DS 2

Option informatique, deuxième année

Julien Reichert

Partie 1 : Programmation

Question 1.1 : Écrire une fonction qui retourne à partir de rien le nombre de façons d'obtenir chaque entier pertinent comme somme de trois dés à six faces. L'esprit de l'exercice étant de servir de support à un calcul de probabilités, l'ordre des dés est important et donc additionner chaque nombre de la réponse doit donner 216. La structure est au choix mais elle doit correctement et explicitement associer chaque somme de trois dés possibles à son nombre d'occurrences.

Question 1.2 : Étendre la fonction précédente à une fonction qui prend en entrée le nombre de dés à six faces. Bien entendu, réussir pleinement cette fonction dispense de faire la question précédente.

Question 1.3 : Quelle est la complexité de la fonction précédente en fonction du nombre de dés ? (Le nombre de faces étant une constante, elle n'apparaît pas dans le calcul de complexité.)

Question 1.4 : Écrire une fonction qui prend en argument une liste et qui retourne la liste de toutes ses permutations.

Question 1.5 : Écrire une fonction qui prend en argument une chaîne de caractères dont on suppose que tous les éléments sont dans la table ASCII de base et qui détermine le ou les caractères le(s) plus fréquent(s), en renvoyant la liste contenant tous les caractères à égalité pour ce maximum du nombre d'occurrences (le nombre d'occurrences lui-même n'est pas à retourner).

Partie 2 : Structures de données et algorithmes

On suppose défini le type arbre de la manière suivante :

```
type arbre = | Feuille of int | Noeud of arbre * arbre ;;
```

On dit qu'un arbre est un peigne si tous les noeuds à l'exception éventuelle de la racine ont au moins une feuille pour fils. [1] On dit qu'un peigne est strict si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est rangé si le fils droit d'un noeud est toujours une feuille. [2] Un arbre réduit à une feuille est un peigne rangé.

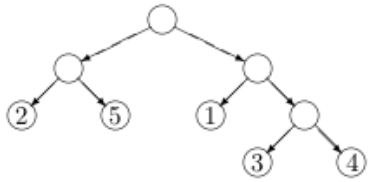

Figure 1 – Un peigne à cinq feuilles

1. Attention, ce n'est donc pas comme dans le cours.
2. Notion de cours correspondante : peigne gauche.

Question 2.1 : Représenter un peigne rangé à 5 feuilles.

Question 2.2 : La hauteur d'un arbre est le nombre de noeuds maximal que l'on rencontre pour aller de la racine à une feuille (la hauteur d'une feuille seule est 0). Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.

Question 2.3 : Écrire une fonction `est_range : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.

Question 2.4 : Écrire une fonction `est_peigne_strict : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict. En déduire une fonction `est_peigne : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne.

On souhaite ranger un peigne donné. Supposons que le fils droit `N` de sa racine ne soit pas une feuille. Notons `A1` le sous-arbre gauche de la racine, `f` l'une des feuilles du noeud `N` et `A2` l'autre sous-arbre du noeud `N`. On va utiliser l'opération de rotation qui construit un nouveau peigne où :

— le fils droit de la racine est le sous-arbre `A2` ;
— le fils gauche de la racine est un noeud de sous-arbre gauche `A1` et de sous-arbre droit la feuille `f`.

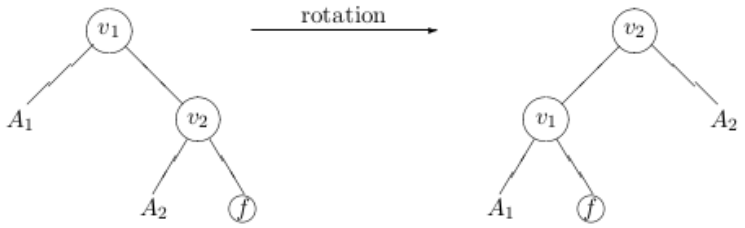

FIGURE 2 – Une rotation

Question 2.5 : Donner le résultat d'une rotation sur l'arbre de la figure 1.

Question 2.6 : Écrire une fonction `rotation : arbre -> arbre` qui effectue l'opération décrite ci-dessus. La fonction renverra l'arbre initial si une rotation n'est pas possible.

Question 2.7 : Écrire une fonction `rangement : arbre -> arbre` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

Partie 3 : Quand même une énigme de logique

Résoudre ce QCM, dont les questions et réponses sont interdépendantes.

Question 1 : Quelle est la réponse à cette question ?

a) a b) b c) c d) d

Question 2 : Quelle est la réponse à la question 5 ?

a) c b) d c) a d) b

Question 3 : Laquelle de ces questions a une réponse différente des trois autres ?

a) La question 3 b) La question 6 c) La question 2 d) La question 4

Question 4 : Quelle paire de questions a la même réponse ?

a) Les questions 1 et 5 b) Les questions 2 et 7 c) Les questions 1 et 9 d) Les questions 6 et 10

Question 5 : Laquelle de ces questions a la même réponse que cette question-ci ?

a) La question 8 b) La question 4 c) La question 9 d) La question 7

Question 6 : Quelle paire de questions a la même réponse que la question 8 (donc les trois ont la même réponse) ?

a) Les questions 2 et 4 b) Les questions 1 et 6 c) Les questions 3 et 10 d) Les questions 5 et 9

Question 7 : Parmi les dix questions, quelle lettre est le moins souvent la réponse ?

a) c b) b c) a d) d

Question 8 : Laquelle de ces questions n'a pas une réponse voisine dans l'alphabet de la réponse à la question 1 ?

a) La question 7 b) La question 5 c) La question 2 d) La question 10

Question 9 : Soient p le booléen « Les réponses aux questions 1 et 6 sont les mêmes. » et q le booléen « Les réponses aux questions x et 5 sont les mêmes ». Si $p \wedge q$ est faux mais $p \vee q$ est vrai, que vaut x ?

a) 6 b) 10 c) 2 d) 9

Question 10 : Parmi les dix questions, que vaut le nombre d'occurrences maximal d'une réponse moins le nombre d'occurrences minimal d'une réponse ?

a) 3 b) 2 c) 4 d) 1

Partie 4 : Jeux à un joueur

Un jeu à un joueur est la donnée d'un ensemble non vide E, d'un élément $e_0 \in E$, d'une fonction $s : E \to \mathcal{P}(E)$ et d'un sous-ensemble F de E. L'ensemble E représente les états possibles du jeu. L'élément e_0 est l'état initial. Pour un état e, l'ensemble $s(e)$ représente tous les états atteignables en un coup à partir de e. Enfin, F est l'ensemble des états gagnants du jeu. On dit qu'un état e_p est à la profondeur p s'il existe une séquence finie de $p+1$ états $e_0 e_1 \dots e_p$ avec $e_{i+1} \in s(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une solution du jeu, de profondeur p. Une solution optimale est une solution de profondeur minimale. On notera qu'un même état peut être à plusieurs profondeurs différentes.

Voici un exemple de jeu :

$$E = N \setminus \{0\} \; ; e_0 = 1 \; ; s(n) = \{2n, n+1\}$$

Question 4.1 : Donner une solution optimale pour ce jeu lorsque $F = \{42\}$.

Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un pseudo-code pour un tel parcours est donné figure 3.

```
BFS()
    A ← {e_0}
    p ← 0
    tant que A ≠ ∅
        B ← ∅
        pour tout x ∈ A
            si x ∈ F alors
                renvoyer VRAI
            B ← s(x) ∪ B
        A ← B
        p ← p+1
    renvoyer FAUX
```

FIGURE 3 – Parcours en largeur

Question 4.2 : Montrer que le parcours en largeur renvoie `VRAI` si et seulement si une solution existe.

Question 4.3 : On se place dans le cas particulier du jeu de la question 4.1 pour un ensemble F arbitraire pour lequel le parcours en largeur de la figure 4 termine. Montrer alors que la complexité en temps et en espace est exponentielle en la profondeur p de la solution trouvée. On demande de montrer que la complexité est bornée à la fois inférieurement et supérieurement par deux fonctions exponentielles en p.

Dans la suite, on suppose donnés un type `etat` et les valeurs suivantes pour représenter un jeu en Caml :

```
initial: etat
suivants: etat -> etat list
final: etat -> bool
```

Question 4.4 : Écrire une fonction `bfs : unit -> int` qui effectue un parcours en largeur à partir de l'état initial et renvoie la profondeur de la première solution trouvée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire.

Indication : On pourra avantageusement réaliser les ensembles `A` et `B` par des listes, sans chercher à éliminer les doublons, et utiliser une fonction récursive plutôt qu'une boucle.

Question 4.5 : Montrer que la fonction `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Comme on vient de le montrer, l'algorithme BFS permet de trouver une solution optimale mais il peut consommer un espace important pour cela, comme illustré dans le cas particulier du jeu de la question 4.1 qui nécessite un espace exponentiel. On peut y remédier en utilisant plutôt un parcours en profondeur. La figure 4 contient le pseudo-code d'une fonction DFS effectuant un parcours en profondeur à partir d'un état `e` de profondeur `p`, sans dépasser une profondeur maximale `m` donnée.

```
DFS(m, e, p)
    si p > m alors
        renvoyer FAUX
    si e ∈ F alors
        renvoyer VRAI
    pour chaque x dans s(e)
        si DFS(m, x, p + 1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

FIGURE 4 – Parcours en profondeur

Question 4.6 : Montrer que `DFS(m, e_0, 0)` renvoie `VRAI` si et seulement si une solution de profondeur inférieure ou égale à `m` existe.

Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec `m = 0`, puis avec `m = 1`, etc., jusqu'à ce que `DFS(m, e_0, 0)` renvoie `VRAI`. (Il s'agit d'une recherche itérée en profondeur.)

Question 4.7 : Écrire une fonction `ids : unit -> int` qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette fonction ne termine pas.

Question 4.8 : Montrer que la fonction `ids` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 4.9 : Comparer les complexités en temps et en espace du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants : (i) il y a exactement un état à chaque profondeur p ; (ii) il y a exactement 2^p états à la profondeur p. On demande de justifier les complexités qui seront données.