Sujet CCINP (`SET COVER` et arbres de Braun) : corrigé

Exercice A

1. Le problème `SET COVER` est :

   $\begin{cases} \textbf{Entrée} : \text{Un ensemble } E \text{ fini, } E_1, ..., E_n \text{ des sous ensembles de } E \text{ et } k \in \mathbb{N}. \\ \textbf{Question} : \text{Existe-t-il } I \subset [\![1, n]\!] \text{ tel que } \bigcup_{i \in I} E_i = E \text{ et } |I| \leq k \,? \end{cases}$

2. Un certificat consiste en une partie $I \subset [\![1, n]\!]$. Une telle partie peut par exemple être représentée par un tableau de taille n contenant vrai en case i si $i \in I$ et faux sinon : la taille d'un certificat est donc polynomiale en la taille de l'entrée.

   Vérifier si une telle partie est solution se fait aussi en temps polynomial en la taille de l'entrée : compter le nombre d'éléments se fait en $O(n)$ et faire l'union des E_i impliqués peut se faire en $O(n|E|)$ en faisant le ou bit à bit de chacun des au plus n tableaux de $|E|$ cases les représentant.

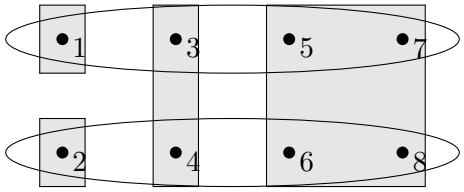
3. Si I_V est une instance positive de `VERTEX COVER` alors il existe un sous ensemble C des sommets de G tel que $|C| \leq k$ et toute arête de G touche au moins un sommet de C. Alors $\{E_s \,|\, s \in C\}$ est de cardinal inférieur à k et l'union de ces ensembles est $A = E$ donc I_S est une instance positive de `SET COVER`.

   La réciproque est vraie : si I_S est une instance positive de `SET COVER`, il existe $C \subset S$ tel que $|C| \leq k$ et $\bigcup_{s \in C} E_s = E = A$ et donc C est une couverture par sommets de G montrant que I_V est une instance positive de `VERTEX COVER`.

4. Étant donnée une instance I_V de `VERTEX COVER`, construire l'instance I_S de `SET COVER` associée se fait en temps polynomial en la taille de I_V. En effet, si on représente G par listes d'adjacence, la construction de E nécessite un parcours des arêtes en $O(|A|)$ et la construction d'un ensemble E_s de l'ordre de $\deg s$ opérations pour un coût total de construction de ces ensembles en $O\left(\sum_{s \in S} \deg s\right) = O(|A|)$.

   Ce fait combiné à la question 3 montre que `VERTEX COVER` se réduit à `SET COVER`. Comme `VERTEX COVER` est NP-dur, il en va donc de même pour `SET COVER` qui est de plus NP d'après la question 1 achevant ainsi la preuve de sa NP-complétude.

5. Un dessin éclaire la situation : voici les sous ensembles de l'instance considérée :

   On peut couvrir E avec E_5 et E_6 et on ne peut pas couvrir E avec un seul sous ensemble donc la taille de la couverture minimale est de 2. Mais l'algorithme glouton proposé peut choisir dans cet ordre les sous ensembles E_1, E_2, E_3, E_4, produisant une couverture de taille 4.

6. On peut représenter I à l'aide d'une liste et les autres ensembles à l'aide de tableaux de taille $m = |E|$. Une itération de la boucle tant que nécessite $O(m)$ opérations pour tester si U est vide, $O(nm)$ opérations pour calculer i tel que E_i couvre le plus possible d'éléments encore non couverts (U pour Uncovered), $O(1)$ opérations pour ajouter i à I et $O(m)$ opérations pour mettre à jour U.

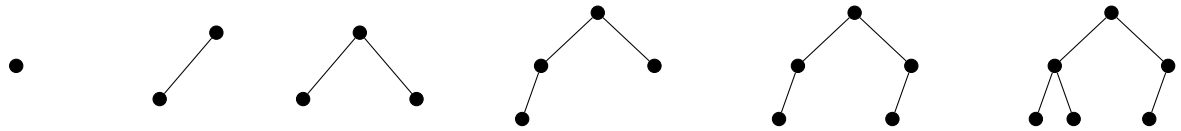
A chaque itération, soit on s'arrête, soit on couvre au moins un nouvel élément de E donc on fait au pire m itérations. On en déduit une complexité à la louche en $O(nm^2)$ qui est bien polynomiale en la taille de l'entrée.

7. Considérons pour tout $n \geq 1$ l'instance I_n suivante : $E = [\![1, 2^n]\!]$, E_1 est l'ensemble des nombres pairs de E, E_2 est l'ensemble des nombres impairs de E et on ajoute à ces deux sous-ensembles n sous ensembles contenant $1, 1, 2, 4, 8, ..., 2^{n-1}$ éléments de E de manière disjointe (à la manière de la question 5 par exemple). C'est toujours possible car $1 + \sum_{i=0}^{n-1} 2^i = 2^n = |E|$.

   Dans ces conditions, il est clair qu'une couverture optimale est donnée par $\{E_1, E_2\}$ et est donc de taille $C_n^\star = 2$. Mais l'algorithme glouton peut sélectionner une partition à $C_n = n$ éléments. S'il existait une constante α tel que l'algorithme proposé soit une α-approxiamtion pour `SET COVER`, on aurait pour tout $n \geq 1$, $C_n/C_n^\star = n/2 \leq \alpha$ ce qui est une contradiction.

Exercice B

1. On obtient les squelettes suivants ;



   On constate que pour tout $n \in [\![1, 6]\!]$ il y a un unique arbre de Braun de taille n. C'est en fait vrai pour tout entier (démonstration par récurrence forte).

2. Les contraintes sur un arbre de Braun font que sa hauteur est la hauteur de son fils gauche donc :

```
let rec hauteur (a:braun) :int = match a with
  |E -> -1
  |N(_,g,_) -> hauteur g + 1
```

   Dans ces conditions, la complexité de est en $O(h)$ donc en $O(\log n)$ d'après l'énoncé.

3. Par définition d'un tas, l'élément minimal est à la racine :

```
let minimum (a:braun) :int = match a with
  |E -> max_int
  |N(r,_,_) -> r
```

4. Dans le code proposé, pour insérer x dans un arbre de la forme $N(r, g, d)$, on construit un arbre de racine $m = \min(x, r)$ et on y insère $M = \max(x, r)$ pour respecter la propriété de tas. Pour respecter la propriété des arbres de Braun, on insère M dans le sous arbre le plus petit, c'est-à-dire d, puis on permute sous arbre gauche et droit.

   Il est clair que l'arbre ainsi construit vérifie la propriété de tas et contient le nouvel élément x. De plus c'est bien un tas de Braun car on est dans l'un des deux cas suivants :

   - Soit on avait $|g| = |d|$ avant insertion. Alors le nouvel arbre droit est plus grand de un que le nouvel arbre gauche donc les permuter rétablit la situation.
   - Soit on avait $|g| = |d| + 1$ avant insertion. Alors les nouveaux arbres gauche et droit sont désormais des arbres de Braun de même taille : peu importe de quel côté de la racine ils sont.

5. C'est immédiat puisque dans le cas où $t = N(r, g, d)$, g et d sont des arbres de Braun de taille égale à 1 près avec g le plus grand lorsqu'elles ne sont pas égales. On est donc pile dans les conditions d'utilisation de la fonction `fusionner` :

```
let rec extraire_min (a:braun) :int*braun = match a with
  |E -> failwith "extraction dans un arbre vide"
  |N(x,g,d) -> x, fusionner g d
```

6. On utilise le même principe que pour l'insertion : on extrait l'élément dans l'arbre le plus gros, donc à gauche, et on inverse fils gauche et fils droit pour conserver la propriété des arbres de Braun.

```
let rec extraire_element (a:braun) :int*braun = match a with
  |E -> failwith "extraction dans un arbre vide"
  |N(r,E,E) -> r, E
  |N(r,g,d) -> let x, gg = extraire_element g in x, N(r,d,gg)
```

7. Cette fois il n'y a besoin de faire attention qu'à la propriété de tas car le nombre d'éléments dans l'arbre ne change pas donc la propriété des arbres de Braun est plus facilement respectée.

```
let rec remplacer_min (a:braun) (x:int) :braun = match a with
  |E -> failwith "pas de minimum à remplacer"
  |N(r,g,d) -> let min_gauche = minimum g and min_droit = minimum d in
               if x <= min min_gauche min_droit
               then N(x,g,d)
               else
                 if min_gauche <= min_droit
                 then N(min_gauche, remplacer_min g x, d)
                 else N(min_droit, g, remplacer_min d x)
```

8. On cherche à fusionner g et d sachant que ce sont des arbres de Braun tels que $|g| = |d|$ ou $|g| = |d|+1$.
   - Si d est vide, on renvoie évidemment g.
   - Si g est vide mais pas d : c'est impossible vu la contrainte sur les tailles.
   - Sinon, $g = N(rg, gg, dg)$ et $d = N(rd, gd, dd)$. Dans ce cas, il faut déterminer qui sera la racine du nouveau tas de Braun, qui sera soit rg soit rd selon comment ces deux éléments se comparent :
     - Si $rg \leq rd$, la nouvelle racine est rg et on peut récursivement fusionner gg et dg pour obtenir un nouveau tas de Braun ayant $|g| - 1$ éléments. Comme pour l'insertion, on permute ce tas avec d pour respecter la condition sur les tailles.
     - Sinon, il faut faire de rd la nouvelle racine mais on ne peut pas diminuer le nombre d'éléments de d sans violer la contrainte sur les tailles. L'astuce est donc de récupérer un élément dans g et d'écraser le minimum de d avec cette valeur pour rééquilibrer.

9. La complexité de `minimum` est en $O(1)$. Celle de `inserer` est en $O(h)$ donc en $O(\log n)$ d'après la question 9. La complexité de `extraire_min` est celle de `fusionner`. Or dans cette fonction, on descend le long d'une branche de l'arbre en faisant des opérations en temps constant à chaque étage (pour un coût total en $O(h)$) et éventuellement si $rg > rd$ on fait ensuite un appel à `extraire_element` et un à `remplacer_min` qui s'exécutent toutes deux en $O(h)$. On obtient donc une complexité totale pour `extraire_min` en $O(\log n)$. On aurait eu les mêmes complexités avec un tas binaire.