Concours Blanc - Epreuve d'informatique

Consignes

- La calculatrice n'est **pas autorisée**.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Représentation et manipulation de l'*ADN

CAPES NSI L3

L'ADN est une molécule formée d'une chaîne de nucléotides, chacun contenant l'une des quatre bases azotées suivantes :

— l'adénine, notée `A` ;

— la thymine, notée `T` ;

— la cytosine, notée `C` ;

— la guanine, notée `G` ;

Ces petites molécules, appelées nucléotides, servent de « *lettres* » pour écrire le code génétique. Trois bases consécutives forment ce qu'on appelle une *codon*. Par exemple, `ATC` ou encore `GTT` sont des codons.

On définit en Python la variable `adn = "ATCGTACGT"`

Q1– Quel est le type de la variable `adn` ?

> `adn` est du type chaine de caractères `str`

Q2– Quel est le type et la valeur de la variable `longueur = len(adn)` ?

> `longueur` est du type `int` et vaut 9, c'est le nombre de caractère de la chaine `adn`.

Q3– Quel est le type et le contenu de la variable `morceau = adn[1:6]` ?

> `morceau` est de type `str` et contient `TCGTA` car elle commence en position 1 de la chaine `adn` et se termine en position `6` *exclu*.

Q4– Écrire une fonction `est_adn` qui prend en argument une chaîne de caractères et renvoie un booléen indiquant si cette chaîne de caractères représente bien un morceau d'ADN, c'est-à-dire si cette chaîne ne contient que les quatre nucléotides `A`, `T`, `C` et `G`. Par exemple,

— `est_adn("ATCGGTA")` renvoie `True` ;

— `est_adn("ATCBGTA")` renvoie `False` ;

— `est_adn("ATCgGTA")` renvoie `False` ;

```python
def est_adn(adn):
    for x in adn:
        if x!='A' and x!='C' and x!='G' and x!='T':
            return False
    return True
```

Q5– Écrire une fonction `occurrences` qui prend en argument une chaîne de caractères et renvoie un dictionnaire associant à chaque caractère représentant un nucléotide son nombre d'occurrences dans le brin d'ADN. On supposera que la chaîne de caractères fournie en argument est bien un brin d'ADN. Par exemple,

— `occurrences("ATCGTACGT")` renvoie le dictionnaire `{"A" : 2, "C" : 2, "G" : 2, "T" : 3 }`;
— `occurrences("TTTTGGTG")` renvoie le dictionnaire `{"A" : 0, "C" : 0, "G" : 3, "T" : 5 }`;

```python
def occurrences(adn):
    assert est_adn(adn), "Ce n'est pas un brin d'adn"
    occ = {"A":0, "C":0, "G":0, "T":0}
    for x in adn:
        occ[x]+=1
    return occ
```

On veut adopter une représentation plus compacte d'un brin d'ADN notamment dans le cas où un même nucléotide apparait plusieurs fois successivement, pour cela on code en base 2 sur deux bits les nucléotides :

Nucléotide	A	C	G	T
Code sur 2 bits	00	01	10	11

Un entier sur 8 bits représente alors un nucléotide répété un certain nombre de fois, les deux premiers bits de l'entier représentent le nucléotide et les 6 bits suivants indiquent le nombre de répétitions. Par exemple

— l'entier 10 s'écrit en binaire sur un octet $(00001010)_2$, les deux premiers bits sont 00 et donc représentent le nucléotide `A`, les 6 bits suivants donnent le nombre de répétitions $(001010)_2 = (10)_{10}$ et donc cet entier code `AAAAAAAAAA`.
— l'entier 167 s'écrit en binaire sur un octet $(10100111)_2$, comme 10 (les deux premiers bits) code le nucléotide `G` et que les 6 bits suivants représentent $(100111)_2 = (39)_{10}$ en décimale, cet entier représente le nucléotide `G` répété 39 fois.
— l'entier 200 s'écrit en binaire sur un octet $(11001000)_2$ et donc représente 8 répétitions du nucléotide `T`.

Q6– Que représente l'entier 101 ?

$(101)_{10} = (01100101)_2$ et donc cet entier représente le nucléotide C répété 37 fois.

Q7– Par quel entier peut-on représenter `TTTTTTTTTT` (10 répétitions de `T`) ?

On doit représenter le nucléotide `T` donc les premiers bits sont 11 puis on écrit 10 en base 2 sur les 6 bits restants ce qui donne $(11001010)_2 = (202)_{10}$.

Q8– Avec ce codage, quel est le nombre maximal de répétitions d'un nucléotide que l'ont peut coder avec un entier sur 8 bits ?

Le nombre de répétitions est codé sur 6 bits et donc le nombre maximal de répétitions est $(111111)_2 = (63)_{10}$.

Q9– Écrire la fonction `encode` qui prend en entrée un nucléotide et son nombre de répétitions et renvoie l'entier qui le représente. Par exemple, `encode('A',10)` renvoie 10, `encode('G',39)` renvoie 167, `encode('T',8)` renvoie 200.

```python
def encode(nucleotide, repetitions):
    code = repetitions
    if nucleotide=="C":
        code += 64
    elif nucleotide=="G":
        code += 128
    elif nucleotide=="T":
        code += 192
    return code
```

❑ **Exercice 2** : *Planification de rendez-vous optimale*

Une plateforme de prise de rendez-vous doit associer les requêtes de n clients à des créneaux disponibles. Chaque client demande un créneau horaire spécifique (représenté par un entier), et on dispose de m créneaux disponibles (avec $m \geqslant n$). L'objectif est d'assigner à chaque client un créneau disponible en minimisant l'*écart total*, défini comme la somme des écarts (en valeur absolue) entre le créneau demandé et le créneau assigné.

Exemple : Il y a 3 clients qui demandent les créneaux `[12, 9, 17]` et cinq créneaux sont disponibles : `[18, 4, 10, 14, 11]`. Une attribution possible est d'affecter le créneau `10` à la demande `12`, le créneau `11` à la demande `9` et le créneau `18` à la demande `17`, ce que l'on peut représenter en Python par la liste de tuples `[(12,10), (9, 11), (17, 18)]`. Cette attribution a un écart total de $|12-10|+|9-11|+|17-18| = 2+2+1 = 5$.

Q10– Dans l'exemple précédent, proposer une meilleure attribution, c'est-à-dire ayant un écart total strictement inférieur à 5.

> On peut faire l'attribution suivante : `[(12,11), (9, 10), (17,18)]` qui a un écart total de 3.

Q11– Écrire une fonction `ecart_total` qui prend en argument une liste de tuples de deux entiers représentant une attribution et renvoie son écart total. Par exemple `ecart_total([(7,10), (5,4), (9,9)])` doit renvoyer 4 ($|7-10|+|5-4|+|9-9|$).

```
1 def ecart_total(attributions):
2     ec = 0
3     for une_attribution in attributions:
4         demande, creneau = une_attribution
5         ec = ec + abs(demande-creneau)
6     return ec
```

Q12– On suppose, dans cette question uniquement, que $n = m$. Afin de rechercher l'attribution optimale (celle ayant l'écart total le plus faible), on propose l'algorithme qui consiste à tester toutes les attributions possibles et à renvoyer celle ayant le plus petit écart total. Quelle est la complexité de cette méthode en fonction de n ? Que peut-on en conclure ?

> On doit tester toutes les attributions possibles, c'est à dire toutes les permutations possibles de la liste des n créneaux. On doit donc tester $n!$ attributions, pour chacune de ces attributions la fonction précédente qui calcule l'écart total a une complexité en $\mathcal{O}(n)$ car la boucle est parcourue n fois et ne contient que des opérations élémentaires. Le complexité finale de cet algorithme est donc en $\mathcal{O}(n \times n!)$, il est donc inutilisable en pratique sauf sur de petites valeurs de n.

Q13– Un autre algorithme consiste à traiter la liste des demandes dans l'ordre et à attribuer au client le créneau disponible le plus proche de sa demande. Par exemple si la liste des demandes est `[7, 3, 10]` et que la liste des créneaux disponibles est `[4, 9, 11]` alors on effectue les attributions suivantes : `7` → `9`, puis `3` → `4` et enfin `10` → `11`. Dans quelle famille d'algorithmes peut se ranger ce procédé ? Justifier.

> C'est un algorithme glouton car on fait un choix optimal local (le meilleur créneau possible pour le client en cours) sans se soucier de l'impact de ce choix sur l'écart total.

Q14– Montrer, sur un exemple de votre choix, que cet algorithme ne fournit pas toujours la solution optimale au problème.

> On peut prendre les demandes `[9, 11]` avec les disponibilités `[10, 7, 16]` l'algorithme fournit alors l'attribution `[(9,10), (11, 7)]` pour un écart total de 5 alors que l'attribution `[(9,7),(11,10)]` a un écart total de 3.

Q15– Écrire une fonction `attribue` qui prend en argument un entier `cible` ainsi qu'une liste `disponibles` et qui renvoie un entier de `disponibles` le plus proche en valeur absolue de `cible`. Par exemple `attribue(12, [7, 11, 14, 9, 3])` renvoie `11`.

```python
def attribue(cible, disponibles):
    res = disponibles[0]
    for d in disponibles:
        if abs(cible-d)<abs(cible-res):
            res = d
    return res
```

Q16– Écrire une fonction `supprime` qui prend en argument un entier `n` ainsi qu'une liste d'entiers `lst` et renvoie la liste obtenue en supprimant la première occurrence de `n` dans `lst`. Par exemple `supprime(11, [7, 11, 14, 9, 3])` renvoie `[7, 14, 9, 3]`.

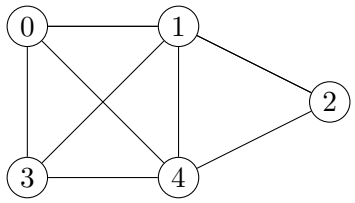
```python
def supprime(n, lst):
    res = []
    suppression = False
    for x in lst:
        if n==x and suppression==False:
            suppression=True
        else:
            res.append(x)
    return res
```

Q17– En utilisant les fonctions écrites aux deux questions précédentes, écrire une implémentation de l'algorithme d'attribution décrit à la question **Q13**. C'est-à-dire une fonction `attribution` qui prend en argument une liste d'entiers `demandes` et une liste de créneaux `créneaux` et renvoie l'attribution (sous la forme d'une liste de tuples) obtenue par la méthode décrite à la question **Q13**.

```python
def attribution(demandes, creneaux):
    attrib = []
    cr = creneaux.copy()
    for d in demandes:
        c = attribue(d, cr)
        cr = supprime(c, cr)
        attrib.append((d,c))
    return attrib
```

❑ **Exercice 3** : *Triangle dans un graphe*

On considère un graphe non orienté $G = (S, A)$ où $S = \{0, \dots, n-1\}$. On dit qu'un sous-ensemble de S à trois éléments $\{s, t, u\}$ est un *triangle* de G lorsque les arêtes $\{s, t\}$, $\{t, u\}$ et $\{s, u\}$ appartiennent à A. Par exemple, dans le graphe g suivant, $\{0, 1, 3\}$ est un triangle, par contre $\{0, 1, 2\}$ n'est pas un triangle car $\{0, 2\}$ n'est pas une arête.



Q18– Donner tous les triangles du graphe g.

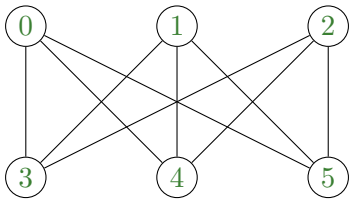
Les triangles du graphe g sont : `{0, 1, 3}`, `{0, 1, 4}`, `{0, 3, 4}`, `{1, 2, 4}` et `{1, 3, 4}`

Q19– Rappeler la définition d'un graphe complet. Donner, en le justifiant rapidement, le nombre de triangles d'un graphe complet à n sommets.

Un graphe complet est un graphe dans lequel pour toute paire de sommets (i, j), on a ij qui est un arc c'est à dire $ij \in A$. Donc tous les triplets de sommets sont des triangles, donc il y a autant de triplets de sommets que de triangle, c'est à dire $\binom{n}{3}$ triangles

Q20– On dit qu'un graphe est bipartite lorsqu'il existe une partition de l'ensemble des sommets S en deux ensembles S_1 et S_2 tel que toute arête ait un élément dans S_1 et l'autre dans S_2. Dessiner un graphe bipartite à 6 sommets et 9 arêtes.

Les sommets $0, 1$ et 2 sont dans S_1 et les sommets $3, 4$ et 5 dans S_2.



Q21– Montrer qu'un graphe bipartite ne contient pas de triangles.

Soit $ij \in A$, et quitte à échanger leurs rôles supposons $i \in S_1$ et $j \in S_2$, alors pour tout autre sommet k du graphe, soit $k \in S_1$ et donc $ik \notin A$, soit $k \in S_2$ et donc $kj \notin A$. Donc ijk n'est pas un triangle.

On suppose maintenant que les graphes sont représentés en Python par *listes d'adjacence* c'est à dire qu'un graphe est la donnée d'un dictionnaire dans lequel les clés sont les numéros de chacun des sommets. La valeur associée à la clé `s` est la liste des voisins du sommet `s`. Les graphes étant *non orientés*, pour tout sommet s, si t apparaît dans la liste d'adjacence de s alors s apparaît dans celle de t.

Q22– Donner le dictionnaire de Python qui représente le graphe g donné en exemple ci-dessus.

```
g = {
    0: [1, 3, 4],
    1: [0, 2, 3, 4],
    2: [1, 4],
    3: [0, 1, 4],
    4: [0, 1, 2, 3]
}
```

Q23– Écrire la fonction `est_arete` qui prend en argument un graphe (représenté par un dictionnaire) ainsi que deux sommets `s` et `t` et renvoie un booléen indiquant si `{s,t}` est une arête de ce graphe.

```
def est_arete(gr, s, t):
    for x in gr[s]:
        if x==t:
            return True
    return False
```

Afin de lister les triangles d'un graphe on propose l'algorithme suivant : pour chaque arête `{s,t}` du graphe, on recherche l'intersection des listes d'adjacence de `s` et de `t`.

Q24– Écrire une fonction `intersection` qui prend en argument deux listes d'entiers *supposées triées* et renvoie leur intersection. On souhaite une fonction de complexité linéaire par rapport à la taille de ces listes.

```
def intersection(lst1, lst2):
    i1 = 0
    i2 = 0
    inter =  []
    while i1<len(lst1) and i2<len(lst2):
        if lst1[i1]==lst2[i2]:
            inter.append(lst1[i1])
            i1 += 1
            i2 += 1
        elif lst1[i1]<lst2[i2]:
            i1 +=1
        else:
            i2+=1
    return inter
```

Q25– En déduire une implémentation de l'algorithme de recherche des triangles d'un graphe sous la forme d'une fonction `triangles` qui prend en argument un graphe `g` dont on suppose les listes d'adjacence triées et renvoie la liste des triangles de ce graphe.

```
def triangles(gr):
    tr = []
    for s in gr:
        for t in gr[s]:
            if t>s:
                for x in intersection(gr[s], gr[t]):
                    if x>t:
                        tr.append((s,t,x))
    return tr
```