Corrigé épreuve informatique concours CCINP MPI 2023

Partie 1 : comptage de palindromes contenus dans un mot donné

1. Soit $u = babb$. Les palindromes contenus dans u sont les mots suivants : b (trois fois), a (une fois), bb et bab chacun une fois, soit **6 sous-palindromes** en comptant les multiplicités éventuelles.

Remarque : on dénote deux fautes dans la ligne du **si ... alors** de l'algorithme 1, il faut remplacer le test par $u[k] \neq u[j-1+i-k]$.

2. L'algorithme 1 est constitué soit de boucle **for** (sur i en $|u|$, sur j en $|u|$, et sur k en $j-i$), soit de lignes qui sont en temps constants (des affectations, des tests, des calculs arithmétiques). Sa complexité temporelle peut donc être bornée (dans le pire des cas, où aucun des "Sortir du pour k" n'est appelé) par $\mathcal{O}(1) + \sum_{i=0}^{|u|-1} \sum_{j=0}^{|u|-1} \sum_{k=i}^{j-1} \mathcal{O}(1)$. La somme se borne par $\mathcal{O}(|u|^3)$, et donc la complexité au pire cas de l'algorithme 1 est $\mathcal{O}(|u|^3)$ (cubique).

3. On fixe $0 \leq i < j \leq |u| - 1$ et $u[i,j]$ un sous-mot de u. $u[i,j] = u_i \dots u_{j-1}$ est un palindrome ssi $u_i u_{i+1} \dots u_{j-2} u_{j-1} = u_{j-1} u_{j-2} \dots u_{i+1} u_i$ ssi $u_i = u_{j-1}$ et $u_{i+1} \dots u_{j-2} = u_{j-2} \dots u_{i+1}$ ssi $u_i = u_{j-1}$ et $u[i+1, j-1]$ est un palindrome.

4. La relation de récurrence que l'on déduit sur les coefficients de P est donc $P[i][j] = \text{True}$ ssi $u_i = u_{j-1}$ et $P[i+1][j-1]$ est vrai, pour l'hérédité, et $P[i][i+1] = \text{True}$ pour $0 \leq i \leq |u|-1$ pour l'initialisation.

5. On cherche à remplir les cases de P jusqu'à pouvoir le nombre de cases telles que $P[i,j] = \text{True}$, qui donne le nombre de palindromes inclus dans u.

```
Pseudocode de cet algorithme de programmation dynamique.
Entree : un mot u de taille n.
Sortie : nombre de palindromes inclus dans u.
Data : matrice P de (n+1)*(n+1) booleens, initialisees a Faux.

Pour i = 0 a n-1 Faire
    P[i][i+1] = Vrai
Fin Pour
Pour i = n-1 a 0 (decroissant) Faire :
    Pour j = 0 a n-1 Faire :
        P[i][j] = (u[i] == u[j-1]) ET P[i+1][j-1]
    Fin Pour
Fin Pour

nombre_palindromes = 0
Pour i = 0 a n-1 Faire :
    Pour j = 0 a n-1 Faire :
        Si P[i][j] Alors :
            nombre_palindromes += 1
    Fin Pour
Fin Pour

Renvoyer nombre_palindromes
```

Cet algorithme de programmation dynamique utilise une mémoire quadratique en $\mathcal{O}(|u|^2)$, et prend ce même temps pour l'initialiser puis pour la remplir, et enfin ce même temp pour la parcourir et calculer le nombre de palindromes inclus dans u.

6. Un sous-mot de $u^\#$ de longueur paire va nécessairement commencer par un # mais ne pas finir par un #, ou l'inverse. Autrement dit, il ne peut pas être un palindrome. Donc les palindromes de $u^\#$, s'ils existent, sont nécessairement tous de longueurs impaires.

7. Avec les notations de cette question, ces deux palindromes correspondant à v dans $u^\#$ seront $v^\#$ et $v'^\#$ qui est $v^\#$ auquel à été retiré # une fois à gauche et une fois à droite, respectivement de longueur $(2k+1)+(2k+2)$ et $(2k+1)+2k$. On vérifie que ces deux longueurs sont bien impaires, comme le donnait la question 6.

8. Les deux palindromes correspondant à v seront exactement pareil, $v^\#$ et $v'^\#$, respectivement de longueur $2k+(2k+1)$ et $2k+2k-1$, qui sont encore bien impaires.

9. On en déduit donc la stratégie suivante de recherche de tous les palindromes de u : on construit $u^\#$ (en temps linéaire en $|u|$), puis on recherche tous les palindromes de $u^\#$, par la stratégie la plus efficace possible (idéalement aussi en temps linéaire en $|u|$), et on en déduit les palindromes de u en effaçant les #.

   Remarque : Si on s'intéresse aussi au nombre de palindromes, ce qui bizarrement n'était pas demandé ici, alors on a le raisonnement suivant : chaque palindrome de u donne exactement deux palindromes de $u^\#$, et donc le nombre $n^\#$ de palindromes de $u^\#$ que l'on va trouver sera exactement le double du nombre de palindromes de u : pour obtenir le nombre n recherché, il suffira de renvoyer $n^\#/2$.

10. On peut, pour chaque position i candidate pour être un centre de sous palindrome, compter le nombre de palindromes centrés en i, et ensuite faire la somme de ces nombres pour obtenir le nombre total de sous palindromes du mot u. Pour évaluer le nombre de palindromes centrés en i, naïvement on peut proposer la méthode suivante : on part de cette lettre centrale, qui donne donc un palindrome (de rayon $\rho = 0$), et on augmente jusqu'à ne plus pouvoir (et donc jusqu'à trouver le rayon maximal $\hat{\rho}_i$). Cette méthode naïve est en temps quadratique $\mathcal{O}(|u|^2)$.

Remarque : pourquoi préciser "ou un symbole spécial #" ? Le symbole # est une lettre. Ou alors, il faudrait préciser $\# \notin \Sigma$.

11. *Remarque* : dans l'énoncé de la question 11, on trouve étrange la notation $0 < j \in [\![1, \hat{\rho}_i]\!]$, le terme $0 < j$ étant inutile.

    On se centre en i, et on considère le plus long palindrome centré en i, qui par définition est donc de rayon $\hat{\rho}_i$. Comme $1 \leq j \leq \hat{\rho}_i$, par définition du palindrome centré en i, on sait que le sous mot gauche $u[i-\hat{\rho}_i, i+1]$ est le miroir du sous mot droite $u[i, i+\hat{\rho}_i+1]$. À gauche, on peut considérer la position $i-j$, et à droite la position $i+j$. À gauche, on peut trouver un palindrome de longueur égale à $\hat{\rho}_{i-j}$ (par définition des $\hat{\rho}$), et donc des palindromes plus petits ($\star$). On peut donc trouver à gauche un palindrome centré en $i-j$ de longueur égale à $\min(\hat{\rho}_{i-j}, \hat{\rho}_i - j)$ (car $j \leq \hat{\rho}_i$). Par symétrie, ce palindrome donne à droite un palindrome miroir centré en $i+j$, de même rayon $\min(\hat{\rho}_{i-j}, \hat{\rho}_i - j)$.

    Comme on a exhibé un palindrome centré en $i+j$ de rayon égal à ce $\min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$, le rayon maximal $\hat{\rho}_{i+j}$ est plus grand ou égal à ce minimum, donc on a bien $\hat{\rho}_{i+j} \geq \min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$.

    Le cas d'égalité correspond à savoir si dans l'étape ($\star$) du raisonnement ci-dessus, le plus grand palindrome possible avait été trouvé. C'est-à-dire quand les deux rayons dans le calcul du minimum sont égaux : il y a égalité $\hat{\rho}_{i+j} = \min(\hat{\rho}_i - j, \hat{\rho}_{i-j})$ ssi $\hat{\rho}_i - j = \hat{\rho}_{i-j}$.

Remarque : On note une typo, dans la dernière ligne avant l'algorithme 2, il s'agit de trouver le plus grand palindrome $u[i - T[i], i + T[i+1] + 1]$ centré en i (il manque un 1 dans $T[i+1]$).

12. Pour une position i donnée, $T[i]$ après l'algorithme de Manacher contient $\hat{\rho}_i$ le rayon maximal. Comme illustré sur l'exemple de $u = abbababab$ donné en début, pour $i = 4$ on avait $\hat{\rho}_i = 2$ et $\hat{\rho}_i + 1 = 3$ sous palindromes correspondant centré en i (b, aba, et $babab$). Ainsi, pour trouver le nombre total de sous

palindromes de u, il suffit de faire la somme des valeurs $1 + T[i]$, pour chaque cases i, ce qui se fait en temps linéaire.

13. La complexité mémoire de l'algorithme 2 est en $\mathcal{O}(|u|)$. Pour la complexité temporelle dans le pire cas, chaque exécution de la boucle **tant que** augmente la variable k de un en un, par ailleurs cette variable ne peut pas dépasser n et elle n'est jamais modifiée en dehors de la boucle **tant que**. Donc au total la boucle **tant que** interne sera exécutée au pire n fois en tout ; ces n exécutions étant réparties entre les n exécutions de la boucle **pour**. Ce qui fait bien une complexité linéaire en $\mathcal{O}(n)$ pour l'algorithme 2.

Partie 2 : traversée d'une rivière par des randonneurs

Le sujet donne ce type, et nous pouvons commencer par donner un exemple de tel chemin, avec `nG` $= 3$ randonneurs à gauche, et `nD` $= 2$ randonneurs à droite, et un caillou vide entre eux.

```
type chemin_caillou = int array

let ex = [|1; 1; 1; 0; 2; 2|]
```

14. Voici la fonction `caillou_vide`, qui calcule la position du dernier caillou libre (ou vide, ou inoccupé). Comme il est supposé qu'il n'y a qu'un seul caillou libre, cela suffit. Notons que l'on pourrait utiliser des exceptions paramétriques pour interrompre le calcul plus tôt, ou une fonction récursive, mais cette version devrait suffire amplement.

```
let caillou_vide chemin =
    let position = ref (-1) in
    let n = Array.length chemin in
    for i = 0 to n-1 do
        if chemin.(i) = 0 then
            position := i
    done;
    !position
```

15. On fera attention à ne pas modifier le tableau d'entrée, mais à utiliser un `Array.copy` :

```
let echange_en_place tab i j =
    let tmp = tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- tmp

let echange chemin i j =
    let chemin2 = Array.copy chemin in
    echange_en_place chemin2 i j
```

16.

```
let randonneurG_avance chemin =
    let i = caillou_vide chemin in
```

```
    i >= 1 && chemin.(i-1) = 1
```

17.

```
let randonneurG_saute chemin =
    let i = caillou_vide chemin in
    i >= 2 && chemin.(i-2) = 1
```

18.

```
let mouvement_chemin chemin =
    let mouvements = ref [] in
    let caillou = caillou_vide chemin in
    if randonneurG_avance chemin then
        mouvements := (echange chemin caillou (caillou-1)) :: !↩
           mouvements;
    if randonneurG_saute chemin then
        mouvements := (echange chemin caillou (caillou-2)) :: !↩
           mouvements;
    if randonneurD_avance chemin then
        mouvements := (echange chemin caillou (caillou+1)) :: !↩
           mouvements;
    if randonneurD_saute chemin then
        mouvements := (echange chemin caillou (caillou+2)) :: !↩
           mouvements;
    !mouvements
```

Remarque : on va recalculer à chaque fois la position du caillou vide, ce qui se fait en temps linéaire. Il aurait donc été opportun d'inclure la position du caillou vide directement dans la configuration, pour y avoir systématiquement accès en temps constant.

19. On va générer le chemin initial, puis explorer depuis cette position initiale, avec la fonction précédente.

    La signature de la fonction `passage` attendue n'était évidemment pas `int -> int`, mais on propose la modification suivante : `passage : int -> int -> chemin_caillou list`, qui renverra la liste des configurations permettant de passer de l'état initial (représenté par `nG` et `nD` deux entiers) à l'état final.

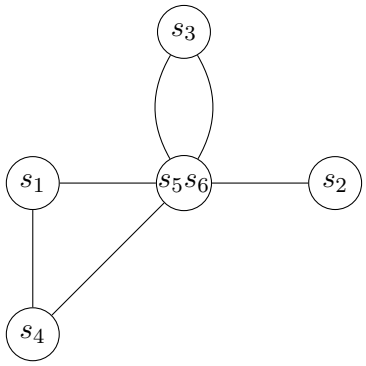
```
let init n1 v1 n2 v2 =
    let chemin = Array.make (n1 + 1 + n2) v1 in
    chemin.(n1) <- 0;
    for i = n1 + 1 to n1 + n2 do chemin.(i) <- v2 done;
    chemin

let passage nG nD =
    let c       = init nG 1 nD 2 in
    let c_final = init nD 2 nG 1 in
    let rec cherche c suite =
        match suite with
        | [] -> if c = c_final then [[c]] else []
```

```
        | c' :: suite ->
            let chemins = cherche c' (mouvement_chemin c') in
            List.map (List.cons c) chemins @ cherche c suite
    in
    cherche c (mouvement_chemin c)
```

On note que le sujet donne la syntaxe OCaml pour `List.init` (qui était attendu pour construire l'état initial du jeu), sauf que l'état du jeu est codé par un tableau (`type chemin_caillou = int array`). Cela peut être source supplémentaire de confusion pour certains candidats.

Partie 3 : algorithme(s) de Karger pour la recherche probabiliste d'une coupe minimum d'un graphe

20. On obtient :



21. On montre par récurrence sur $n \geq 2$ la propriété $H(n)$ suivante : "Si u est un supersommet tel que $|S_u| = n$ alors pour tous $s, t \in S_u$ il existe un chemin dans G_0 entre s et t dont toutes les arêtes ont été contractées pour produire u". L'initialisation découle de la définition d'une contraction et l'hérédité est immédiate. Le résultat demandé s'ensuit.

22. Si $C = (S_1, S_2)$ est une coupe minimum dans G/st, sans perte de généralité le sommet $st \in S_1$ et alors $((S_1 \setminus \{st\}) \cup \{s, t\}, S_2)$ est une coupe dans G qui est :

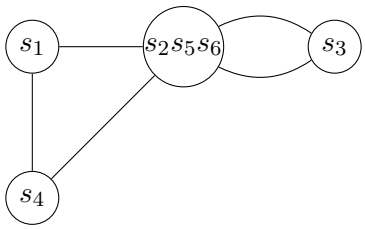
    - De même taille que C.
    - Par définition de taille plus grande (large) que la plus petite des tailles d'une coupe dans G.

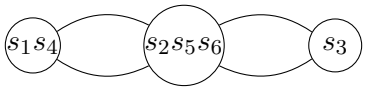
    Donc une coupe minimum dans G/st est de taille supérieure à celle d'une coupe minimum dans G.

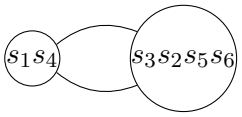
23. On note N_{st} la taille d'une coupe minimum dans G/st et N celle d'une coupe minimum dans G.

    - Si $N_{st} > N$, soit (S_1, S_2) une coupe minimum de G et supposons par l'absurde que (s, t) n'en soit pas une arête. Alors (S_1', S_2) où S_1' est égal aux sommets de S_1 auxquels on a enlevé s et t et ajouté st est une coupe de G/st de même cardinal que (S_1, S_2) et (par définition) de cardinal supérieur à N_{st}. Donc $N_{st} \leq N$ ce qui n'est pas.
    - Réciproquement, si (s, t) est une arête de toutes les coupes minimum de G, alors toute coupe où s et t sont dans le même sous-ensemble est de taille strictement plus grande, et comme il y a une bijection (qui conserve le cardinal des coupes) entre les coupes de G où s et t sont dans le même sous-ensemble, et les coupes de G/st, cela donne le résultat attendu : $N_{st} > N$.

24. La première étape de l'algorithme produit le graphe obtenu à la question 1. On obtient ensuite :







On obtient la coupe $(\{1,4\},\{2,3,5,6\})$ de taille 2 qui n'est pas minimale puisque $(\{1,3,4,5,6\},\{2\})$ est de taille 1.

Remarque : Dans l'algorithme 3, G est modifié au fur et à mesure donc a priori A et S aussi. Pour A, c'est tant mieux car c'est ce qu'on veut. Pour S c'est problématique car $|S|$ est la borne de la boucle pour.

25. On dénombre de deux manières différentes le nombre n d'extrémités d'arêtes dans le graphe G. Chaque arête ayant deux extrémités, on a d'une part $n = 2|A|$. D'autre part, le nombre d'extrémités d'arêtes total est égal à la somme sur tous les sommets s des nombres d'extrémités d'arêtes touchant s donc $n = \sum_{s\in S} d(s)$.

26. Notons k la taille d'une coupe minimum de G. Alors, chaque sommet de G a un degré au moins égal à k sinon il existe un sommet $u \in G$ tel que $d(u) < k$ et dans ce cas $(\{u\}, S \setminus \{u\})$ est une coupe de taille strictement inférieure à k ce qui est une contradiction. De ceci et de la question 25 on déduit :

$$2|A| = \sum_{s\in S} d(s) \geq \sum_{s\in S} k = |S|k$$

et donc que $k \leq 2|A|/|S|$ comme attendu (si le graphe est non vide, évidemment).

Remarque : Pour cette question, il faut que le graphe soit non vide pour avoir $|S| \neq 0$. Cette hypothèse aurait pu être indiquée dès le début de la partie 3.

27. *Remarque : Cette question me semble être très mal posée car "la" probabilité de choisir une arête qui traverse C est mal définie : la probabilité de choisir une arête de C à la première contraction ou à la deuxième contraction n'est pas la même... Je corrige cette question en "Quelle est la probabilité maximum de choisir une arête qui traverse C lors de la première contraction ?"*

La probabilité de choisir une arête de C à la première contraction vaut

$$\frac{\text{nombre d'arêtes de } C}{\text{nombre d'arêtes de } G} \leq \frac{2|A|}{|S|} \times \frac{1}{|A|} = \frac{2}{|S|}$$

d'après la question 26 et le fait que C est une coupe minimum.

28. On note $n = |S|$, supposé $n \geq 2$. Notons par ailleurs E_i l'événement "à la i-ème itération de l'algorithme de Karger, l'arête choisie pour être contractée ne fait pas partie de C" et E l'événement "l'algorithme de Karger renvoie C". L'énoncé nous informe que $E = \bigcap_{i=1}^{n-2} E_i$ puisque l'algorithme renvoie C si et seulement si on ne choisit jamais une arête de C pour être contractée lors des $n-2$ contractions. D'après la formule des probabilités composées :

$$\mathbb{P}(E) = \mathbb{P}(E_1) \times \mathbb{P}(E_2|E_1) \times \cdots \times \mathbb{P}\left(E_{n-2} \middle| \bigcap_{i=1}^{n-3} E_i\right)$$

D'après la question 27, on a $\mathbb{P}(E_1) \geq 1 - 2/n$. Une fois que E_1 s'est produit, une coupe minimale dans le graphe contracté (qui contient maintenant $n-1$ sommets) est toujours de même taille que C

et donc en appliquant le même raisonnement qu'à la question précédente, $\mathbb{P}(E_2|E_1) \geq 1 - 2/(n-1)$ et ce même raisonnement s'étend de proche en proche montrant que pour tout $i \in [\![2, n-2]\!]$,

$$\mathbb{P}\left(E_i \middle| \bigcap_{j=1}^{i-1} E_j\right) \geq 1 - \frac{2}{n-i+1}$$

On en déduit que

$$P_{|S|} = \mathbb{P}(E) \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\frac{n-i-1}{n-i+1} = \frac{(n-2)(n-3)\times\cdots\times 2\times 1}{n(n-1)(n-2)\times\cdots\times 3} = \frac{2}{n(n-1)}$$

Remarque : Je ne sais pas répondre proprement à cette question autrement que par le raisonnement suivant. Il me semble parfaitement irréaliste qu'un candidat l'ait trouvé. Donc, soit c'était autre chose qui était attendu mais je ne vois pas quoi, soit cette question est beaucoup trop difficile.

29. Notons $C_1, C_2, \ldots, C_r$ les coupes minimales du graphe G. Par l'absurde, si $r > |S|(|S|-1)/2$, la probabilité P que l'algorithme de Karger renvoie une coupe minimum serait telle que

$$P = \sum_{i=1}^{r} \mathbb{P}(\text{l'algorithme renvoie } C_i) \geq \sum_{i=1}^{r} \frac{2}{|S|(|S|-1)} > 1$$

l'avant dernière inégalité provenant de la question 28. C'est impossible et par conséquent le nombre de coupes minimum de $G = (S, A)$ est inférieur ou égal à $|S|(|S|-1)/2$.

Remarque : Je ne suis pas d'accord avec la phrase "Le résultat précédent n'est pas satisfaisant d'un point de vue complexité", c'est au contraire très satisfaisant. Ce qui ne l'est pas c'est que la probabilité de tomber sur une coupe minimum est vraiment très petite. D'où l'intérêt de l'amplifier, ce qui est justement l'objectif de la partie suivante.

30. D'après la question 28, la probabilité de ne pas trouver une coupe minimum en une itération de l'algorithme 4 est inférieure à $1 - 2/|S|(|S|-1)$. Les N tentatives faites dans cet algorithme étant indépendantes, on obtient le résultat.

31. Notons P la probabilité que l'algorithme 4 renvoie une coupe minimum. D'après la question 30,

$$P \geq 1 - \left(1 - \frac{2}{|S|(|S|-1)}\right)^N$$

L'indication de l'énoncé implique que pour tout $x \in \mathbb{R}$, $-(1+x)^N \geq -e^{Nx}$ et en appliquant cette inégalité avec $x = -2/|S|(|S|-1)$ on obtient $P \geq 1 - e^{-2N/(|S|-1)|S|}$.

Pour répondre à la suite de la question il s'agit de trouver le plus petit N tel que

$$\frac{2N}{|S|(|S|-1)} \geq c \log N$$

Remarques :

- Il manque un "moins" dans l'exponentielle de la formule proposée par l'énoncé. De toutes façons avec un plus, on serait en train de minorer une probabilité par une quantité négative quand N est grand ce qui n'a pas grand intérêt.
- Pour la deuxième partie de la question, une analyse de la fonction $N \mapsto N/\log N$ étudiée sur $[2, +\infty[$ montre qu'elle est strictement croissante et divergente au dela de e donc que le N recherché existe, mais en donner une forme explicite paraît inenvisageable.

32. Ces deux algorithmes sont de type Monte Carlo : ils fournissent un résultat correct avec une certaine probabilité mais en un temps indépendant des choix aléatoires effectués.

33. On propose de représenter une paire de sommets (donc une arête le graphe étant non orienté) par :

```
struct Arete {
    int sommet1;
    int sommet2;
};
typedef struct Arete Arete;
```

L'énoncé semble suggérer de représenter un graphe comme étant une struct à trois champs, l'un indiquant le nombre de sommets, le deuxième le nombre d'arêtes et le troisième correspondant à un tableau contenant les arêtes. D'où :

```
struct Graphe {
    int nb_sommets;
    int nb_aretes;
    Arete* tab_aretes;
};
typedef struct Graphe Graphe;
```

Remarque : Il manque un ; final dans le rappel de la syntaxe permettant de définir un type dans l'énoncé.

34. La seule difficulté est de comprendre que la donnée du graphe initial et de la partition actuelle de ses sommets permet de manipuler indirectement le graphe contracté sans avoir à le construire explicitement :

```
int contracteArete(Graphe G, subset subsets[], Arete a)
{
    int s = a.sommet1;
    int t = a.sommet2;
    if (Trouver(subsets, s) == Trouver(subsets, t))
    {
        return 0;
    }
    else
    {
        Unir(subsets,s,t);
        return -1;
    }
}
```

Remarques : Cette question me semble très obscure et trompeuse :

- *L'énoncé semble impliquer que l'on représente un graphe contracté par la donnée du graphe initial et de la partition actuelle des sommets. C'est raisonnable puisque ces deux informations permettent de reconstruire à la volée le graphe contracté mais pas du tout clair dans l'énoncé.*
- *Avec cette façon de faire, il est tout à fait inutile de faire du graphe G un argument de con-*

tracteArete. La présence de cet argument incitera à coup sûr les candidats à vouloir modifier le graphe donné en argument dans la fonction contracteArete ce qui est impossible puisqu'il faudrait un pointeur dessus pour le faire et est de toutes façons une gageure vu le type Graphe choisi.

- *L'entier renvoyé par contracteArete est incongru, on s'attendrait à renvoyer* 0 *en cas de comportement "normal", c'est-à-dire dans le cas où la contraction s'est passée sans encombre. L'énoncé impose au contraire de renvoyer* 0 *en cas d'erreur, c'est étrange.*

35. En réinterprétant les entrées de cette fonction comme étant le graphe initial avant les contractions et la partition des sommets de ce graphe si on avait fait toutes les contractions, il suffit pour chaque arête de regarder si ses deux extrémités sont dans le même morceau de la partition ou non :

```
int compteAretesCoupe(Graphe G, subset subsets[])
{
    int taille_coupe = 0;
    int n = G.nb_aretes;
    for (int i = 0; i < n; i++)
    {
        Arete a = G.tab_aretes[i];
        if (Trouver(subsets, a.sommet1) != Trouver(subsets, a.↩
           sommet2))
        {
            taille_coupe++;
        }
    }
    return taille_coupe;
}
```

Remarque : Là encore l'énoncé est très confus et la principale difficulté est de comprendre ce qui est attendu. Le graphe G en entrée n'est pas le graphe contracté (sans quoi il suffirait de renvoyer G.nb_aretes pour répondre à la question sans se soucier de l'argument subsets !)

36. La seule difficulté est de remarquer que lorsqu'on tire aléatoirement une arête à contracter dans le graphe G on peut retomber sur une arête qui a déjà été contractée puisqu'on ne modifie jamais G. Il ne suffit donc pas de faire $n-2$ contractions il faut faire $n-2$ contractions "réussies". Ça tombe bien, la question 34 permet de vérifier si il y a réellement eu contraction ou pas.

```
int kargerMinCut(Graphe G0)
{
    int ns = G0.nb_sommets;
    int na = G0.nb_aretes;

    // Initialisation de la partition initiale des sommets
    subset* partition = (subset*)malloc(n*sizeof(subset));
    for (int i = 0; i < ns; i++)
    {
        partition[i].parent = i;
        partition[i].rang = 0;
    }

    // Contraction de n-2 aretes
    int nb_aretes_contractees = 0;
```

```
    while (nb_aretes_contractees != ns-2)
    {
        int indice_arete = rand() % na;
        Arete a = G0.tab_aretes[indice_arete];
        int res = contracteArete(G0, partition, a);
        if (res == -1)
        {
            nb_aretes_contractees++;
        }
    }

    // Calcul de la taille de la coupe et liberation memoire
    int taille_coupe = compteAreteCoupe(G0,partition);
    free(partition);
    return taille_coupe;

}
```

Techniquement ceci n'est pas l'algorithme de Karger car on ne devrait pas pouvoir rechoisir une arête déjà contractée étant donné que cette dernière devrait être supprimée. Mais en l'occurrence on ne peut pas modifier le graphe au fur et à mesure. D'ailleurs, en théorie l'algorithme ci-dessus peut ne pas terminer car les choix aléatoires pourraient faire considérer en boucle la même arête ad vitam æternam.

Remarque : Je ne vois aucun intérêt dans toute cette partie à distinguer G et G_0 étant donné que l'énoncé utilise fréquemment la notation G pour désigner le graphe initial – en particulier dans les questions 34 et 35. Peut être faudrait-il mentionner que l'algorithme demandé ne fonctionnerait plus sur des graphes ayant trop d'arêtes à cause du fonctionnement de rand*.*

37. Cette complexité dépend de celle de Unir et Trouver, qu'on ne connaît pas puisque leur implémentation n'est pas donnée. Etant donné que l'énoncé parle d'une implémentation de union-find avec union par rang et compression de chemin, on fait l'hypothèse que ces deux fonctions sont correctement implémentées et qu'elles ont donc une complexité en $O(\alpha(n))$ où n est le nombre d'éléments manipulés dans la partition et α l'inverse de la fonction d'Ackermann.

    Comme dit en question 36, techniquement l'implémentation proposée produit un algorithme qui pourrait ne pas terminer... On va estimer (ce qui est clairement abusif) qu'il est possible à chaque itération de la boucle while de mettre à jour la liste d'arêtes encore choisissables pour être contractées en temps constant de sorte à ce qu'on ne puisse jamais rechoisir une arête déjà contractée.

    Notons $n = |S|$ et $m = |A|$. Alors, l'initialisation de la partition se fait en $O(n)$. Avec l'estimation précédente, on fait exactement $n - 2$ contractions, qui chacune nécessite un appel à Unir d'où un coût en $O(n\alpha(n))$ pour les contractions. Puis on calcule la taille de la coupe en faisant $O(m)$ appels à Trouver d'où $O(m\alpha(n))$ opérations. On obtient une complexité totale en $O((m+n)\alpha(n))$. Donc pas linéaire en la taille du graphe mais tout comme, ce qui confirme que la remarque "Le résultat précédent n'est pas satisfaisant d'un point de vue complexité" au début de la partie III est pessimiste.

    Remarque : Il me semble difficile de répondre à cette question de manière pertinente. Les complexités de Unir *et* Trouver *pourraient en théorie être n'importe quoi puisqu'on n'en a pas l'implémentation, ce qui avec un peu de mauvaise foi fait qu'on ne peut pas répondre à la question. Le fait qu'on ne supprime pas les arêtes du graphe au fil des contractions fait que la complexité de l'algorithme de la question 36 – qui n'est pas l'algorithme de Karger – est aléatoire : je ne pense pas qu'une analyse de l'espérance de sa complexité était attendue et si elle l'était, je ne sais pas la faire proprement.*