ÉPREUVE SPÉCIFIQUE — PCSI

INFORMATIQUE

Durée : 2 heures

L'usage de calculatrices est interdit. — Aucun document n'est autorisé.

La présentation, la lisibilité, l'orthographe, la qualité de la rédaction, la clarté et la précision des raisonnements entreront pour une part importante dans l'appréciation des copies. En particulier, les candidats devront apporter les commentaires suffisants à la compréhension de leurs programmes et veilleront à utiliser des noms de variables explicites. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

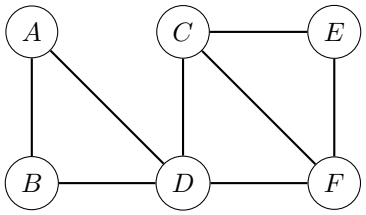
Le sujet est composé de quatre exercices indépendants.

***Gestion du temps** — En admettant une durée d'une 5 minutes pour la lecture et l'assimilation du sujet, il est vivement conseillé de consacrer environ 15 minutes sur l'exercice 1, 40 minutes sur l'exercice 2, 30 minutes sur l'exercice 3 et 30 minutes sur l'exercice 4.*

Vous rédigerez sur 2 copies séparées les exercices {1, 2} et {3, 4}.

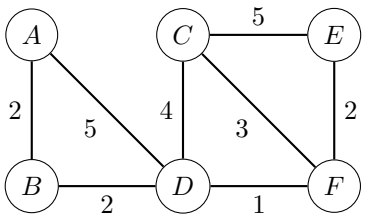
Exercice 1 – Parcours en profondeur et algorithme de Dijkstra

On s'intéresse au graphe suivant, implémenté avec des listes d'adjacence triées par ordre alphabétique.



Question 1.1. *Appliquer l'algorithme de parcours en profondeur à ce graphe, pour le sommet de départ A.*
La réponse prendra la forme d'un tableau à deux colonnes, avec une ligne pour chaque étape de l'algorithme. La première colonne contiendra l'état de la file ou pile (précisez laquelle il faut utiliser ici), la deuxième colonne contiendra la liste des sommets déjà marqués par l'algorithme (inutile de recopier le graphe).

Pour la suite de l'exercice, on pondère les arêtes du graphe :



Question 1.2. *Appliquer l'algorithme de Dijkstra à ce graphe pondéré, pour le sommet de départ A.*
La réponse prendra la forme d'un tableau dans lequel on détaillera les valeurs des étiquettes, avec une ligne pour chaque étape de l'algorithme. On indiquera le nom du sommet de provenance en cas de modification de l'étiquette.

Question 1.3. *À l'aide du tableau construit à la question précédente, déterminer :*

— *un plus court chemin de A à C.*

— *un plus court chemin de A à E.*

Exercice 2 – Correction et complexité des algorithmes

Un professeur examine un immeuble à n étages (numérotés de 1 à n) et désire déterminer à partir de quel étage une calculatrice chutant au sol est cassée. Il dispose pour cela de k calculatrices. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : jeter la calculatrice par la fenêtre. Si elle n'est pas cassée, on peut la réutiliser ensuite, sinon on ne peut plus. (On suppose qu'il existe un étage à partir duquel toutes les chutes sont fatales, et que toutes les chutes depuis un étage inférieur ne le sont pas.)

On cherche un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (l'algorithme renverra $n+1$ si un saut de l'étage numéro n n'est pas fatal) en faisant le minimum d'essais.

On rappelle que pour tout nombre réel x, $\lfloor x \rfloor$ désigne la partie entière de x.

On notera $S(n)$ le nombre d'essais à effectuer dans le pire des cas pour déterminer l'étage fatal.

Dans la première partie du problème, on suppose qu'on ne dispose que d'une calculatrice à sacrifier, c'est-à-dire que $k = 1$. On propose alors l'algorithme `Etage1` suivant.

```
Etage1(n)
    i prend la valeur 0
    Tant que i<=n et la calculatrice n'est pas cassée
        i prend la valeur i+1
        Jeter la calculatrice par la fenêtre de l'étage i
    Retourner i
```

Question 2.1. *Donner une précondition et une post-condition pour l'algorithme.*

Question 2.2. *Montrer que l'algorithme termine en utilisant un variant de boucle.*

Question 2.3. *Donner un invariant de boucle et montrer la correction de l'algorithme.*

Question 2.4. *Donner une estimation asymptotique de $S(n)$ pour cet algorithme.*

On suppose maintenant que $k = 2$, c'est-à-dire que l'on peut sacrifier deux calculatrices. On utilise alors un algorithme fondé sur l'idée suivante : découper l'immeuble en blocs de taille $\lfloor \sqrt{n} \rfloor$, puis lancer la calculatrice par la fenêtre en montant d'un bloc à chaque fois.

Si cette première calculatrice est cassée, repartir du début de l'avant-dernier bloc et lancer la calculatrice par la fenêtre d'étage en étage.

Cet algorithme nommé `Etage2` peut s'écrire plus formellement de la manière suivante :

```
Etage2(n)
    b prend la valeur partie entière de racine de n
    i prend la valeur 0
    Tant que i<=n et la première calculatrice n'est pas cassée
        i prend la valeur i+b
        Jeter la première calculatrice par la fenêtre de l'étage i
    Si i<=n
        j prend la valeur i-b
        Tant que la seconde calculatrice n'est pas cassée
               j prend la valeur j+1
               Jeter la seconde calculatrice par la fenêtre de l'étage j
        Retourner j
    Sinon
        Retourner n+1
```

Question 2.5. *Donner une estimation asymptotique de $S(n)$ pour cet algorithme.*

On suppose maintenant que l'on dispose d'un nombre infini de calculatrices à sacrifier, et on décide de procéder par dichotomie.

On commence par faire lancer une calculatrice de l'étage n. Si elle n'est pas cassée, on retourne $n+1$. Sinon, l'étage fatal se situe entre 1 et n.

On procède alors de façon récursive à l'aide de l'algorithme `Etage3` suivant.

```
Etage3(m,n)
   Si m=n
       Retourner m
   Sinon
       p prend la valeur partie entière de (m+n)/2
       Jeter une calculatrice par la fenêtre de l'étage p
       Si la calculatrice n'est pas cassée
           Retourner Etage3(p+1,n)
       Sinon
           Retourner Etage3(m,p)
```

Question 2.6. *Justifier rapidement que cet algorithme termine.*

On se place dans le cas où l'étage fatal est compris entre 1 et n et on désire donner une estimation asymptotique du nombre $S(n)$ d'essais effectués.

Question 2.7. *On se place pour cela pour commencer dans le cas où il existe un entier p tel que $n = 2^p$. Expliquer pourquoi $S(n) = S(\frac{n}{2}) + 1$. En déduire l'expression de $S(n)$.*

Question 2.8. *En déduire une estimation asymptotique de $S(n)$ dans le cas général.*

Exercice 3 – Permutations et tri par dénombrement

Dans cet exercice, on ne considérera que des listes contenant des entiers naturels. Toutes les fonctions pourront être implémentées en partant de ce principe.

Si $n \in \mathbb{N}$, on appelle *progression d'ordre n* la liste $[0, 1, ..., n-1]$. La progression d'ordre 0 est la liste vide. Plus généralement, on dit qu'une liste est une progression s'il s'agit d'une liste de ce type.
Exemple : $L = [0, 1, 2, 3, 4]$ est une progression (plus exactement, la progression d'ordre 5).

Question 3.1. *Écrire une fonction* `progression` *qui prend en argument une liste* `L` *et dont le rôle est de reconnaître une progression. La fonction renverra* `True` *si* `L` *est une progression,* `False` *sinon.*

Si $n \in \mathbb{N}$, on dit qu'une liste `L` est une *permutation d'ordre n* si elle peut être obtenue en permutant les éléments de la n-progression. Par exemple, les permutations d'ordre 3 sont les listes :

$$[0,1,2], \quad [0,2,1], \quad [1,0,2], \quad [1,2,0], \quad [2,0,1] \quad \text{et} \quad [2,1,0]$$

Par convention, l'unique permutation d'ordre 0 est la liste vide. On dit qu'une liste `L` est une permutation s'il existe un entier naturel $n \in \mathbb{N}$ tel que `L` soit une permutation d'ordre n.

Question 3.2. *Implémenter une fonction* `appartient` *qui prend en argument une liste* `L` *et un entier* `p` *et qui renvoie* `True` *si* `p` *appartient à la liste* `L`, `False` *sinon.*

Une façon simple de savoir si une liste `L` de longueur n est une permutation d'ordre n est de vérifier successivement que chacun des nombres de $[\![0, n-1]\!]$ appartient bien à la liste `L`.

Question 3.3. *Écrire une fonction* `permutation` *qui prend en argument une liste* `L` *et qui renvoie* `True` *si* `L` *est une permutation,* `False` *sinon, en utilisant ce principe.*

Question 3.4. *Estimer la complexité de cet algorithme dans le pire des cas.*

On cherche à optimiser l'algorithme précédent. Pour cela, on initialise une liste `presence` de la façon suivante :

```
presence = [False for i in range(len(L))]
```

Question 3.5. *Implémenter une fonction* `permutationR` *qui prend en argument une liste* `L` *de longueur n et qui renvoie* `True` *si* `L` *est une permutation,* `False` *sinon, de façon à ce que sa complexité, dans le pire des cas, soit désormais en $\mathcal{O}(n)$.*

La plupart des algorithmes de tris s'appuient sur la comparaison de deux valeurs du tableau. Ces **tris par comparaison** ne font aucune hypothèse sur les valeurs à trier. Mais si on dispose d'informations supplémentaires, par exemple une plage de valeurs possibles, on peut les utiliser, par exemple en comptant dans un tableau *histogramme* les occurrences de chaque valeur du tableau à trier. Un seul parcours suffit et il est très facile de reconstituer le tableau dans l'ordre croissant à partir de l'histogramme, et on peut même le faire en place.

Question 3.6. *Compléter la fonction* `tri_comptage(tab, binf, bsup)` *donnée ci-dessous avec sa spécification.*

```
def tri_comptage(tab, binf, bsup):
    """
    Tri en place par comptage le tableau
    tab tel que pour tout 0 <= k < len(tab)
    on a  binf <= tab[k] <= bsup
    Parameters
    ----------
    tab : tableau d'entiers
    binf, bsup : int et int
    Returns
    -------
    None.
    """
    histo = [0 for _ in range(bsup - binf + 1)]
    # on remplit l'histogramme
    # ...(à compléter)...
    # on reconstitue le tableau
    # ...(à compléter)...
```

Question 3.7. *En utilisant la fonction précédente, comment obtenir une progression d'ordre n à partir d'une permutation d'ordre n ?*

Exercice 4 – Contrôle de parité croisé

Le contrôle de parité croisé est un code correcteur, c'est-à-dire une technique de codage de l'information basée sur la redondance, qui vise à détecter et éventuellement corriger une erreur de transmission d'un message transmis sous la forme d'une suite de bits. Par exemple, la probabilité d'erreur (un zéro devient 1 et inversement) sur une ligne téléphonique est de l'ordre de 10^{-6}. Avec un tel taux d'erreur et une connexion à 1 Mo/s, en moyenne 8 bits erronés sont transmis chaque seconde.

Dans ce qui suit, on appellera **mot binaire** un tableau unidimensionnel d'entiers 0 ou 1 représentant des bits, par exemple `[0,1,0,1,1,1,0]` où les $(b_i)_{i\in[\![0,6]\!]}$ se lisent $[b_6, b_5, b_4, b_3, b_2, b_1, b_0]$. Pour illustrer le principe du contrôle de parité croisé, considérons un train d'informations de 42 bits regroupés en 6 mots binaires $(M_j)_{j\in[\![0,5]\!]}$ de 7 bits dans un tableau `T` bidimensionnel (liste de listes). Le contrôle de parité croisé est basé sur une représentation des données sous forme d'un bloc de 7 lignes et autant de colonnes que de mots binaires de 7 bits, ici 6 sur la représentation ci-dessous.

```
>>> T
[[0, 1, 1, 1, 0, 0],
 [1, 1, 1, 1, 1, 1],
 [1, 0, 0, 0, 1, 1],
 [1, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1],
 [1, 1, 0, 1, 0, 1],
 [0, 1, 1, 1, 1, 1]]
```

	M_0	M_1	M_2	M_3	M_4	M_5
b_0	0	1	1	1	0	0
b_1	1	1	1	1	1	1
b_2	1	0	0	0	1	1
b_3	1	1	0	0	0	0
b_4	0	0	0	0	1	1
b_5	1	1	0	1	0	1
b_6	0	1	1	1	1	1

Le principe du contrôle de parité croisé est d'étendre à ce bloc de données le contrôle de parité « simple ». On rappelle que le code de parité « simple » consiste à ajouter à une séquence de 7 bits à transmettre un bit supplémentaire dit de parité qui vaut 0 si la somme des 7 bits est paire et 1 si elle impaire. Le contrôle de parité croisé consiste à effectuer :

— le contrôle longitudinal de redondance ou LRC (*Longitudinal Redundancy Check*), c'est-à-dire calculer le bit de parité pour chacune des 7 lignes ;

— le contrôle vertical de redondance ou VRC (*Vertical Redundancy Check*), c'est-à-dire calculer le bit de parité de chacune des colonnes, y compris celle de LRC.

Le bloc initial complété avec ces deux mots de contrôle est donné ci-dessous.

	M_0	M_1	M_2	M_3	M_4	M_5	LRC
b_0	0	1	1	1	0	0	1
b_1	1	1	1	1	1	1	0
b_2	1	0	0	0	1	1	1
b_3	1	1	0	0	0	0	0
b_4	0	0	0	0	1	1	0
b_5	1	1	0	1	0	1	0
b_6	0	1	1	1	1	1	1
VRC	**0**	**1**	**1**	**0**	**0**	**1**	**1**

Enfin, on précise qu'une façon astucieuse de calculer le bit de parité est de faire un « ou exclusif » (l'un ou l'autre mais pas les deux) entre tous les bits du mot.

À toutes fins utiles, on initialise le code suivant :

```
T= [[0,1,1,1,0,0],[1,1,1,1,1,1],[1,0,0,0,1,1],[1,1,0,0,0,0],\
    [0,0,0,0,1,1],[1,1,0,1,0,1],[0,1,1,1,1,1]]
```

Question 4.1. *Définir de façon récursive, puis proposer une implémentation d'une fonction* `Parite` *qui prend comme argument un tableau* `M` *correspondant à un mot binaire et qui renvoie l'entier 0 ou 1 associé au bit de parité.*

Question 4.2. *Définir une fonction* `LRC` *qui prend comme argument un tableau* `T` *et qui renvoie un nouveau tableau contenant en plus le mot binaire associée au LRC.*
ℹ *On pourra faire appel à la fonction* `deepcopy` *du module* `copy`.

Appliquée au tableau `T`, cette fonction doit renvoyer :

```
>>> LRC(T)
[[0, 1, 1, 1, 0, 0, 1],
 [1, 1, 1, 1, 1, 1, 0],
 [1, 0, 0, 0, 1, 1, 1],
 [1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1, 0],
 [1, 1, 0, 1, 0, 1, 0],
 [0, 1, 1, 1, 1, 1, 1]]
```

Question 4.3. *Et tenant compte de la symétrie des rôles entre l'action sur les lignes de la fonction* `LRC` *et celle sur les colonnes de la fonction* `VRC`*, proposer une implémentation de la fonction* `VRC` *exploitant la fonction* `LRC` *qui prend comme argument un tableau* `T` *(déjà codé avec le LRC) et qui renvoie un nouveau tableau contenant en plus le mot binaire associé au VRC.*
ℹ *On pourra définir une fonction* `transpose` *qui renvoie la transposée d'un tableau.*

Appliquée au tableau `T`, la composition de cette fonction avec la précédente doit renvoyer :

```
>>> VRC(LRC(T))
[[0, 1, 1, 1, 0, 0, 1],
 [1, 1, 1, 1, 1, 1, 0],
 [1, 0, 0, 0, 1, 1, 1],
 [1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1, 0],
 [1, 1, 0, 1, 0, 1, 0],
 [0, 1, 1, 1, 1, 1, 1],
 [0, 1, 1, 0, 0, 1, 1]]
```

Question 4.4. *Définir une fonction* `extraire` *qui prend comme argument un tableau* `T` *codé avec contrôle de parité croisé et qui renvoie le tableau initial sans les bits de contrôle.*

La vérification du bloc est plus robuste aux erreurs que le code de parité simple puisqu'il assure un double contrôle horizontal et vertical. En cas d'apparition d'une (et une seule) erreur, il est possible de la corriger puisqu'elle est localisable dans le message reçu.

Question 4.5. *Expliquer précisément comment détecter s'il n'y a eu aucune erreur, exactement une erreur ou plus d'une erreur lors de la transmission. En déduire une succession d'étapes à réaliser pour implémenter une fonction permettant d'extraire le tableau et notamment pour corriger l'erreur dans le cas où elle est unique.*

Question 4.6. *Définir une fonction* `decode` *qui prend comme argument un tableau* `T` *codé avec contrôle de parité croisé et qui renvoie le tableau extrait s'il n'y a pas d'erreur, le tableau extrait avec le bit corrigé s'il y avait une seule et unique erreur et* `False` *s'il y a au moins deux erreurs.*

— Fin de l'énoncé —