DS 4

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou dans le titre du problème, en fonction de la situation.

On impose `type 'a arbin = Vide | Noeud of 'a arbin * 'a * 'a arbin` pour les arbres binaires en OCaml.

Questions de cours ou d'application directe du cours

Question de cours 1 : Donner un exemple d'ensemble ordonné sans élément minimal, un exemple d'ensemble ordonné ayant plusieurs éléments minimaux et un exemple d'ensemble ordonné sans plus petit élément mais avec un seul élément minimal. Justifier dans les trois cas.

Question de cours 2 : Quel résultat du cours a légitimé les preuves de terminaison des fonctions récursives ?

Question de cours 3 : Définir la notion d'arbre binaire de recherche.

Question de cours 4 : Expliquer brièvement (pas plus d'une dizaine de lignes) le principe de la programmation dynamique (problématique, intérêt, mise en œuvre).

Question de cours 5 : Définir la notion d'arité dans au moins un des deux contextes (associés...) où elle était mentionnée en cours.

Question de cours 6 : Pourquoi ne peut-on pas dire rigoureusement qu'un arbre binaire est un arbre dont chaque nœud a au plus deux enfants ?

Question de cours 7 : Expliquer brièvement (moins de 10 lignes) comment justifier que le meilleur algorithme de tri faisant des comparaisons est de complexité dominant $n \log n$, où n est la taille de la séquence à trier.

Question de cours 8 : Donner deux applications utiles de l'algorithme utilisant le principe de la « médiane des médianes » vu au TD 7.

Exercices issus des TD et TP

Exercice T1 : Écrire un algorithme glouton pour le problème d'allocation des salles de cours, détaillé ci-après.

Le problème d'allocation des salles de cours consiste à sélectionner pour chaque cours, représenté par un temps de début et un temps de fin, une salle où il peut avoir lieu, sans que deux cours n'aient lieu en même temps dans une même salle, et en minimisant le nombre de salles. On ne considère pas que deux cours ont lieu en même temps si le temps de fin de l'un est égal au temps de début de l'autre.

Exercice T2 : Trouver un algorithme DPR qui calcule le nombre d'inversions dans un tableau ou dans une liste (au choix). L'écrire en OCaml.

On rappelle qu'une inversion dans `t` est un couple `(i,j)` tel que `i < j` et `t.(i) > t.(j)`. Si la complexité n'est pas optimale, la question n'est pas corrigée.

Exercice T3 : Écrire une fonction en C qui vérifie si une chaîne de caractères correspond à un mot bien parenthésé au regard de `'('` et `')'`. On pourra ignorer les autres caractères de la chaîne, mais on ne supposera pas leur absence.

Exercice T4 : Écrire en C une fonction qui calcule le nombre d'arbres binaires de taille `n` de formes différentes.

Exercices

Exercice 1 : Écrire en C une fonction prenant en argument un tableau d'entiers et sa taille, et déterminant si pour tout entier naturel `i` strictement inférieur à la taille du tableau, l'élément d'indice `i` est un optimum local strict de la partie du tableau démarrant à sa position, c'est-à-dire qu'il est strictement inférieur à tous les éléments à sa droite ou strictement supérieur à tous les éléments à sa droite.

Exercice 2 : On considère un tableau croissant de caractères qui sont tous des lettres en majuscule. Ce tableau représente les paquets où déposer vos copies, pour mettre en contexte. On considère également un tableau dont chaque élément est un couple de chaînes de caractères représentant vos identités. Écrire en OCaml une fonction qui permet de calculer à partir de ces deux arguments le nombre de personnes qui ne peuvent pas se tromper en déposant leur copie car le prénom et le nom conduiraient à déposer la copie dans le même paquet.

Explication pour l'exercice 2 : Si le tableau est `[| 'C' ; 'H' ; 'V' |]`, cela correspond à un paquet avec les initiales de A à C, un avec les initiales de D à H, et un avec les initiales de I à V. Quelqu'un appelé J. R. ne pourrait par exemple pas se tromper car J et R mènent au troisième paquet. Et de toute manière J. R. ne se trompe pas.

Problème 1 : Autour de la suite de Syracuse [intégralement en C].

On considère la (en pratique un pluriel s'imposerait sans doute) suite de Syracuse, définie pour un u_0 quelconque par la relation de récurrence

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

La conjecture de Collatz (portant également d'autres noms) s'énonce ainsi : « Pour tout entier naturel non nul u_0, existe-t-il un rang n pour lequel $u_n = 1$? ». Cette conjecture n'est toujours pas démontrée à l'heure actuelle. Le but de cette partie est de faire quelques exercices autour de la suite de Syracuse.

Question P1.1 : Calculer à la main les termes de la suite avec $u_0 = 19$ jusqu'à atteindre 1.

On définit plusieurs valeurs associées à un entier u_0 :
- Le temps de vol est le premier indice n tel que $u_n = 1$. Ce sera 20 pour l'exemple.
- Le temps de vol en altitude est le premier indice n tel que $u_n < u_0$.
- L'altitude maximale est le maximum de la suite (en admettant que 1 soit atteint).

Question P1.2 : Pour la suite avec $u_0 = 19$, préciser le temps de vol en altitude et l'altitude maximale. Par ailleurs, que dire du temps de vol en altitude si u_0 est pair, et comment se comporte la suite après que 1 a été atteint ?

Pour le calcul informatique des valeurs ci-avant, dans certains cas il est pertinent de le faire indépendamment, mais dans d'autres mieux vaut profiter d'autres valeurs de u_0 en faisant appel à la mémoïzation.

Question P1.3 : Commencer par écrire une fonction calculant le terme suivant de la suite en prenant le terme actuel en argument. On utilisera cette fonction dans les exercices suivants.

Question P1.4 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant le temps de vol en altitude de la suite de Syracuse avec u_0 valant `N`.

Question P1.5 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant le tableau des temps de vol de toutes les suites de Syracuse avec u_0 compris entre `1` et `N`. Il est attendu que l'on ne gaspille pas de la complexité en temps par le calcul de l'intégralité des suites en question. L'approche top-down est attendue vu l'impossibilité de faire un bottom-up a priori. La structure de mémoire intermédiaire pourra au choix être un tableau ou une implémentation de tableau associatif.

Question P1.6 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant le tableau des altitudes maximales de toutes les suites de Syracuse avec u_0 compris entre `1` et `N`. Les consignes sont les mêmes que pour l'exercice précédent.

Problème 2 : Recherche de minimum local [intégralement en OCaml].

On définit un minimum local dans une liste, un tableau ou un tableau de tableaux comme un élément inférieur ou égal à tous ses voisins directs (jusqu'à deux dans une liste ou un tableau, jusqu'à quatre dans un tableau de tableaux).

Question P2.1 : Prouver qu'une séquence (liste, tableau, etc.) finie non vide possède un minimum local. Prouver aussi qu'une séquence infinie dont les éléments sont pris dans un ensemble muni d'un ordre total bien fondé possède un minimum local.

Question P2.2 : Écrire une fonction qui prend en entrée un tableau et qui retourne **un indice** d'un minimum local dans ce tableau.

Question P2.3 : Si cela n'a pas été fait à l'exercice précédent, réécrire cette fonction afin que la complexité soit logarithmique en la taille du tableau.

Question P2.4 : Écrire une fonction qui prend en entrée une liste et qui retourne un minimum local de cette liste. Il est interdit de passer par une structure de tableau. La complexité peut-elle être logarithmique ?

Question P2.5 : Écrire une fonction `minlocalfonction : (float -> float) -> float -> float -> float -> float` telle que `minlocalfonction f a b eps` retourne une abscisse entre `a` et `b` approchant un minimum local de `f` à `eps` près, en supposant que la fonction soit continue dans l'intervalle étudié et définie uniquement dans cet intervalle.

Question P2.6 : Écrire une fonction qui prend en entrée un tableau de tableaux (tous de même taille, et inutile de le vérifier) et qui retourne un couple d'indices (ligne, colonne) d'un minimum local de ce tableau de tableaux. Déterminer la complexité de la fonction.

Question P2.7 : Quelle serait la complexité si le tableau de tableaux était remplacé par une liste de listes (et avec quel algorithme) ?

Question P2.8 : Si cela n'a pas été fait à la question P2.6, écrire une fonction utilisant le paradigme diviser pour régner pour la recherche de minimum local dans un tableau de tableaux. Il s'agit de diviser le tableau de tableaux en quatre parties, d'étudier le cadre des quatre parties et d'en déduire laquelle de ces quatre parties est certaine de contenir un minimum local. Serait-il à ce sujet possible d'appliquer le même algorithme en restreignant l'étude à la croix séparant le tableau de tableaux en quatre parties ?

Question P2.9 : Donner la complexité de l'algorithme en écrivant la complexité c_n pour un tableau de tableaux à n lignes et n colonnes en fonction de $c_{\frac{n}{2}}$.

Question P2.10 : Adapter le calcul précédent à un algorithme alternatif (que l'on ne demandera pas d'écrire) divisant le tableau de tableaux en deux horizontalement ou verticalement (au choix), et écrire la complexité c_m pour un tableau de tableaux contenant au total m éléments (donc m est la somme des tailles des tableaux) en fonction de $c_{\frac{m}{2}}$.

Question P2.11 : On suppose maintenant que le tableau de tableaux contient des booléens et on donne l'ordre `false < true`. Écrire une fonction de complexité la plus faible possible (à déterminer) déterminant la position d'un minimum local.

Question P2.12 : Finalement, le tableau de tableaux contient des entiers naturels et on suppose que le maximum de la matrice est relativement faible par rapport à sa taille. Écrire une fonction de complexité linéaire en ce maximum et indépendante de la taille du tableau de tableaux déterminant la position d'un minimum local.

Annexe

On rappelle à toute fin utile des fonctions du module `Hashtbl` avec des informations sur leur spécification :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option pour éviter de lever une exception si la clé est absente ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).
- `Hashtbl.fold f th acc` appelle la fonction fournie, prenant des clés et des valeurs ainsi qu'un accumulateur (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage). La fonction renverra la valeur de l'accumulateur à la suite de tous ces appels successifs.

On donne également une version du Master Theorem :

Soit un algorithme DPR dont la complexité est définie par une relation de la forme $c_n = ac_{\frac{n}{b}} + \Theta(n^\alpha)$, avec $a \geq 1, b \in \mathbb{N} \setminus \{0,1\}$ et $\alpha \in \mathbb{R}_+$. Alors :

- Si $\alpha < \log_b(a)$, alors $c_n = \Theta(n^{\log_b(a)})$.
- Si $\alpha > \log_b(a)$, alors $c_n = \Theta(n^\alpha)$.
- Si $\alpha = \log_b(a)$, alors $c_n = \Theta(n^\alpha \log_b(n))$.