DS 3

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.

Questions de cours ou d'application directe du cours

Question de cours 1 : Définir la notion d'invariant de structure et préciser le lien avec la barrière d'abstraction.

Question de cours 2 : Quels sont les adjectifs contraires de « modifiable » pour une structure de données ?

Question de cours 3 : Quelle est la complexité de l'accès à l'élément du fond d'une liste simplement chaînée ?

Question de cours 4 : Proposer un type en C (écrire la ligne introduite par `struct` et le renommage par la suite si on le souhaite) pour implémenter une structure de maillon de liste chaînée d'entiers, et écrire le prototype de trois opérations élémentaires sur cette structure.

Question de cours 5 : Proposer un type en OCaml pour implémenter une structure de file au choix, et écrire la signature des fonctions formant une interface cohérente.

Question de cours 6 : Expliquer comment manipuler les arguments de la ligne de commande en donnant un exemple minimaliste de programme (par exemple la somme des entiers fournis, leur nombre étant arbitraire), une ligne de commande pour compiler (tolérance large pour des erreurs ici vu le peu de pratique) et une ligne de commande pour exécuter le programme en précisant le résultat obtenu.

Question de cours 7 : Définir la notion de sérialisation.

Exercices issus des TD et TP

Exercice T1 : Soit le type `nombre` défini ci-après. Écrire en OCaml une fonction prenant en argument deux valeurs de ce type et déterminant si elles sont égales.

```
type nombre = Entier of int | Flottant of float | Fraction of int * int | Moins_inf | Plus_inf
```

Exercice T2 : Implémenter en C la structure de pile bornée d'entiers, avec le prototype suivant pour les opérations.

```
int* creer_pileb(int capacite);
void empiler_pileb(int* p, int capacite, int x);
int depiler_pileb(int* p);
bool est_vide_pileb(int* p);
```

Exercice T3 : Écrire la fonction `exists` en OCaml de deux façons : d'abord sous forme d'une récursion, puis avec une des deux fonctions « fold ». Les signatures sont rappelées ci-après.

```
List.exists : ('a -> bool) -> 'a list -> bool
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Exercice T4 : Implémenter le tri cocktail sur des tableaux d'entiers en C. On rappelle l'algorithme : faire un passage montant du tri bulles, puis un passage descendant, et s'arrêter dès qu'un passage a permis de constater que le tableau est trié. On fera en outre des passages de plus en plus courts.

Exercices

Exercice 1 : Écrire en C une fonction de prototype `int classementsansmajor(int* tab, int taille, int x)` qui prend en argument un tableau d'entiers et sa taille ainsi qu'un entier et qui renvoie le rang de la valeur `x` selon un classement décroissant des valeurs du tableau en ignorant toutes les occurrences du maximum. On déclenchera une erreur (par exemple par `exit(1)`) si `x` n'est pas un élément du tableau ou s'il est égal au maximum.

Par exemple, la fonction précédente renverra `3` pour le tableau contenant `4, 1, 5, 3, 6, 3, 6, 5, 4`, sa taille de `9` et la valeur `4`, puisque les deux éléments valant `4` sont troisièmes après les deux éléments valant `5` et en ayant ignoré les deux éléments valant `6`.

Exercice 2 : On considère deux joueurs de poker (juste pour l'exemple) qui veulent abréger leur partie et font tapis sur chaque donne. En d'autres termes, à chaque étape, en notant a et b leurs fortunes, avec $a < b$ pour fixer les idées, soit le joueur de fortune a gagne et sa fortune devient $2a$ (celle de l'adversaire devient alors $b - a$) et l'étape suivante a lieu, soit le joueur de fortune a perd et la partie s'arrête sur une victoire de son adversaire. L'identité du gagnant de chaque partie étant stockée dans une séquence de booléens `vict` (liste ou tableau, au choix, les `true` signalant une victoire du premier joueur), et les fortunes initiales étant stockées dans deux variables `f1` et `f2`, écrire en OCaml une fonction `quigagne vict f1 f2` renvoyant un entier valant `-1` si le premier joueur finit ruiné à la fin de la séquence, `1` si le deuxième joueur finit ruiné à la fin de la séquence ou `0` si aucun joueur n'est ruiné à la fin de la séquence. Il faudra déclencher une erreur si un joueur finit ruiné avant la fin de la séquence.

Exercice 3 : Écrire en C une fonction prenant en argument un tableau d'entiers positifs (pas à vérifier) et sa taille et renvoyant le quatrième plus grand élément du tableau. Une contrainte supplémentaire faisant partie de la notation : on ne parcourt qu'une fois le tableau (et pas pour créer une copie). Il s'agit alors de stocker les quatre plus grands éléments dans une variable (attention à l'initialisation). Prouver la correction de la fonction écrite.

Exercice 4 : On considère un tableau de chaînes de caractères toutes constituées de mots en minuscules (pas à vérifier). Écrire en OCaml une fonction prenant en argument un tel tableau et qui renvoie la liste des couples de mots qui sont tous deux dans le tableau et qui de plus sont le miroir l'un de l'autre, le premier élément du couple devant être strictement inférieur au deuxième dans l'ordre alphabétique (ceci exclut en particulier les palindromes). L'ordre d'apparition des couples dans la liste n'est pas imposé. Calculer la complexité en temps dans le pire des cas de la fonction écrite.

Problème 1 : Multiset [intégralement en C]

La structure de données abstraite de multi-ensemble (*multiset* en anglais) est une collection non ordonnée d'éléments pouvant apparaître en plusieurs exemplaires, étendant ainsi la structure d'ensemble.

Comme un ensemble peut être vu comme un dictionnaire « bridé » en ignorant les valeurs associées aux clés, on peut envisager d'implémenter un multi-ensemble en implémentant un dictionnaire dont les clés sont les éléments apparaissant au moins une fois et les valeurs sont les nombres d'occurrences associés à chaque élément.

Une autre possibilité est d'utiliser un tableau (redimensionnable, vu que les structures d'ensemble et de multi-ensemble permettent l'ajout et le retrait d'élément) où les éléments sont stockés, soit dans l'ordre croissant (sous réserve que les éléments proviennent d'un ensemble ordonné), soit non, avec également la possibilité de rapprocher les éléments identiques. C'est cette dernière solution qui sera retenue pour l'intérêt pédagogique en programmation.

Les éléments de notre multi-ensemble seront forcément des entiers, et le type associé sera le tableau redimensionnable d'entiers.

Les opérations à écrire sont la création d'un multi-ensemble vide, l'ajout d'un élément à un multi-ensemble (c'est-à-dire l'ajout d'une occurrence d'un élément qui figure potentiellement déjà), le retrait d'un élément précisé d'un multi-ensemble (sans effet si l'élément n'y figure pas), le test de vacuité, la récupération du nombre d'occurrences d'un élément (potentiellement zéro), le test d'égalité de deux multi-ensembles, le test d'inclusion d'un multi-ensemble dans un autre, la réunion, l'intersection, la somme et la différence.

La spécification de ces quatre dernières fonctions mérite une clarification :

- La réunion de E et de F contient tous les éléments de E et de F, le nombre d'occurrences associé étant alors le maximum des nombres d'occurrences des éléments dans chaque multi-ensemble.
- L'intersection de E et de F contient tous les éléments communs à E et F, le nombre d'occurrences associé étant alors le minimum des nombres d'occurrences des éléments dans chaque multi-ensemble.
- La somme de E et de F contient tous les éléments de E et de F, le nombre d'occurrences associé étant alors la somme des nombres d'occurrences des éléments dans chaque multi-ensemble.
- La différence de E et de F contient tous les éléments apparaissant strictement plus souvent dans E que dans F, le nombre d'occurrences associé étant alors la différence des nombres d'occurrences des éléments qui s'en déduit.

Question P1.1 : Écrire l'interface du type tableau redimensionnable d'entiers, c'est-à-dire la création du type (ligne introduite par `struct` et le renommage par la suite si on le souhaite) et le prototype des opérations élémentaires, sans les écrire.

Question P1.2 : Écrire le prototype de toutes les opérations élémentaires sur la structure de multi-ensemble mentionnées ci-avant.

Question P1.3 : Écrire la fonction d'ajout d'un élément à un multi-ensemble. On rappelle qu'il s'agit de laisser les éléments identiques côte à côte.

Question P1.4 : Écrire la fonction de récupération du nombre d'occurrences d'un élément dans un multi-ensemble.

Question P1.5 : Écrire la fonction de test d'inclusion d'un multi-ensemble dans un autre et en déduire la fonction de test d'égalité de deux multi-ensembles.

Question P1.6 : Écrire les fonctions de construction de la réunion et de l'intersection de deux multi-ensembles. Si des similarités apparaissent dans la programmation, on peut se permettre de gagner du temps.

Question P1.7 : Écrire la fonction de construction de la somme de deux multi-ensembles.

Question P1.8 : Écrire la fonction de construction de la différence de deux multi-ensembles.

Problème 2 : À la découverte du tarot hongrois [intégralement en OCaml]

Le tarot hongrois, jeu de cartes par excellence de la promotion actuelle en MPI, est une source incroyable d'exercices (en plus d'être d'un intérêt certain pour les réflexions logiques et la théorie des jeux). Ce sujet en présente une partie du principe nécessaire pour chaque fonction de manière résumée et aussi indépendante que possible, en vue de résoudre les exercices uniquement (autrement dit, a priori il n'est pas possible de jouer simplement à partir de ces informations).

Le jeu se joue avec un paquet de quarante-deux cartes, dont vingt-deux atouts : l'excuse, notée `S` et associée au numéro vingt-deux, et les autres numérotés de `I` à `XXI` en chiffres romains. Les vingt cartes restantes sont, dans les quatre enseignes usuelles (pique, cœur, carreau et trèfle), les cinq cartes suivantes : un roi, une dame, un cavalier, un valet et une carte basse (dix noir / as rouge, qu'on appellera *pip* en utilisant le terme anglais pour les cartes dont la valeur est numérique donc qui ne sont pas des figures). La représentation retenue en OCaml pour l'ensemble des exercices est la suivante, où l'entier associé à un atout est compris entre `1` et `22` inclus :

```
type couleur = Pique | Coeur | Carreau | Trefle
type valeur = Roi | Dame | Cavalier | Valet | Pip
type carte = Coul of valeur * couleur | Atout of int
```

Quatre joueurs disposent de neuf cartes chacun et un talon de six cartes complète le paquet. Une **main** est un tableau de cartes, et une **distribution** est un tableau contenant quatre mains et le tableau des six cartes constituant le talon.

Question P2.1 : Une main est considérée comme cohérente si elle contient exactement neuf cartes elles-mêmes cohérentes (pas d'entier dépassant les limites annoncées ci-avant pour les atouts). Écrire une fonction vérifiant la cohérence d'une main en renvoyant un booléen.

Question P2.2 : Écrire une fonction vérifiant si une distribution est cohérente, dans le sens où les quatre mains sont cohérentes, le talon est cohérent (mais avec six cartes au lieu de neuf) et chaque carte du paquet apparaît exactement une fois.

Question P2.3 : Que dire de la complexité de la fonction précédente ?

La partie commence par une phase d'enchères pour déterminer qui sera le preneur et combien de cartes du talon il pourra récupérer. Pour avoir le droit de produire une enchère, les trois premiers joueurs après le donneur doivent posséder un `bout` (`I`, `XXI` ou `S`).

Question P2.4 : Écrire une fonction prenant en argument une main et déterminant si elle contient au moins un bout.

L'enchère finale est un nombre entre zéro et trois. Le joueur l'ayant produite récupère ce nombre de cartes du talon (les premières), puis le joueur suivant récupère un tiers de ce qui reste (arrondi supérieur), puis le joueur suivant récupère la moitié de ce qui reste (arrondi supérieur), puis le dernier joueur récupère ce qui reste. Cela donne donc dans l'ordre $3+1+1+1$ ou $2+2+1+1$ ou $1+2+2+1$ ou $0+2+2+2$ cartes.

Question P2.5 : Écrire une fonction qui prend en argument un indice de carte du talon (entre zéro et cinq) et un indice de joueur (entre zéro et trois) et qui détermine l'ensemble des enchères finales permettant au joueur en question de prendre la carte ayant cet indice. Les enchères finales comprendront l'information de la valeur de l'enchère et du joueur l'ayant produite.

Par exemple, pour la carte d'indice 0 et le joueur `2`, les enchères possibles seront `3` par le joueur `2`, `2` par le joueur `2`, `1` par le joueur `2` et `0` par le joueur `1`. On remarquera qu'il y aura toujours quatre enchères finales possibles, une par nombre de cartes. Cela peut guider dans le choix de la représentation de la réponse.

Les joueurs écartent par la suite autant de cartes que le nombre de cartes reçues, mais il est interdit d'écarter des rois et des bouts.

Par la suite, le joueur ayant remporté l'enchère appelle son partenaire : le joueur ayant le `XX` ou, sauf volonté explicite de jouer seul, le joueur ayant le plus haut atout inférieur à `XX` absent de la main du joueur ayant remporté l'enchère.

Question P2.6 : Écrire une fonction prenant en argument une main et retournant le numéro de l'atout à appeler selon le principe ci-avant si on veut un partenaire.

Par la suite, les joueurs vont faire des déclarations (mises sur des événements de jeux, qui ne seront pas détaillées ici) et le jeu démarre (pas de détail non plus). À la fin, les deux camps disposent chacun d'une partie du paquet, on compte les points et on détermine les bonus.

Le nombre de points d'une carte se calcule ainsi : cinq pour les bouts, cinq pour les rois, quatre pour les dames, trois pour les cavaliers, deux pour les valets, un pour les vingt-trois autres cartes.

Question P2.7 : Écrire une fonction prenant en argument un tableau de cartes (les cartes récupérées par un certain camp) et déterminant le nombre de points accumulés.

Un bonus est accordé à un camp s'il capture les quatre rois.

Question P2.8 : Écrire une fonction prenant en argument un tableau de cartes et déterminant s'il contient les quatre rois.

Le bonus le plus fort du jeu est remporté si un joueur capture le `XXI` d'un adversaire avec son excuse.

Question P2.9 (difficile, et encore, c'est une version simplifiée) : Écrire une fonction prenant en argument un tableau de tableaux de cartes contenant les neuf plis, quel que soit le camp qui les remporte, dans l'ordre des joueurs (la première carte de chaque tableau est toujours celle que le joueur `0` a jouée) ainsi qu'un tableau de quatre booléens précisant pour chaque joueur s'il est dans le camp du preneur, et qui renvoie un entier annonçant si le `XXI` a été pris par l'excuse d'un adversaire : `0` si cela n'était pas le cas, `1` si l'excuse du camp du preneur a capturé le `XXI` adverse et `-1` si le `XXI` du camp du preneur a été pris par l'excuse adverse.

(Il faut bien considérer que dans un éventuel moteur qui permettrait de jouer au jeu, la représentation des plis ne serait pas forcément la même, car il faudrait plutôt une version adaptée au calcul des points et des autres bonus, en séparant les plis entre les deux camps et en forçant à étudier chaque pli pour savoir qui l'a fait. On pourra consulter l'exercice `89` de la liste des exercices de l'année pour se faire une idée…)

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.clear th` vide la table de hachage en argument. La fonction `Hashtbl.reset` a le même effet mais la table de hachage revient en plus à son nombre initial de places.
- `Hashtbl.copy th` crée une copie de la table de hachage en argument.
- `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.length th` renvoie le nombre de clés (doublons compris) 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 (sans valeur de retour, donc effectuant juste un effet secondaire) 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.

Pour finir ce devoir, un petit jeu de code golf. Le but est de résoudre un exercice de programmation en écrivant une fonction en le moins de caractères possibles, espaces, tabulations et sauts de ligne compris (on exclut le caractère de fin). Bien entendu, l'esthétique est bafouée dans ce genre de situations.

Défi numéro 1 : écrire en Python une fonction prenant en argument un nombre de `1` à `12`, correspondant au numéro d'un mois, et renvoyant un booléen déterminant si le mois en question a trente-et-un jours.
Score à battre : 23 caractères.