Partie I - Grammaire non contextuelle

Soit la grammaire non contextuelle $G = (\Sigma, V, P, S)$, l'ensemble des règles de production P étant défini par :

$$\begin{aligned} S &\rightarrow SaS \mid A \\ A &\rightarrow AbA \mid B \\ B &\rightarrow BcB \mid \varepsilon \end{aligned}$$

où Σ (respectivement V) est l'ensemble de symboles terminaux (respectivement non terminaux), $S \in V$ est le symbole initial de G et ε dénote le mot vide.

Soit u le mot abc.

Q1. Donner une dérivation à gauche de u. Que peut-on en déduire sur le mot u ?

Q2. Donner deux arbres de dérivation pour le mot aa. En déduire que G est ambigüe.

Une grammaire est dite *récursive gauche directe* si elle possède une règle de la forme $A \rightarrow A\alpha$, où α est une suite de symboles terminaux et non terminaux. A est dite *variable récursive gauche*. Une telle grammaire pose divers problèmes en analyse syntaxique descendante, on cherche alors à éliminer cette récursivité. Ceci est toujours possible pour les langages réguliers.

Q3. Identifier dans G les variables récursives gauches.

Pour éliminer la récursivité gauche, on utilise l'algorithme suivant :
Soit $A \rightarrow A\alpha \mid \beta$ une règle, où α est une suite de symboles terminaux et non terminaux et β est une suite de symboles terminaux et non terminaux ne commençant pas par A.
On remplace cette règle par :

(i). une règle commençant par A : $A \rightarrow \beta A'$,
(ii). une règle pour une nouvelle variable A' : $A' \rightarrow \alpha A' \mid \varepsilon$.

Q4. Construire la grammaire non récursive gauche directe G' équivalente à G.

Q5. Prouver que le langage reconnu par G' (ou G) est $\{a, b, c\}^*$.

Partie II - Problème de bin-packing

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

Soient $n, k \in \mathbb{N}^2$. On appelle *rangement* de n objets dans k boîtes une fonction $\mathcal{R} : [\![1, n]\!] \rightarrow [\![1, k]\!]$. L'ensemble $B_j = \mathcal{R}^{-1}(j) = \{i \in [\![1, n]\!], \mathcal{R}(i) = j\}$ est le contenu de la boîte $j \in [\![1, k]\!]$.

Étant données des tailles $a = (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0, 1])^n$, a_i étant la taille de l'objet i, on dit qu'un rangement $\mathcal{R}$ de n objets dans k boîtes est *valide* pour les tailles a si pour tout $j \in [\![1, k]\!]$ on a $\sum_{i \in B_j} a_i \leq 1$.

On considère alors le problème de décision BIN-PACKING qui, étant donnés $n, k \in \mathbb{N}^2$ et $a \in (\mathbb{Q} \cap]0, 1])^n$, décide s'il existe un rangement $\mathcal{R}$ de n objets dans k boîtes valide pour les tailles a.

On peut décrire une solution de ce problème par un couple $(k, \mathcal{R})$.

Q6. Montrer que BIN-PACKING est dans NP.

Soit le problème PARTITION suivant :
Pour n entiers $c_1 \cdots c_n$, existe-t-il un sous-ensemble S de $[\![1, n]\!]$, tel que $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$?

On admet que le problème PARTITION est NP-complet.

Q7. Montrer par réduction depuis PARTITION que le problème BIN-PACKING est NP-complet.

On s'intéresse par la suite au problème MIN-BIN-PACKING qui demande de déterminer le plus petit k pour lequel il existe un rangement $\mathcal{R}$ de n objets de taille a dans k boîtes. On introduit pour résoudre ce problème une heuristique dite *Premier Casier Décroissant* (**algorithme 2**), se basant sur l'heuristique *Premier Casier* décrite dans l'**algorithme 1**. Dans cet algorithme, le rangement $\mathcal{R}$ est modélisé par une liste de boîtes $\mathcal{L}$.

Algorithme 1 - Algorithme Premier Casier

Entrées : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0,1])^n$
Sorties : Une solution $(k, \mathcal{R})$
début
  $\mathcal{L}$: liste de boîtes ouvertes
  $\mathcal{L} = \emptyset$
  pour *i=1 à n* **faire**
    Rechercher la première boîte de $\mathcal{L}$ dans laquelle l'objet i peut être placé
    si *une telle boîte existe* **alors**
      l'objet i est placé à l'intérieur
    sinon
      une nouvelle boîte est ajoutée à $\mathcal{L}$ et l'objet i y est placé
  k := longueur($\mathcal{L}$)
  Déduire $\mathcal{R}$ de $\mathcal{L}$.

Algorithme 2 - Algorithme Premier Casier Décroissant

Entrées : $n \in \mathbb{N}, (a_1 \cdots a_n) \in (\mathbb{Q} \cap]0,1])^n$
Sorties : Une solution $(k, \mathcal{R})$
début
  σ = permutation telle que $a_{\sigma(1)} \geq a_{\sigma(2)} \cdots \geq a_{\sigma(n)}$
  $k, \mathcal{R}_1$:= `Premier Casier`$(n, a_{\sigma(1)}, \cdots, a_{\sigma(n)})$
  pour *i = 1 à n* **faire**
    $\mathcal{R}(i) := \mathcal{R}_1(\sigma(i))$

Nous allons montrer que l'algorithme *Premier Casier Décroissant* est une $\frac{3}{2}$-approximation pour le problème MIN-BIN-PACKING.
On note dans la suite k le nombre de boîtes retournées par l'**algorithme 2**, k^* le nombre de boîtes optimal (défini comme le nombre minimal de boîtes permettant de répondre au problème MIN-BIN-PACKING) et $k_1 = \lceil 2k/3 \rceil$, où $\lceil . \rceil$ désigne la partie entière supérieure. B_{k_1} fait référence à la boîte déduite du rangement renvoyé par l'**algorithme 2**.

Q8. Montrer que si B_{k_1} contient un objet i de taille $a_i > \frac{1}{2}$, alors $k^* \geq k_1 \geq \frac{2}{3}k$.

Q9. Montrer que sinon, les boîtes $B_{k_1} \cdots B_k$ contiennent au moins $2(k - k_1) + 1$ objets, aucun d'eux ne pouvant rentrer dans les boîtes $B_1 \cdots B_{k_1 - 1}$.

Q10. Justifier que $\sum_{i=1}^{n} a_i > min(k_1 - 1, 2(k - k_1) + 1)$. En admettant que pour $k \in \mathbb{N}$, $\frac{2}{3}k + \frac{2}{3} \geq \lceil 2k/3 \rceil$, en déduire que $k^* \geq \left\lceil \frac{2}{3}k \right\rceil \geq \frac{2}{3}k$.

On définit des types record pour représenter les objets et les boîtes en OCaml :

```
type objet = {
  id : int; (*identifiant de la boite*)
  taille : float; (*taille de la boite*)
}

type boite = {
  charge : float; (*somme des tailles des objets de la boîte*)
  objets : objet list; (*liste courante des tailles des objets dans la boîte*)
}
```

Q11. Écrire une constante `boite_vide : boite` représentant une boîte vide.

Q12. Écrire une fonction de signature

`ajoute_boite : objet -> boite -> boite`

qui renvoie la boîte obtenue en ajoutant un objet à une boîte donnée.

Q13. Écrire une fonction récursive de signature

`trouve_boite : boite list -> objet -> int`

telle que `trouve_boite bl o` retourne l'indice (à partir de 0) de la première boîte dans `bl` pouvant contenir `o` ou la taille de la liste `bl` si aucune boîte ne peut contenir `o`.

Q14. Écrire une fonction récursive de signature

`transforme : 'a -> ('a -> 'a) -> int -> 'a list -> 'a list`

telle que `transforme d f i l` retourne la liste `l` où l'élément `x` à l'indice `i` (à partir de 0) a été remplacé par `f x`. Si `i`est plus grand que la taille de la liste, `f d` est ajouté à la fin.

Q15. Écrire une fonction de signature

`premier_casier : objet list -> boite list`

qui applique l'**algorithme 1**. On retournera directement la liste de boîtes, on ne cherchera pas à calculer la fonction $\mathcal{R}$. On pourra utiliser la fonction `transforme` précédemment écrite.

Q16. Écrire une fonction de signature

`premier_casier_decroissant : objet list -> boite list`

qui applique l'**algorithme 2**.
On pourra utiliser `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` qui trie la liste passée en argument dans l'ordre croissant suivant la fonction passée en argument (qui renvoie -1, 0 ou 1 pour inférieur, égal ou supérieur respectivement). On pourra utiliser la fonction de comparaison générique `compare : 'a -> 'a -> int`. Là encore, on retournera directement la liste des boîtes.

Partie III - Algorithmes de couplage

Cette partie comporte des questions nécessitant un code en C.

On s'intéresse ici à un problème d'appariement entre deux groupes d'individus U et V, que l'on suppose de même cardinalité n.
On cherche à apparier les éléments de V aux éléments de U, chaque élément devant être apparié à exactement un élément de l'autre ensemble. Chaque élément de V (respectivement de U) a, de plus, un ordre de préférence total entre tous les éléments de U (respectivement de V).

Définition 1 (Couplage, couplage parfait)
Un ensemble de paires $A \subseteq V \times U$ est un couplage *si tout $x \in V$ et tout $y \in U$ apparaissent dans au plus un élément de A. Le couplage est dit* parfait *si tout $x \in V$ et tout $y \in U$ apparaissent dans un et un seul élément de A.*