ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE MPI

INFORMATIQUE

Durée : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. 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.

RAPPEL DES CONSIGNES

- *Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.*
- *Ne pas utiliser de correcteur.*
- *Écrire le mot FIN à la fin de votre composition.*

Les calculatrices sont interdites.

Le sujet est composé de trois parties, toutes indépendantes.

Partie I - Palindromes

En langue française, "ressasser" est le mot palindrome le plus long, tandis qu'il semble que "saippuakauppias" soit le plus long mot palindrome au monde, désignant un marchand de savon en Finlande. L'objet de cette partie est de compter le nombre de palindromes présents dans un mot donné.

Soit Σ un alphabet fini contenant au moins deux lettres. On note $u = u_0 \cdots u_{n-1}$ un *mot* sur Σ, composé de n lettres $u_i \in \Sigma, i \in [\![0, n-1]\!]$. La longueur de u est notée $|u|$. Pour $0 \leq i < j \leq n$, on note $u[i, j]$ le mot $u_i \cdots u_{j-1}$. L'ensemble des mots construits sur Σ et contenant le mot vide ϵ est noté Σ^*.

Définition 1 (Miroir)
Soit $u \in \Sigma^*$. Le *miroir* de $u = u_0 \cdots u_{n-1}$, noté $\bar{u}$, est le mot $\bar{u} = u_{n-1} \cdots u_0$. Par convention $\bar{\epsilon} = \epsilon$.

Définition 2 (Palindrome)
$u \in \Sigma^*$ est un *palindrome* si et seulement si $u = \bar{u}$.

Par convention, le mot vide ϵ n'est pas considéré comme un palindrome.

On dira qu'un palindrome u est *pair* (respectivement *impair*) lorsque sa longueur $|u|$ est paire (resp. impaire).

On recherche donc dans cette partie le nombre de palindromes facteurs d'un mot $u \in \Sigma^*$ (comptés avec les multiplicités éventuelles), soit le cardinal de l'ensemble $\left\{(i, j), 0 \leq i < j \leq |u|,\ u[i, j] = \overline{u[i, j]}\right\}$. Dit autrement, on recherche le nombre de palindromes contenus dans u.

Q1. Si $\Sigma = \{a, b\}$, donner le nombre de palindromes contenus dans le mot $u = babb$.

En parcourant naïvement les lettres d'un mot u donné, on peut proposer un algorithme en pseudo-code permettant de compter le nombre palindromes de u (**algorithme 1**).

Algorithme 1 - Décompte naïf du nombre de palindromes contenus dans un mot donné.

Entrées : Un mot u.
Sorties : Le nombre de palindromes contenus dans u.

```
début
    nb = 0
    pour i=0 à |u|-1 faire
        pour j=0 à |u|-1 faire
            estPalindrome=True
            pour k=i à j − 1 faire
                si u[i] ≠ u[j − k − 1] alors
                    estPalindrome=False
                    Sortir du pour k
            si estPalindrome alors
                nb = nb+1
    retourner nb
```

Q2. Évaluer la complexité au pire des cas de l'**algorithme 1** en fonction de $|u|$.

On souhaite bien sûr améliorer cette première idée. Pour ce faire, on utilise tout d'abord le paradigme de la programmation dynamique.

Pour $u \in \Sigma^*$, on définit un tableau de booléens P de taille $(|u| + 1) \times (|u| + 1)$, $P[i][j]$ étant vrai si $u[i, j]$ est un palindrome. On a donc pour tout $i \in [\![0, |u| - 1]\!]$, $P[i][i + 1] = True$.

Q3. Soit $u[i, j]$ un mot. À quelles conditions sur u_i, u_{j-1} et $u[i + 1, j - 1]$ le mot $u[i, j]$ est-il un palindrome ?

Q4. En déduire une relation de récurrence vérifiée par les coefficients de P.

Q5. Écrire un algorithme de programmation dynamique en pseudo-code résolvant le problème. Évaluer sa complexité.

On insère maintenant entre chaque paire de lettres de u, ainsi qu'au début et à la fin du mot, un symbole spécial noté #. On appelle ce nouveau mot $u^{\#}$. Ainsi, le mot $u = abba$ se transforme en $u^{\#} = \#a\#b\#b\#a\#$.

Q6. Montrer que les palindromes de $u^{\#}$, s'ils existent, sont tous impairs.

Q7. Soit v un palindrome de longueur $2k+1$ de u, $k \in \mathbb{N}$. On construit le mot $u^{\#}$. Donner les deux palindromes correspondants à v dans $u^{\#}$.

Q8. Même question si v est un palindrome de longueur $2k$ de u.

Q9. En déduire une stratégie de recherche de tous les palindromes de u.

On voit que les palindromes impairs sont importants. On va donc construire un algorithme de recherche de ce type de palindrome.

Définition 3 (Rayon d'un palindrome)
Soient $u \in \Sigma^*$ et $i \in [\![0, |u|-1]\!]$. On dit qu'il existe un *palindrome centré en i de rayon $\rho > 0$* si :

(i). $i - \rho \geq 0$,

(ii). $i + \rho + 1 \leq |u|$,

(iii). $u[i-\rho, i+\rho+1]$ est un palindrome.

Le *rayon maximal* $\hat{\rho}_i$ d'un palindrome centré en i est le plus grand rayon d'un palindrome centré en i.

Par exemple, si $u = abbababab$ et $i = 4$, alors b est un palindrome centré en 4 de rayon 0, aba est un palindrome centré en 4 de rayon 1, $babab$ est un palindrome centré en 4 de rayon 2 et c'est le plus grand, donc $\hat{\rho}_4 = 2$.

Q10. En remarquant qu'un palindrome impair est centré sur une lettre (ou un symbole spécial dans le cas de $u^{\#}$), proposer un algorithme en pseudo-code permettant de compter le nombre de palindromes impairs d'un mot u. Évaluer sa complexité.

Q11. Soient $u \in \Sigma^*$, $i \in [\![0, |u|-1]\!]$ et $0 < j \in [\![1, \hat{\rho}_i]\!]$. En exploitant les différentes symétries, montrer qu'il existe un palindrome centré en $i+j$ de rayon $min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$. En déduire $\hat{\rho}_{i+j} \geq min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$. Préciser à quelle condition il y a égalité.

En utilisant cette remarque, on développe un algorithme, dit de Manacher, qui construit un tableau d'entiers T permettant de compter le nombre de palindromes d'un mot u. Plus précisément, pour chaque position $i \in [\![1, |u|-1]\!]$, $T[i]$ indique le rayon maximal $\hat{\rho}_i$, donc tel que la sous-chaîne de $i - \rho_i$ à $i - \rho_i + 1$ est un palindrome. L'algorithme 2 consiste à incrémenter $T[i]$ jusqu'à trouver le plus grand palindrome $u[i - T[i], i + T[i+] + 1]$ centré en i.

Algorithme 2 - Algorithme de Manacher

Entrées : Un mot u
Sorties : Un tableau T.
début
  Initialiser un tableau T de $|u|$ cases, initialisées à 0
  k=0
  pour $i = 1$ *à* $|u| - 1$ **faire**
    $j = i - k$
    si $T[k] \geq j$ **alors**
      $T[i] = min(T[k-j], T[k] - j)$
    tant que $(i - (T[i]+1)) \geq 0$ **ET** $(i + T[i] + 1) < |u|$ **ET** $(u[i - (T[i]+1)] = u[i + T[i] + 1])$ **faire**
      $T[i] = T[i] + 1$
      $k = i$
  retourner T

Q12. Comment trouver à partir de cet algorithme le nombre de palindromes de u ?

Q13. Quelle est la complexité de cet algorithme ? Justifier.

Partie II - Traversée de rivière

Cette partie comporte des questions nécessitant un **code OCaml**.

Dans une vallée des Alpes, un passage à gué fait de cailloux permet de traverser la rivière. Deux groupes de randonneurs arrivent simultanément sur les berges gauche et droite de cette rivière et veulent la traverser. Le chemin étant très étroit, une seule personne peut se trouver sur chaque caillou de ce chemin (**figure 1**). Un randonneur sur la berge de gauche peut avancer d'un caillou (vers la droite sur la **figure 1**) et sauter par dessus le randonneur devant lui (un caillou à droite) si le caillou où il atterit est libre. De même, chaque randonneur de la berge de droite peut avancer d'un caillou (vers la gauche sur la **figure 1**) et sauter par dessus le randonneur devant lui, dans la mesure où le caillou sur lequel il atterit est libre. Une fois engagés, les randonneurs ne peuvent pas faire marche arrière. De plus, pour simplifier, on suppose qu'une fois tous les randonneurs sur le chemin, il ne reste qu'un caillou de libre.

Figure 1 - Les randonneurs et le chemin de cailloux

Le chemin de cailloux est défini par un tableau d'entiers :

```
type chemin_caillou = int array
```

Dans ce tableau, un randonneur venant de la berge de gauche est représenté par un 1, un randonneur issu de la berge de droite par un 2 et un caillou libre par un 0.

Q14. Écrire une fonction de signature `caillou_vide : chemin_caillou -> int` qui détermine la position du caillou inoccupé.

Q15. Écrire une fonction de signature `echange : chemin_caillou -> int -> int -> chemin_caillou` qui permute les valeurs codées sur deux cailloux. Le tableau d'entiers initial représentant le chemin n'est pas modifié. On pourra utiliser ici la fonction `copy` du module `Array`.

Q16. Écrire une fonction de signature `randonneurG_avance : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse avancer (vers la droite).

Q17. Écrire une fonction de signature `randonneurG_saute : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse sauter (vers la droite) au-dessus d'un randonneur.

On supposera dans la suite les fonctions de signature `randonneurD_avance : chemin_caillou -> bool` et `randonneurD_saute : chemin_caillou -> bool` écrites de manière similaire pour les randonneurs venant de la berge de droite.

Q18. Écrire une fonction de signature `mouvement_chemin : chemin_caillou -> chemin_caillou list` qui, en fonction de l'état du chemin, calcule l'état suivant après les opérations suivantes (si elles sont permises) :

(i). déplacement d'un randonneur venant de la berge de gauche,

(ii). déplacement d'un randonneur venant de la berge de droite,

(iii). saut d'un randonneur venant de la berge de gauche,

(iv). saut d'un randonneur venant de la berge de droite.

On donne la syntaxe OCaml pour créer une liste de N entiers i : `List.init N (fun x->i)`.
Par exemple, `List.init 5 (fun x->2);;` renvoie [2 ; 2 ; 2 ; 2 ; 2].

Q19. Écrire une fonction de signature `passage : int -> int`, utilisant la question précédente, telle que l'appel `passage nG nD` résout le problème de passage de `nG` randonneurs venant de la berge de gauche et `nD` randonneurs venant de la berge de droite. Par exemple, `passage 3 2` permet de passer de [1 ;1 ;1 ;0 ;2 ;2] à [2 ;2 ;0 ;1 ;1 ;1].

Partie III - Calcul d'une coupe minimum d'un graphe

Un agent secret a pour mission de perturber le réseau de communications ennemi en coupant certains fils dans le réseau. Ayant peu de moyens, l'agent a pour consigne de couper le moins de fils possibles pour accomplir cette tâche. Le réseau de communications est modélisé à l'aide d'un *multigraphe* non orienté $G = (S, A)$, où S est l'ensemble des sommets du graphe et A l'ensemble de ses arêtes. Résoudre le problème de l'agent secret, c'est rechercher une *coupe minimum* dans G.

Définition 4 (Multigraphe)
Un multigraphe est un graphe dans lequel des couples de mêmes sommets peuvent être reliés par plus d'une arête.

Dans toute la suite, on considérera les multigraphes sans arêtes du type (s, s) (boucles).

Définition 5 (Coupe)
Une coupe d'un multigraphe non orienté $G = (S, A)$ est une partition non triviale (S_1, S_2) de S, c'est-à-dire telle que $S_1 \cup S_2 = S, S_1 \cap S_2 = \emptyset$ et $S_1, S_2 \neq \emptyset$. On dira que les arêtes reliant S_1 à S_2 sont les arêtes de la coupe.

La *taille* d'une coupe (S_1, S_2) est définie par $|(S_1, S_2)| = |\{(s, t) \in A, s \in S_1, t \in S_2\}|$. Autrement dit, c'est le nombre d'arêtes de G qui relient S_1 et S_2.

Définition 6 (Coupe minimum)
Une coupe minimum est une coupe de taille minimale.

Pour trouver une coupe minimum d'un multigraphe $G = (S, A)$, on pourrait énumérer l'ensemble des coupes possibles (il y en a $2^{|S|}$...), ou encore utiliser un algorithme de flot (le plus efficace étant en $O(|S|^3)$). Nous proposons dans la suite un algorithme probabiliste, l'algorithme de Karger, basé sur la notion de contraction d'arête.

III.1 - Contraction d'arête

Soient $G = (S, A)$ un multigraphe et $a = (s, t) \in A$ une arête de G de sommets s et t. *Contracter* l'arête a consiste à :

(i). créer un nouveau sommet $u = (st)$ dans S. u est un *supersommet* du nouveau graphe,

(ii). pour toute arête (r, s) ou (r, t), $r \in S$, ajouter une arête (r, u) à A. Dans le cas où (r, s) et (r, t) existent, deux arêtes reliant r à u sont créées.

(iii). Supprimer de A toutes les arêtes ayant s ou t comme extrémité.

(iv). Supprimer s et t de S.

Un supersommet peut donc être vu comme un ensemble de deux sommets du graphe initial, obtenu après une contraction d'arête.
Le multigraphe obtenu est appelé *graphe contracté* et est noté G/st. Si plusieurs arêtes relient s à t dans G, contracter l'une d'entre elles supprime toutes les autres du graphe G/st. Dans toute la suite, on notera G_0 le graphe initial, avant transformations par contractions.
On considère le graphe G_0 de la **figure 2**.

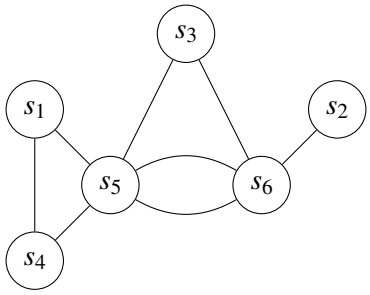

Figure 2 - Graphe G_0 exemple

Q20. Donner le graphe obtenu par contraction d'une arête (s_5, s_6).

Après plusieurs contractions, un supersommet u du graphe résultant contient un sous-ensemble $S_u \subseteq S$ de sommets de G.

Q21. Soit u un supersommet et $s, t \in S_u$. Montrer qu'il existe un chemin Γ entre s et t dans G_0 tel que chaque arête de Γ a été contractée.

Q22. Montrer qu'en contractant une arête s, t dans un multigraphe G, la taille d'une coupe minimum du graphe contracté G/st est au moins égale à la taille d'une coupe minimum de G.

Q23. Montrer que la taille d'une coupe minimum de G/st est strictement plus grande que celle d'une coupe minimum de G si et seulement si l'arête (s, t) est une arête de toutes les coupes minimum de G.

III.2 - Premier algorithme

On construit un premier algorithme qui essaye de trouver une coupe minimum en contractant aléatoirement une arête de G, jusqu'à ce qu'il ne reste que deux sommets (**algorithme 3**).

Algorithme 3 - Algorithme de Karger.

Karger(G, n**)**
Entrées : $G = (S, A)$ multigraphe non orienté, $n \in \mathbb{N}$
Sorties : Une coupe minimum de G.
début
  pour $i = |S|$ *à* n *en décrémentant de 1* **faire**
    Tirer aléatoirement (loi uniforme) une arête $a = (s, t) \in A$
    $G = G/st$

```
// Appel
```
Karger(G,2)

Q24. Appliquer l'**algorithme 3** au graphe de la **figure 2**, en contractant successivement (s_5, s_6), $((s_5 s_6), s_2)$, (s_1, s_4) et $((s_5 s_6 s_2), s_3)$. Chaque graphe contracté sera représenté. La coupe calculée est-elle une coupe minimum ?

D'après la **Q23**, tant que l'on ne contracte pas une arête faisant partie de toutes les coupes minimum de G, alors l'**algorithme 3** trouvera une coupe minimum. On cherche alors la probabilité de ne jamais contracter une telle arête.

Q25. Soit $d(s)$ le degré d'un sommet $s \in S$. Montrer que $\sum_{s \in S} d(s) = 2|A|$ (lemme des poignées de main).

Q26. En déduire qu'une coupe minimum a une taille d'au plus $2|A|/|S|$.

Soit $C = (S_1, S_2)$ une coupe minimum de G. L'objet des questions **Q27** et **Q28** est de minorer la probabilité que l'**algorithme 3** renvoie C. Pour cela, on remarque que cet algorithme ne renvoie pas C si et seulement si lors d'une itération on choisit une arête qui traverse C, c'est-à-dire telle qu'un sommet est dans S_1 et l'autre dans S_2.

Q27. Quelle est la probabilité maximum de choisir une arête qui traverse C ?

Q28. En déduire que la probabilité $P_{|S|}$ que l'**algorithme 3** renvoie C est supérieure ou égale à $\frac{2}{|S|(|S|-1)}$.

Q29. Déduire de la question précédente qu'un multigraphe non orienté $G = (S, A)$ possède au plus $\frac{|S|(|S|-1)}{2}$ coupes minimum.

III.3 - Deuxième algorithme

Le résultat précédent n'est pas satisfaisant d'un point de vue complexité. On peut cependant l'améliorer facilement en utilisant une technique d'*amplification* : on itère l'**algorithme 3** plusieurs fois et on retourne la valeur minimale obtenue (**algorithme 4**).

Algorithme 4 - Algorithme de Karger amplifié.

```
KargerAmplifie(G, N)
Entrées : G = (S, A) multigraphe non orienté, N ∈ ℕ.
Sorties : Une coupe minimum de G.
début
    t = ∞
    pour i = 1 à N faire
        X = Karger(G,2)
        si |X| < t alors
            t = |X|
            C = X
    retourner C
```

Q30. Montrer que la probabilité maximale que l'**algorithme 4** ne trouve pas une coupe minimum est égale à $\left(1 - \frac{2}{|S|(|S|-1)}\right)^N$.

Q31. On rappelle que pour tout $x \in \mathbb{R}$, $1 + x \leq e^x$. Montrer que la probabilité que l'**algorithme 4** renvoie une coupe minimum est supérieure à $1 - e^{2N/|S|(|S|-1)}$. En déduire, pour $c > 0$ fixé, la plus petite valeur de N telle que cette probabilité soit supérieure à $1 - \frac{1}{N^c}$.

Q32. Les algorithmes proposés dans cette partie sont des algorithmes probabilistes. Sont-ils de type Monte Carlo ou Las Vegas ? Justifier la réponse.

III.4 - Implémentation en langage C

Cette partie comporte des questions de programmation qui seront abordées en utilisant **exclusivement le langage C**. Les codes seront commentés de manière pertinente.

On suppose qu'un graphe est codé à l'aide de l'ensemble de ses arêtes, chaque arête étant définie par ses deux sommets, représentés par des entiers.

Q33. Définir en C des types structurés `Arete` et `Graphe`. Pour ce dernier, on s'attachera à avoir un accès direct au nombre de sommets et au nombre d'arêtes. On rappelle la définition d'un type structuré par `struct nom_s {type1 champ1;...;typen champn;}` et ensuite `typedef struct nom_s nom;`.

Pour tout u supersommet, les S_u forment une partition de S. Pour coder ces sous-ensembles, on a donc naturellement recours à la structure de données Unir & Trouver (Union-Find), qui va permettre de trouver rapidement à quel sous-ensemble S_u un sommet $s \in S$ appartient (Trouver) et de fusionner deux supersommets S_u et S_v déjà construits (Unir).
On suppose donc disposer :

(i). d'un type `subset`, décrivant un supersommet d'un graphe contracté.

```
struct subset
{
    int parent; // recherche par compression de chemin pour Trouver
    int rang;   // Union par rang.
};
typedef struct subset subset;
```

(ii). d'une fonction de prototype `int Trouver(subset subsets[], int s)`, qui retourne le numéro du supersommet dans lequel le sommet s se trouve (par compression de chemin),

(iii). d'une fonction de prototype `void Unir(subset subsets[], int Su, int Sv)` qui fusionne les deux supersommets S_u et S_v (union par rang) et maintient à jour la liste des supersommets.

Il n'est pas demandé d'écrire ces deux dernières fonctions.

Q34. Écrire une fonction de prototype `int contracteArete(Graphe G,subset subsets[],Arete a)` qui contracte l'arête `a`. On veillera à ne pas contracter l'arête si ses deux extrémités sont dans le même supersommet. La fonction renvoie 0 si aucune arête est contractée et -1 sinon.

Q35. Écrire une fonction de prototype `int compteAretesCoupe(Graphe G,subset subsets[])` qui compte le nombre d'arêtes du graphe contracté final G résultant de l'**algorithme 3**.
`subsets` est la partition des sommets du graphe initial S calculée par l'**algorithme 3**.

Q36. En déduire une fonction de prototype `int kargerMinCut(Graphe G0)` qui renvoie la taille de la coupe calculée par l'**algorithme 3** sur le graphe G_0. Les supersommets initiaux sont les $|S|$ sommets de G_0, chacun ayant un rang nul et un numéro parent égal au numéro du sommet. Pour effectuer un tirage aléatoire uniforme, on utilisera la fonction `int rand();` de la librairie `stdlib.h` qui renvoie un nombre aléatoire entre 0 et `RAND_MAX`≥ 32767. `RAND_MAX` est une constante définie dans `stdlib.h`.

Q37. Donner la complexité au pire des cas de cet algorithme.

FIN