3 Shell

```
>>> tri(3, 1, 2)
[1, 2, 3]
```

1. Que font les lignes 5-6-7-8 dans le programme précédent ?
2. Écrire un ensemble de tests fonctionnels «crédible» `test_tri` pour ma fonction de tri.
3. Lesquels de vos tests échouent ? Expliquez ?

Exercice 10. Prenons un nombre entier naturel n non nul. Calculons la somme des carrés de ses chiffres. On obtient un nouveau nombre. Recommençons cette opération avec ce nouveau nombre, on obtient un troisième nombre.

Que se passe-t-il si on itère ces opérations plusieurs fois? Observons deux exemples.

- Avec $n = 19$, on obtient successivement

$$19 \to 82 \to 68 \to 100 \to 1 \to 1 \dots$$

et le 1 se répète indéfiniment.

- Avec $n = 200$, on obtient successivement

$$200 \to 4 \to 16 \to 37 \to 58 \to 89 \to 145 \to 42 \to 20 \to 4 \dots$$

et le cycle $4 \to 16 \to 37 \to 58 \to 89 \to 145 \to 42 \to 20$ se répète indéfiniment.

On peut démontrer qu'en appliquant un tel processus, à partir d'un entier quelconque non nul, on finit toujours par arriver sur l'une ou l'autre de ces deux situations, c'est-à-dire à arriver sur le nombre 1 ou sur le nombre 4.

On dira qu'un nombre entier naturel non nul n est **heureux** si le processus mène à 1, et **malheureux** s'il mène à 4.

1. Quelles sont les opérateurs Python permettant d'obtenir le quotient et le reste de la division euclidienne d'un entier `n` par un autre entier `p`.

2. Écrire une fonction `somme_carres` qui renvoie la somme des carrés des chiffres de l'entier passé en paramètre.

Python 3 Shell

```
>>> somme_carres(19)
82
>>> somme_carres(200)
4
```

3. Après avoir pris soin de bien numéroter les lignes de votre fonction `somme_carres`, simuler l'exécution de l'appel `somme_carres(3407)`. On présentera l'évolution des valeurs prises par vos variable sous forme d'un tableau. Préciser à chaque fois le numéro de la ligne exécutée.

4. Réalisez un prédicat (c'est-à-dire une fonction qui retourne un booléen) nommé `est_heureux` qui renvoie `True` si l'entier passé en paramètre est heureux, et `False` sinon.

Python 3 Shell

```
>>> est_heureux(19)
True
>>> est_heureux(200)
False
```

5. On appelle **altitude**, le terme maximum atteint par la suite obtenue à partir de l'entier *n*. Par exemple, l'altitude de 19 est 100 et l'altitude de 200 est 200.

   Écrire une fonction `altitude` ayant pour paramètre un entier naturel `n` et qui retourne l'altitude partant de `n`.

Python 3 Shell

```
>>> altitude(19)
100
>>> altitude(200)
200
```

Lycée César Baggio — PTSI — 2021/2022
Devoir de contrôle d'informatique
Durée 2h

- Sans document, sans calculatrice ou autre dispositif électronique.
- Lorsque l'on demande d'écrire un code `Python`, on prendra soin de numéroter les lignes et de faire apparaître très clairement l'indentation.
- La complexité, ou le temps d'exécution, d'un programme `P` est le nombre d'opérations élémentaires nécessaire à l'exécution de `P`. Lorsqu'il est demandé de donner une certaine complexité, cette dernière sera donnée par une borne supérieure asymptotique (en $O(\dots)$) *et sera justifiée*.

Lecture de code

Exercice 1. Voici une fonction écrite en `Python`

```
1 def mystere(a):
2     s = 0
3     b = a
4     while b > 0:
5         s = s + b % 10
6         b = b // 10
7     return s
```

1. Exprimez en français ce que font respectivement les lignes 5 et 6.
2. Donnez, sous forme d'un tableau dont les colonnes représentent les variables `s` et `b` les valeurs successives de ces deux variables lors du calcul de `mystere(3209)`.
3. Lorsque `a` est un nombre entier positif ou nul, que représente la valeur de l'expression `mystere(a)`?
4. Quelle est la valeur de l'expression `mystere(a)` lorsque `a` est un entier négatif?
5. Comment est affectée la valeur retournée si l'on intervertit l'ordre des deux lignes 5 et 6 dans la fonction `mystere`?
6. Réécrivez la fonction `mystere` de sorte que l'expression `help(mystere)` donne une documentation utile au programmeur.
7. Écrire un jeu de tests adaptées à la fonction `mystere`.

Exercice 2. On considère la fonction `caroline` suivante ayant pour paramètre une liste de nombres `A`:

```
def caroline(A):
    n = len(A)
    for i in range(1, n):
        j = i
        while j > 0 and A[j - 1] > A[j]:
            A[j - 1], A[j] = A[j], A[j - 1]
            j = j - 1
            print(i, j, A)
```

1. Donner la suite de lignes affichées par l'instruction `print(i, j, A)` lors de l'appel

   Python 3 Shell

   ```
   >>> caroline([5, 11, 2, 4, 1, 8])
   ```

2. Donner la suite de lignes affichées par l'instruction `print(i, j, A)` lors de l'appel

   Python 3 Shell

   ```
   >>> caroline([8, 5, 3, 1])
   ```

3. Que fait la fonction `caroline` ?

4. Déterminer la complexité en temps dans le pire cas de la fonction `caroline` en fonction de `n = len(A)`.

5. Déterminer la complexité en temps dans le meilleur cas de la fonction `caroline` en fonction de `n = len(A)`.

Exercice 3. On considère la fonction suivante qui calcule la racine carrée entière par défaut

```
1  def racine(n):
2      """
3      Donnée: n : nombre dont on veut calculer la racine (entier)
4      Résultat: r : racine carrée entière par défaut (entier >= 0)
5
6      Pré condition: n >= 0
7      Post condition: r * r <= n < (r + 1) * (r + 1)
8      """
9      r = 0
10     y = 1
11     z = 1
12     while y <= n:
13         z = z + 2
14         y = y + z
15         r = r + 1
16     return r
```

1. Démontrer la validité du triplet de Hoare suivant:

$$\{\ y \leq n \text{ et } y = (r+1)^2 \text{ et } z = 2r+1 \text{ et } r^2 \leq n\ \}$$
```
z = z + 2
y = y + z
r = r + 1
```
$$\{\ y = (r+1)^2 \text{ et } z = 2r+1 \text{ et } r^2 \leq n\ \}$$

2. Démontrer que, si ce programme termine, il calcule la racine carrée entière par défaut (preuve partielle).

3. Donner un variant (une fonction de terminaison) pour la boucle (12–15). Montrer que ce programme termine.

4. Donner un plan de test pour cette fonction.

5. Écrire une fonction `test_racine` conforme à votre plan de test.

Écriture de code

Exercice 4. Si un train fait un trajet aller-retour entre 2 villes à la vitesse constante v_1 pour l'aller et à la vitesse constante v_2 au retour, la vitesse moyenne du trajet total n'est pas la moyenne arithmétique des deux vitesses, mais leur moyenne harmonique

$$\frac{2}{\frac{1}{v_1}+\frac{1}{v_2}}$$

Pour une série statistique discrète $x = (x_1, \dots, x_p)$, formée de p nombres > 0, la moyenne harmonique est définie par

$$G = \frac{p}{\frac{1}{x_1}+\frac{1}{x_2}+\cdots+\frac{1}{x_p}} = \frac{p}{\sum_{i=1}^{p}\frac{1}{x_i}}.$$

Écrire une fonction `moyenne_harmonique` retournant la moyenne harmonique des éléments d'une liste non vide. On supposera la liste constituée uniquement de nombres strictement positifs.

Exercice 5. On considère un fichiers de notes d'étudiants `liste_etudiants.csv` dont voici un extrait:

```
11400130|Sarah|Croche|9.6,18.9
11402978|Gilbert|Lingot|14.2,14.2
11403526|Jean|Rage|10.3,9.7
11500571|Chantal|Gique|10.2,9.8
11502148|Jerry|Kahn|10.7,10.2
11503156|Aubin|Sahalor|15.1,13.9
11503188|Oscar|Ibou|11.2,12.8
11503442|Patrice|Taisse|8.2,4.1
11505350|Alphonse|Urlotorouth|4.4,17.3
90000002|Terry|Dicule|0.0,6.2
90000003|Anne|Aconda|5.5,7.8
90000004|Théo|Jasmin|20.0,4.1
98001398|Ursule|Fureux|14.7,11.1
99990125|Harry|Zona|11.8,10.2
99990179|José|Patelefaire|7.5,6.3
```

Chaque ligne contient le numéro d'identification de l'étudiant (NIP), son prénom, son nom, sa note à l'épreuve de mathématique, sa note à l'épreuve d'informatique. Ces différent champs sont séparés par le caractère `|` , à l'exception des notes qui sont séparées par une virgule. Ce fichier est trié par NIP croissant.

1. Écrire un code `Python` ouvrant le fichier `liste_etudiants.csv` et qui définit quatre listes comme variables globales.

   - `liste_NIP`, la liste des NIP des étudiants (nombres entiers)
   - `liste_nom`, la liste des noms des étudiants sous la forme `"Prénom Nom"`
   - `liste_note_math`, la liste des notes de mathématique (nombres flottants).
   - `liste_note_info`, la liste des notes d'informatique (nombres flottants).

   L'ordre des éléments de ces listes étant celui du fichier. On obtiendra donc les listes suivantes (seul les premiers termes sont représentés.

   Python 3 Shell

   ```
   >>> liste_NIP
   [11400130, 11402978, 11403526, ...
   >>> liste_nom
   ['Sarah Croche', 'Gilbert Lingot', 'Jean Rage', ...
   >>> liste_note_math
   [9.6, 14.2, 10.3, ...
   >>> liste_note_info
   [18.9, 14.2, 9.7, ...
   ```

2. Écrire une fonction `recherche_NIP` ayant pour argument une chaine de caractère `nom` représentant le nom l'étudiant sous la forme `"Prénom Nom"`. Cette fonction retourne

- le numéro d'identification étudiant si celui-ci apparait sur la liste,
- l'entier `0` sinon.

Python 3 Shell

```
>>> recherche_NIP("Aubin Sahalor")
11503156
>>> recherche_NIP("Ray Défess")
0
```

3. Écrire une fonction `recherche_position` ayant pour argument un entier `NIP`, un numéro d'étudiant valide, et qui retourne la position de ce `NIP` dans la liste `liste_NIP`.

Puisque la liste `liste_NIP` est triée par ordre monotone croissant, *on utilisera une recherche dichotomique*.

Python 3 Shell

```
>>> recherche_position(11403526)
2
```

4. Expliciter un invariant permettant de prouver que votre fonction `recherche_position` est correcte (on ne demande pas les détails de la preuve).

5. Écrire une fonction `admis` ayant pour argument un flottant `barre`. Cette fonction retourne la liste des noms des étudiants dont la moyenne des notes de math et d'informatique dépasse la valeur `barre`.

Python 3 Shell

```
>>> admis(14)
['Sarah Croche', 'Gilbert Lingot', 'Aubin Sahalor']
>>> admis(18)
[]
>>> admis(10)
['Sarah Croche', 'Gilbert Lingot', 'Jerry Kahn',
'Aubin Sahalor', 'Oscar Ibou', 'Alphonse Urlotorouth',
'Théo Jasmin', 'Ursule Fureux', 'Harry Zona']
>>>
```

6. Quelle est la complexité en temps de la fonction `admis` ?

Lycée César Baggio — PTSI 1&2 — 2021/2022
Devoir de contrôle d'informatique
Concours Blanc
Durée 2h

- Sans document, sans calculatrice ou autre dispositif électronique.
- Lorsque l'on demande d'écrire un code `Python`, on prendra soin de numéroter les lignes et de faire apparaître très clairement l'indentation.
- La complexité, ou le temps d'exécution, d'un programme `P` est le nombre d'opérations élémentaires nécessaire à l'exécution de `P`. Lorsqu'il est demandé de donner une certaine complexité, cette dernière sera donnée par une borne supérieure asymptotique (en $O(\dots)$) *et sera justifiée*.

Exercice 1. Dans cet exercice, on manipule des suites (finies) d'entiers sous forme de listes d'entiers. Ainsi la suite `(0, 1, 2, 3)` sera représentée par la liste `[0,1,2,3]`. La liste est croissante (respectivement décroissante, monotone) si la suite est croissante (respectivement décroissante, monotone).

1. Écrire une fonction `estCroissante` d'argument une liste d'entiers et qui renvoie un booléen indiquant si cette liste est croissante ou pas. Cette fonction devra être de complexité linéaire dans le pire des cas.
2. Ecrire de même des fonctions `estDecroissante` et `estMonotone` qui testent si une liste d'entiers est respectivement décroissante, monotone.

   Soit la liste `L` $= [u_0, u_1, \dots, u_{n-1}]$ de longueur n. On appelle tranche de `L` une liste de la forme $[u_i, u_{i+1}, ..., u_j]$ où $0 \leq i \leq j < n$. On cherche une tranche de `L` croissante et de longueur maximale. Par exemple, une tranche croissante de longueur maximale de `[0,1,0,1,2,3,4,0,1,2]` est `[0,1,2,3,4]`, correspondant aux indices $i = 2$ à $j = 6$.
3. Ecrire une fonction `LC` de deux arguments, une liste `L` et un entier p, qui renvoie l'entier d tel que : la liste $[u_p, \dots, u_{p+d}]$ est croissante et soit $p + d = n - 1$, soit $u_{p+d} > u_{p+d+1}$.
4. Ecrire une fonction `maxCroissante` d'argument une liste `L` qui renvoie la plus longue tranche croissante de `L`. S'il n'y a pas unicité, on renvoie la première trouvée.

Exercice 2.

Les parties sont dans une large mesure indépendantes.

Dans cet exercice, les graphes ont un ensemble de sommets de la forme $\{0, 1, \dots, n-1\}$ et sont orientés complets et pondérés par des entiers : pour tout couple (i, j) de sommets distincts, il existe un arc de i à j de poids $m_{i,j} \in \mathbb{Z}$. Le graphe est représenté par sa matrice d'adjacence $M = (m_{i,j})_{0 \leq i,j \leq n}$ avec la convention que $m_{i,i} = 0$ pour tout sommet i (il n'y a pas d'arc d'un sommet vers lui-même).

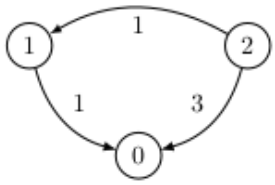
On considère une élection, à laquelle se présentent n candidats. Chaque électeur inscrit sur son bulletin lénsemble des candidats (tous les candidats sont classés), par ordre de préférence. Lénsemble des bulletins est rassemblé dans une urne. Le nombre de votants est noté p, l'urne contient donc p bulletins.
Par exemple, s'il y a trois candidats et qu'un électeur préfère le candidat 2, puis le candidat 0 et enfin considère que le candidat 1 est le moins souhaitable, son bulletin de vote sera `[2, 0, 1]`.
Une fois le vote effectué dans l'urne u, on compare les résultats de deux candidats particuliers i et j en comptant le nombre de bulletins qui classent le candidat i avant le candidat j. On exprime le résultat de la comparaison entre les candidats i et j en calculant la différence entre le nombre de bulletins de vote qui placent i avant j et le nombre de ceux qui placent j devant i.

Exemple 1 : On considère une élection avec trois candidats et trois votants : $n = 3$ et $p = 3$. L'urne u est composée des trois bulletins `[2, 0, 1]`, `[2, 1, 0]` et `[1, 2, 0]`. L'urne u est donc la liste

```
u1=[[2, 0, 1], [2, 1, 0], [1, 2, 0]]
```

La comparaison entre le candidat numéro 0 et le candidat numéro 1 donne -1 car le candidat est placé une fois avant le candidat 1 et deux fois après.

On peut alors synthétiser le contenu d'une urne u en construisant le graphe de préférence des votants: ses sommets correspondent aux candidats et l'arc du sommet i vers le sommet j est pondéré par la comparaison entre le candidat i et le candidat j.
La figure suivante donne le graphe de préférence obtenu à partir de l'exemple 1 ainsi que sa matrice d'adjacence M. Afin d'alléger le schéma, seuls les arcs avec un poids strictement positifs sont représentés.



$$M = \begin{pmatrix} 0 & -1 & -3 \\ 1 & 0 & -1 \\ 3 & 1 & 0 \end{pmatrix}.$$

Partie A Matrice d'adjacence

A1. Écrire une fonction `compte` prenant en argument un bulletin b et deux candidats i et j et retournant `True` si le candidat i est placé avant le candidat j dans le bulletin b, `False` sinon.

A2. Estimer la complexité de la fonction `compte` en fonction de n, le nombre de candidats.

A3. Écrire une fonction `duel` prenant en argument une urne u et deux candidats i et j et retournant la comparaison entre les candidats i et j pour tous les bulletins de l'urne u. Dans l'exemple 1, on avait par exemple

Python 3 Shell

```
>>> duel(u1, 0, 1)
-1
```

A4. Estimer la complexité de la fonction `duel` en fonction de n, le nombre de candidats, et de p, le nombre de bulletins.

A5. Exemple 2 : On considère une élection avec trois candidats et quatre votants : $n = 3$ et $p = 4$. À l'issue du vote, le contenu de l'urne u est

```
u2 = [[0, 1, 2], [1, 2, 0], [0, 2, 1], [0, 2, 1]]
```

Tracer le graphe de préférence de l'urne (en ne dessinant que les arcs ayant un poids strictement positif) et donner sa matrice d'adjacence.

A6. Expliquer pourquoi la matrice d'adjacence d'un graphe de préférence est antisymétrique et pourquoi tous ses coefficients non-diagonaux ont la même parité.

A7. Etant donné une urne u, on note Mat(u) la matrice du graphe de préférence associée à cette urne. Écrire une fonction `depouillement` prenant en argument le nombre de candidats n et une urne u et retournant la matrice Mat(u).

A8. Estimer la complexité de la fonction `depouillement` en fonction de n, le nombre de candidats, et de p, le nombre de bulletins.

Partie B Vainqueur de Condorcet

L'objectif de cette partie est de déterminer le vainqueur, ou les vainqueurs *ex aequo*, d'un vote par préférence. On appelle vainqueur de Condorcet tout sommet tel que, dans le graphe des préférence, les arcs sortant de ce sommet ont tous un poids positif ou nul. Ainsi, dans l'exemple 1, le candidat 2 est un vainqueur de Condorcet.

B1. Expliquer pourquoi un vainqueur de Condorcet peut être qualifié de "vainqueur" de l'élection.

B2. En se plaçant dans le cas $n = 3$ et $p = 3$, construire une urne pour laquelle il n'existe pas de vainqueur de Condorcet. Tracer le graphe de préférence correspondant.

B3. En se plaçant dans le cas $n = 3$ et $p = 4$, construire une urne pour laquelle il existe plusieurs vainqueurs de Condorcet. Tracer le graphe de préférence correspondant.

B4. Écrire une fonction `condorcet` prenant en argument la matrice d'adjacence d'un graphe de préférence et retournant la liste des vainqueurs de Condorcet.

Partie C Lecture des résultats des votes

Les résultats d'un vote sont stocké dans le fichier texte `urne2.txt` ci-dessous

```
3 4
0 1 2
1 2 0
0 2 1
0 2 1
```

- La première ligne définit deux entiers : le nombre de candidats et le nombre de votes.
- Chaque ligne, à partir de la deuxième, représente un bulletin.

C1. Écrire une fonction `recup(nomFichier)` qui prend pour argument une chaîne de caractère `nomFichier` donnant le nom du fichier texte contenant la description d'une urne, et renvoie l'urne correspondante.

Python 3 Shell

```
>>> recup("urne2.txt")
[[0, 1, 2], [1, 2, 0], [0, 2, 1], [0, 2, 1]]
```

On retrouve ici l'urne u2 de l'exemple 2.

Exercice 3.

Soit `A` un tableau de n éléments numérotés de 0 à $n-1$, `A[0]`, `A[1]`,..., `A[n-1]`. On suppose que l'on peut comparer ces éléments à l'aide de l'opérateur «==».

Pour $x \in$ `A`, on note E_x l'ensemble des positions où apparaît l'élément x dans `A` :

$$E_x = \{\ k \mid 0 \leq k \leq n-1 \text{ et } A[k] = x\ \}.$$

On désigne par c_x le cardinal de cet ensemble, c'est-à-dire le nombre d'occurrences d'un objet x dans le tableau `A`.

On dit qu'un élément $x \in$ `A` est **majoritaire** si

$$c_x = \text{card}\{\ k \mid 0 \leq k \leq n-1 \text{ et } A[k] = x\ \} > \frac{n}{2};$$

c'est-à-dire si l'élément x apparaît dans une proportion $> 50\%$ dans A. En particulier, si A possède un élément majoritaire, celui-ci est unique.

Partie A Un algorithme naïf

A1. On propose la fonction suivante, qui calcule c_x, le nombre d'occurrence de x dans la liste A.

```
def nb_occurences(A, x):
    n = len(A)
    cx = 0
    i = 0
    while i < n:
        if A[i] == x:
            cx = cx + 1
        i = i + 1
    return cx
```

Vérifier la correction partielle du segment itératif à l'aide de l'invariant

$$0 \leq \texttt{i} \leq \texttt{n} \text{ et } \texttt{cx} = \text{card}\{\ k \mid 0 \leq k \leq i-1 \text{ et } A[k] = x\ \}.$$

A2. Donner une fonction de terminaison (un variant) pour la boucle précédente.
En déduire la preuve totale de cette fonction.

A3. Montrer que la complexité en temps dans le pire cas de `nb_occurences` en fonction de la longueur `n` du tableau `A` est un $\Theta(n)$.

A4. Écrire une fonction `majoritaire_naif` ayant pour arguments une liste `A` et qui renvoie l'élément majoritaire de `A` s'il existe, et qui renvoie la valeur `None` sinon. Cette fonction testera chaque élément de la liste `A` pour savoir si il est majoritaire. On pourra faire appel à la fonction `nb_occurences`.

A5. Quelle est la complexité en temps dans le pire cas de la fonction `majoritaire_naif` en fonction de la longueur `n` du tableau `A` ?

Partie B Une approche diviser pour régner

Afin d'améliorer l'algorithme, on part de la remarque suivante : scindons le tableau `A` en deux sous tableaux `A1` et `A2` de longueur $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$.

Si x est l'élément majoritaire de `A`, alors x est majoritaire dans `A1` ou dans `A2`. En effet, si x n'est pas majoritaire ni dans `A1`, ni dans `A2`, on a

$$c_x \leq \frac{\lfloor n/2 \rfloor}{2} + \frac{\lceil n/2 \rceil}{2} \leq \frac{n}{2}.$$

B1. On donne le début d'une fonction récursive `majoritaire` qui utilise l'approche diviser pour régner afin de résoudre le problème ci-dessus.

```
1  def majoritaire(A):
2      print(A)
3      n = len(A)
4      if n == 1:
5          return A[0]
6      elif n > 1:
7          x1 = majoritaire(........
8          x2 = majoritaire(........
9  ....
10 ....
11 ....
12 ....
13 ....
14 ....
15 ....
16 ....
17 ....
```

Compléter le code de la fonction `majoritaire`. On pourra utiliser la fonction `nb_occurences` un nombre limité de fois.

Indication : Inutile de recopier les premières lignes sur votre copie. On commencera par la ligne 6 et l'on numérotera les lignes suivantes.

Dans la suite, on note $T(n)$ la complexité en temps dans le pire cas pour la fonction `majoritaire` pour les listes de longueur n. On pourra supposer que n est une puissance de 2 si nécessaire.

B2. Quelle relation de récurrence vérifie $T(n)$?

B3. Déduire des questions précédentes une borne asymptotiquement approchée de $T(n)$.

Indication : On pourra par exemple tracer l'arbre des appels récursifs associé à la fonction `majoritaire` et préciser la profondeur (ou hauteur) de cet arbre. On peut également faire une preuve analytique.

B4. Déterminer une borne asymptotiquement approchée de la complexité en espace dans le pire cas de la fonction `majoritaire` en fonction de la longueur `n` du tableau `A`.

Exercice 4. Les règles du golf sont simples: vous disposez d'une boule d'approximativement 5 cm de diamètre posée sur une boule d'environ 12 000 km de diamètre. Le but est de frapper la première sans toucher la deuxième.

On programme alors un robot pour qu'il joue parfaitement au golf. Quand il frappe la balle, il va directement vers le trou sur le green et la balle parcours toujours exactement la distance donnée pour le club utilisé. Chaque frappe de balle est considérée comme un coup et le but du jeu est d'atteindre le trou sur le green en un minimum de coup. Pour cela, notre robot dispose d'un nombre déterminé de clubs permettant chacun de parcourir une distance particulière.

Écrire la fonction `golf` permettant à notre robot de choisir la meilleure combinaison de clubs pour atteindre cet objectif. Celui-ci devra aussi être capable de déterminer s'il n'est pas possible de trouver une combinaison de coups permettant d'atteindre l'objectif.

- Données : un entier `distance` et une liste d'entiers `liste_clubs`. Le premier représente la distance à parcourir, le deuxième représente la distance que permet de couvrir chaque club à disposition.
- Résultat : la (une) liste des coups, la plus faible, permettant d'atteindre le trou sans le dépasser si c'est possible. La liste vide `[]` dans le cas contraire.

Python 3 Shell

```
>>> golf(100, [33, 44, 66, 22, 1])
[33, 66, 1]
>>> golf(100, [33, 44, 66, 22, 11])
[]
>>> golf(123, [33, 44, 66, 22, 1])
[33, 44, 44, 1, 1]
```

Votre programme devra être correctement commenté.

Indication : Une approche récursive est fortement recommandée.