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 la différence entre les notions d'élément minimal et de plus petit élément dans un ensemble ordonné.

Question de cours 2 : Donner la définition de l'ordre structurel, exemple à l'appui (qui peut être donné en créant une structure en OCaml, si on souhaite).

Question de cours 3 : Montrer que l'ordre produit de deux ordres bien fondés est bien fondé.

Question de cours 4 : Justifier que les preuves par induction structurelle sont correctes.

Question de cours 5 : Expliquer le principe du paradigme « diviser pour régner », avec un exemple de problème adapté et un algorithme correspondant (on n'écrira pas de programme).

Question de cours 6 : Quelle est la borne inférieure de la complexité dans le pire cas d'un algorithme de tri utilisant des comparaisons ?

Question de cours 7 : Proposer un type en C permettant de réaliser des arbres binaires d'entiers.

Question de cours 8 : Proposer un type en OCaml permettant de réaliser des arbres d'arité quelconque dont les étiquettes sont de type quelconque.

Exercices issus des TD et TP

Exercice T1 : En admettant qu'une fonction nommée `mediane` fournisse la médiane d'un tableau d'entiers en temps linéaire sans fuite de mémoire, implémenter en C un tri rapide (en place ou non) sur des tableaux d'entiers utilisant la médiane comme pivot à chaque étape qui n'est pas un cas de base, et en calculer la complexité.

Exercice T2 : Écrire un algorithme glouton pour le problème du rendu de monnaie en euros (en centimes d'euros, en pratique, pour rester dans les entiers) et l'implémenter en C.

Exercice T3 : Écrire en OCaml une fonction qui prend en argument deux listes d'entiers et qui détermine si elles contiennent les mêmes éléments, chacune avec le même nombre d'occurrences.

Exercice T4 : Écrire en OCaml une fonction qui calcule la hauteur d'un arbre binaire.

Exercices

Exercice 1 : Si on voulait créer une structure d'ABR presque complet, quelle serait la complexité dans le pire des cas de l'ajout d'un élément et pourquoi ? On pourra expliquer le principe d'un algorithme d'ajout sans écrire de programme.

Exercice 2 : On souhaite représenter les nœuds d'un arbre binaire dans un tableau en OCaml, selon la représentation utilisée pour les arbres binaires presque complets. Ceci étant, les arbres binaires que l'on représentera ne seront pas presque complets. Pour pallier ce manque, on utilisera pour les tableaux un type option. Écrire une fonction de conversion des arbres selon le type imposé en début de sujet vers les tableaux qui les représentent et une fonction de conversion dans l'autre sens.

Exercice 3 : On définit un nombre semi-premier comme un nombre qui s'écrit comme le produit d'exactement deux nombres premiers, potentiellement égaux. Écrire en C une fonction prenant en argument deux entiers supérieurs à deux `mini` et `maxi` et renvoyant le tableau contenant tous les nombres `n` compris au sens large entre `mini` et `maxi` et tels que `n-1`, `n` et `n+1` soient tous trois semi-premiers.

Exercice 4 : On considère un escalier de `n` marches, et sur certaines marches figurent un objet (la dernière n'en contient pas). On peut grimper les escaliers à raison d'une à quatre marches par pas. Écrire en OCaml une fonction prenant en argument un tableau de booléens déterminant la présence d'un objet où non sur chaque marche et renvoyant le nombre minimum de marches contenant un objet sur lesquelles on est obligé de passer.

Exercice 5 : Même exercice, mais il existe aussi des marches avec des anti-objets, qui font diminuer le nombre d'objets effectivement empruntés d'un (le total pouvant devenir négatif). Le tableau en argument contient alors des entiers entre `-1` et `1`.

Exercice 6 : Même exercice, mais le nombre d'objets par marche est désormais n'importe quel entier naturel, ce qui adapte encore la nature du tableau. **Cette fois-ci, il faut programmer en C, et la taille du tableau en argument est un deuxième argument.**

Problème : Autour de la recherche d'un élément majoritaire.

On considère une séquence (tableau d'entiers en C, tableau ou liste en OCaml dont les éléments sont quelconques) de taille notée `n`. L'élément majoritaire de la séquence, s'il existe, est un élément de la séquence dont le nombre d'occurrences est strictement supérieure à la moitié de `n`, ce qui correspond exactement à la notion de majorité absolue. Il existe de nombreux algorithmes de recherche associés, et nous allons en étudier quelques-uns dans ce problème.

Question P1 [cadeau] : Justifier l'utilisation d'un article défini pour désigner **l'**élement majoritaire (avec une preuve rapide).

Question P2 : Écrire en C une fonction prenant en argument un tableau d'entiers et sa taille ainsi qu'un entier et déterminant le nombre d'occurrences de cet entier dans le tableau.

Question P3 : En déduire une fonction en C déterminant par un algorithme naïf l'existence d'un élément majoritaire dans un tableau (la taille est encore en argument) et calculer sa complexité. Difficulté particulière : pour éviter les faux positifs, la fonction renverra un pointeur d'entier égal à `NULL` s'il n'existe pas d'élément majoritaire et dont le déréférencement fournira l'élément majoritaire sinon.

Dans un but d'amélioration de la complexité, voici un algorithme DPR élégant. On prend les éléments deux par deux et on en met une occurrence dans une nouvelle séquence s'ils sont égaux, mais on ne fait rien s'ils sont différents. Si la taille de la séquence est impaire, on met le dernier élément dans la nouvelle séquence. Une fois le parcours de la séquence terminé, on recommence le tout sur la nouvelle séquence, jusqu'à ce qu'il ne reste plus qu'un élément ou aucun. S'il n'en reste aucun, on conclut qu'il n'y a pas d'élément majoritaire, et s'il en reste un, on le compte dans la séquence de départ pour conclure.

Question P4 : Implémenter cet algorithme sur des listes en OCaml. La fonction renverra un élément majoritaire ou rien en utilisant un type option.

Question P5 : Après avoir fait ce qu'il faut faire préalablement à un calcul de complexité, poser la formule de récurrence donnant la complexité de chacune des deux étapes, puis en déduire la complexité de l'algorithme (inutile de faire une preuve longue, mais une phrase de justification pour le passage de la formule de récurrence à la valeur est attendue).

Une autre version de complexité acceptable revient simplement à considérer une version triée de la séquence et à étudier le nombre de valeurs identiques consécutives pour chaque élément.

Question P6 : Implémenter un algorithme de tri au choix **mais de complexité négligeable devant le carré de la taille de l'argument** sur un tableau en OCaml (on pourra muter l'argument ou non).

Question P7 : Implémenter l'algorithme de recherche d'élément majoritaire fondé sur un tri détaillé ci-avant sur un tableau en OCaml (on ne mutera pas l'argument). La fonction à écrire renverra le premier indice dans le tableau de départ de l'élément majoritaire s'il existe et `-1` sinon.

En mentant sur la complexité des opérations sur un dictionnaire, on peut faire mieux : créer le dictionnaire des nombres d'occurrences de chaque élément et le parcourir afin de conclure.

Question P8 : Implémenter cet algorithme sur des listes en OCaml avec la même valeur de retour que pour la question P4.

Question P9 : Il existe en pratique une méthode qui est forcément de complexité linéaire en temps et en espace et qui utilise un algorithme avancé vu cette année. De quoi s'agit-il ? (Préciser l'algorithme en question et comment il peut s'utiliser pour répondre au problème de la recherche de l'élément majoritaire.)

Raffinement ultime auquel il fallait penser : l'algorithme de vote de Boyer et Moore (qui est différent d'un autre algorithme des mêmes auteurs que l'on verra dans un chapitre ultérieur). L'algorithme est de complexité linéaire en temps et constante en espace, et revient à maintenir une variable (le « candidat » à être majoritaire à l'heure actuelle) et un compteur (son « score » à l'étape actuelle). Concrètement, pour chaque élément de la séquence pendant son parcours :

— soit le compteur est nul et on mémorise l'élément en tant que candidat, en passant le compteur à un ;
— soit l'élément traité est le candidat actuel avec un compteur non nul, et on incrémente le compteur ;
— soit l'élément traité est différent du candidat actuel avec un compteur non nul, et on décrémente le compteur.

À la fin du parcours, le candidat est le seul élément pouvant être l'élément majoritaire. On le compte dans la séquence pour vérifier s'il est effectivement majoritaire.

Question P10 : Implémenter cet algorithme sur des tableaux d'entiers en C, en renvoyant la réponse comme dans la question P3.

Question P11 : Prouver l'affirmation « À la fin du parcours, le candidat est le seul élément pouvant être l'élément majoritaire. » ci-avant.

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.