DS 5

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'arbre bicolore, avec les règles concernant les couleurs.

Question de cours 2 : Expliquer les structures de tas et de file de priorité, et dire comment la première permet d'implémenter la deuxième.

Question de cours 3 : Expliquer deux représentations possibles d'un graphe informatiquement parlant.

Question de cours 4 : Définir la notion de graphe non orienté connexe et donner une méthode algorithmique (sans programmer) permettant de vérifier si un graphe est connexe.

Question de cours 5 : Écrire en pseudo-code un algorithme de recherche de chemin de poids minimal dans un graphe pondéré. L'algorithme est au choix et tous les arcs sont supposés avoir un poids strictement positif. Prouver la correction de l'algorithme (la preuve compte pour l'essentiel des points de la question).

Question de cours 6 : Expliquer un algorithme non naïf de recherche de motifs, sans programmer ni même écrire de pseudo-code. On pourra cependant illustrer le principe sur un exemple pertinent.

Question de cours 7 : Rappeler la définition d'un tri topologique. Donner une condition nécessaire et suffisante pour qu'un graphe admette un **unique** tri topologique.

Exercices issus des TD et TP

Exercice T1 : Écrire en OCaml une fonction de conversion d'un trie à un arbre PATRICIA et la fonction réciproque.

Pour la question précédente, on impose les types suivants :

```
type trie = N of char * bool * trie list

type patricia = NN of string * bool * patricia list;;
```

Exercice T2 : Implémenter la structure de dictionnaire en OCaml à l'aide d'un ABR.

Exercice T3 : Écrire en C une fonction de sérialisation d'un graphe et la fonction de désérialisation associée, en créant d'abord une structure au choix. La méthode de sérialisation est au choix.

Exercice T4 : Écrire en C une fonction prenant en argument un tableau d'entiers et déterminant s'il admet un facteur carré (c'est-à-dire deux sous-tableaux consécutifs identiques formés d'éléments consécutifs).

Exercices

On considère pour les trois premiers exercices des chaînes de caractères composées uniquement de `'0'` et de `'1'` (pas à vérifier).

Exercice 1 : Écrire en C une fonction prenant en argument une telle chaîne et déterminant si la plus longue suite de `'1'` consécutifs dans la chaîne est de taille exactement trois.

Exercice 2 : Même exercice, mais il faut déterminer si la plus longue suite de `'1'` consécutifs est de taille exactement trois ou la plus longue suite de la plus longue suite de `'0'` consécutifs est de taille exactement trois. Un seul booléen sera renvoyé et correspondra à la disjonction des deux conditions.

Exercice 3 : Même exercice en remplaçant « ou » par « et ».

Exercice 4 : Soient `u`, `v` et `w` trois chaînes de caractères. On dit que `w` est un shuffle (ou entrelacement) de `u` et `v` si on peut extraire de `w` les caractères de `u` dans l'ordre, et que ce qui reste forme exactement `v` dans l'ordre. Écrire en OCaml une fonction prenant en argument trois chaînes de caractères et déterminant si la troisième est un shuffle des deux premières. Il est impératif d'utiliser la programmation dynamique pour que l'exercice soit évalué.

Exercice 5 : Donner la complexité de la fonction écrite dans l'exercice précédent (sous réserve qu'elle soit effectivement écrite) et estimer la complexité qu'aurait eu un algorithme d'exploration exhaustive.

Exercice 6 : On considère l'ensemble des cases d'un échiquier à un moment d'une partie comme des sommets d'un graphe orienté. Les sommets sont associés à des informations sur le numéro de la case (afin d'avoir un identifiant) et le propriétaire de l'éventuelle pièce qui s'y trouve (`1` pour les noirs, `-1` pour les blancs, `0` si la case est vide). Une case occupée est dite *protégée* si au moins une pièce du même propriétaire que la pièce se trouvant sur la case en question peut se déplacer à l'endroit où elle se trouve (en faisant une prise de l'éventuelle pièce adverse qui s'y placerait entre temps). Une case occupée est dite *menacée* si au moins une pièce de l'adversaire du propriétaire de la pièce se trouvant sur la case en question peut se déplacer à l'endroit où elle se trouve (en faisant une prise, là aussi). Les informations sur les possibilités de déplacement sont prises en compte au moment de construire les arcs du graphe, et on ignore les subtilités quant aux déplacements spéciaux des pions, par exemple. Écrire en OCaml une fonction prenant en argument la représentation d'un échiquier (type au choix à écrire auparavant) et renvoyant la liste des cases menacées et non protégées où se situe une pièce blanche.

Problème 1 : Encore une histoire de rendu de monnaie

On s'intéresse dans ce problème à une extension du problème de rendu de monnaie : cette fois-ci, on considère une somme à payer, notée `s`, un système monétaire stocké dans un tableau `t` **supposé décroissant**, et on cherche à minimiser le nombre total de pièces et billets (on ne parlera que de pièces pour simplifier) utilisés pour le paiement puis pour le rendu de la monnaie.

Pour les questions en OCaml, il est autorisé de considérer que le système est stocké dans une liste décroissante.

Concrètement, la somme payée sera une valeur `sbis` supérieure ou égale à `s`, et la différence pourra être supérieure au maximum de `t`. On suppose que toutes les sommes sont des entiers et que le tableau `t` contient forcément la valeur `1`.

Question P1.1 : Donner un cas de système monétaire et de somme à payer pour lesquels la somme donnée est strictement supérieure à la somme à payer plus la plus grande valeur d'une pièce dans une transaction minimisant le nombre de pièces échangées.

Dans un premier temps, on considère un algorithme glouton, qui revient à utiliser le maximum de `t` tant que le total ne dépasse pas `s` si la pièce est incorporée, puis à compléter par la plus petite pièce qui fait dépasser `s`, puis pour la phase de rendu d'utiliser à chaque fois la plus grande pièce disponible sans que le total (décroissant) ne repasse en-dessous de `s`.

Question P1.2 : Implémenter cet algorithme en C, la fonction devra prendre en argument le système, sa taille et la somme à payer et renvoyer le nombre total de pièces utilisées en respectant le principe ci-avant.

Question P1.3 : Donner un cas de système monétaire et de somme à payer pour lesquels l'algorithme glouton ne donne pas la valeur optimale, en détaillant les calculs faits par l'algorithme et en fournissant une solution strictement meilleure.

Question P1.4 : Calculer la complexité de la fonction écrite en fonction des paramètres pertinents du problème.

Au vu de la faible complexité de l'algorithme glouton, on pourra se servir de la réponse fournie si on cherche une borne supérieure à la solution dans les algorithmes à suivre.

Commençons par un IDS adapté à notre problème. Concrètement, il s'agit d'explorer l'arborescence des possibilités de paiement et de rendu et de tout tester jusqu'à une certaine profondeur.

Question P1.5 : Écrire en OCaml une fonction procédant au parcours en profondeur itéré de toutes les possibilités. On renverra un couple de listes correspondant respectivement à la somme payée et à la somme rendue, dans une manière optimale de faire le paiement. L'ordre à l'intérieur des listes n'ayant pas d'importance.

Indication : Le plus intuitif est d'écrire une fonction récursive ayant pour arguments la somme actuelle et le détail actuel des pièces utilisées, dans un contexte où le système monétaire et la profondeur maximale sont connus, et de lancer cette fonction jusqu'à tomber sur la somme exacte en augmentant la profondeur maximale à chaque étape.

Question P1.6 : Calculer la complexité de la fonction précédente (on considèrera que la preuve de terminaison est faite).

Une méthode alternative consiste à simuler un parcours en largeur sans construire de graphe à proprement parler. Ici, dans la mesure où les sommes peuvent être négatives ou incontrôlablement élevées, on se servira d'un dictionnaire qui recensera pour toutes les sommes possibles (les clés) le nombre minimal de pièces nécessaires pour produire la somme en question (les valeurs), et les clés seront traitées dans une structure de file (on ne se servira pas du module `Queue`).

Question P1.7 : Implémenter une structure de file au choix en OCaml. Attention, on ne peut pas se contenter de files bornées ici.

Question P1.8 : Écrire en OCaml une fonction qui résout le problème avec les structures de données suggérées en renvoyant le nombre minimal de pièces nécessaires pour le paiement et le rendu de la somme en argument, le système étant également en argument.

Question P1.9 : Signaler comment adapter la fonction précédente pour renvoyer un couple de listes (comme dans la question P1.5), en écrivant les lignes de code qui en remplacent d'autres.

Problème 2 : Tri minmax [intégralement en C]

Avec un an d'avance, une petite incursion dans la théorie des jeux. Aucune connaissance dans le domaine n'est nécessaire pour répondre aux questions de ce problème.

On considère le tri par comparaison d'un tableau (de taille connue) comme un jeu entre Ève (qui souhaite trier la liste le plus vite possible et dont la stratégie s'apparente à un algorithme écrit à l'avance - ce qui correspond à l'étymologie de « programmer ») et Adam, qui souhaite ralentir la progression du tri en répondant aux comparaisons d'Ève de manière la plus défavorable possible.

Ce faisant, le tableau en argument du tri peut être considéré comme jamais révélé par Adam (on est dans le cadre de jeux à information imparfaite).

Par exemple, pour le tableau dont les éléments sont `1, 3, 4, 2`, une partie pourrait être formée ainsi :

— Ève : le premier élément est-il inférieur ou égal au deuxième ?
— Adam : Oui.
— Ève : le troisième élément est-il inférieur ou égal au quatrième ?
— Adam : Non.
— Ève : le premier élément est-il inférieur ou égal au quatrième ?
— Adam : Oui.
— Ève : le troisième élément est-il inférieur ou égal au deuxième ?
— Adam : Non.
— Ève : le deuxième élément est-il inférieur ou égal au quatrième ?
— Adam : Non.

La partie s'arrête car Ève peut conclure que l'ordre des éléments est : premier, quatrième, deuxième et troisième.

Nous allons réaliser ce jeu, certes de manière non optimale en termes de complexité, en remplaçant l'arène par une structure de mémoire pour Ève contenant le tableau des permutations encore possibles au vu des informations fournies.

Chaque réponse élimine donc des permutations possibles et rapproche Ève de la réponse quant à la façon de trier le tableau, sauf bien entendu si la question qu'elle a posée n'a pas d'intérêt.

Dans quelle mesure Adam a-t-il un rôle ? La question est légitime. En fait, puisque la connaissance du tableau n'est pas encore fournie à Ève au moment où le tri est programmé, on peut laisser à Adam le soin de répondre ce qu'il veut, à condition de rester cohérent, et donc de choisir la séquence qui l'arrange le plus à chaque étape.

Ainsi, le nombre de permutations que la réponse d'Adam permet d'éliminer sera toujours supposé inférieur au nombre de permutations que la réponse d'Adam ne permet pas d'éliminer.

Nous faisons donc face à une situation illustrant l'algorithme minmax, qui sera aussi au programme de deuxième année.

On supposera dans cette partie qu'il n'y aura jamais d'égalités entre les éléments, ce qui facilitera le travail au niveau de la fin de la partie et permettra de ne pas avoir à se poser la question d'utiliser des comparaisons larges ou strictes.

Le but de ce problème est de mettre dans un autre contexte la preuve de la borne inférieure de la complexité dans le pire des cas des tris par comparaison.

Question P2.1 : Rappeler sans preuve combien il existe de permutations d'un ensemble de taille n.

Pour raccourcir les énoncés suivants, on appellera *peen* une permutation de l'ensemble des entiers de `0` à `n-1`. On suppose par ailleurs que `n` sera toujours supérieur ou égal à deux.

Question P2.2 : Écrire une fonction permettant d'engendrer toutes les peen. La valeur de retour sera un tableau de tableaux (chaque tableau étant une peen), donc le prototype sera `int** engendre_peen(int n)`.

Question P2.3 : Écrire une fonction prenant en argument un tableau de peen (supposé non vide), un pointeur vers sa taille, la taille des peen (si on veut, on peut ne pas la mettre en argument en fait), deux indices distincts, notés respectivement `i` et `j` et entre `0` et `n-1`, et un booléen `b`. Cette fonction renverra un tableau contenant les peen du tableau de départ pour lesquelles l'élément d'indice `i` est inférieur à l'élément d'indice `j` si, et seulement si, le booléen `b` est vrai, et seulement elles. Il faudra aussi muter le pointeur vers la taille pour qu'il devienne le nombre de peen gardées **et mieux vaut anticiper la libération de la mémoire qui sera nécessaire dans les fonctions suivantes** (rien n'est cependant imposé, donc aucun choix pertinent n'est pénalisé).

Question P2.4 : Écrire une fonction prenant en argument un tableau de peen (supposé non vide), sa taille (et la taille des peen si on veut, comme dans la question précédente) et deux indices distincts comme dans la fonction précédente, et renvoyant le booléen qui maximise la taille du tableau qui serait renvoyé par la fonction précédente (en cas d'égalité, on pourra renvoyer n'importe quel booléen).

Question P2.5 : Écrire une fonction prenant en argument un tableau de peen et sa taille ainsi que la taille des peen et renvoyant un couple d'indices qui minimise la taille du tableau qui serait renvoyé par la fonction de la question P2.3 si le booléen renvoyé par la question P2.4 était utilisé avec ce couple d'indices. Il faudra éventuellement se servir d'une version légèrement modifiée de la fonction précédente (signaler les modifications éventuelles).

Question P2.6 : En déduire une fonction appelée `tri_minmax` prenant en argument un tableau et sa taille et qui filtre les peen possibles jusqu'à ce qu'il n'en reste qu'une (à renvoyer). Attention à la façon d'utiliser les fonctions précédentes !

Question P2.7 : Rappeler la formule de Stirling et s'en servir pour donner un équivalent en $+\infty$ du nombre de tours effectués dans le jeu que l'on simule par la fonction précédente.

Question P2.8 : Estimer cependant la complexité de la fonction de la question P2.6 et comprendre qu'on évitera d'employer un tel algorithme de tri.

Problème 3 : Oops ! [intégralement en OCaml]

Voici un problème autour d'un jeu découvert à l'automne et qui est très riche du point de vue de l'étude de graphes dynamiques.

Le jeu « Oops ! » est un casse-tête qui se présente sous la forme d'un plateau **de cinq cases par cinq** où figurent ce que l'on appellera des figurines : une tête, un haut-de-forme et des éléments colorés, parfois empilés par deux dès le début. Ci-après figure la mise en place du premier problème du jeu.

Le but du jeu est de former des empilements contenant au total tous les éléments colorés, avec la tête par-dessus l'un d'entre eux, et de les déplacer dans le haut-de-forme, l'empilement contenant la tête en dernier.

Les règles de déplacement sont les suivantes :
- Le haut-de-forme ne peut pas être déplacé, mais toutes les autres figurines le peuvent.
- Un déplacement consiste à prendre une figurine ou un empilement et à suivre latéralement le plus court chemin **en considérant que toutes les autres figurines sont des obstacles** pour terminer sur une cas où se trouve déjà une figurine ou un empilement. Dans ce cas, ce qui a été déplacé s'empile avec ce qui figure sur la case d'arrivée.
- Les empilements ne se séparent plus jamais et se déplacent totalement ensemble. En particulier, il est interdit d'empiler quelque chose sur la tête, et de déplacer l'empilement contenant la tête dans le haut-de-forme alors qu'il reste d'autres éléments ailleurs.
- Le chemin d'un déplacement doit être exactement de taille le nombre de figurines d'un empilement.

Tout problème a une unique solution.

Pour donner une idée, la solution au premier problème est la suivante :
- Déplacer la tête (distance 1) sur l'élément bleu.
- Déplacer la tête et l'élément bleu (distance 2) sur l'élément rouge.
- Déplacer l'empilement ainsi formé (distance 3) sur l'élément jaune.
- Déplacer l'empilement ainsi formé (distance 4) sur l'élément vert.
- Déplacer l'empilement ainsi formé (distance 5) sur le haut-de-forme.

La modélisation informatique du problème consistera dans un premier temps à simuler la grille en tant que matrice de codes, puis d'en déduire le graphe des figurines. Pour le graphe en question, on ne construira que les arcs pouvant effectivement être empruntés en respectant les règles.

Les codes correspondront à un type somme dont la définition se comprend intuitivement :

```
type case = Vide | Hautdeforme | Tete of int | Element of int
```

Les entiers paramétrant les deux derniers constructeurs seront les tailles des empilements, avec une distinction permettant de savoir quelle figurine contient la tête. En cas de fusion avec le haut-de-forme, l'information de la taille disparaît car elle n'a alors plus de pertinence.

Pour les graphes, on utilisera la représentation donnant explicitement les sommets puis les arcs :

```
type sommet = { ligne : int ; colonne : int ; est_tete : bool ; taille : int }
type graphe = { sommets : sommet list ; arcs : (sommet * sommet) list }
```

L'ordre des sommets dans la liste pourrait être arbitraire, mais pour faciliter la lecture et la compréhension, ils seront donnés dans l'ordre de lecture usuel (ligne par ligne et colonne par colonne, ce qui correspond à l'ordre lexicographique de la double indexation). Le haut-de-forme correspond à un sommet dont l'information sur la taille est zéro. Les deux dernières informations sur les sommets sont certes redondantes avec la matrice des codes mais cela permet au graphe de se suffire à lui-même (en vue de ne plus repasser par la matrice ultérieurement).

La représentation sous forme de graphe du problème donné en exemple est la suivante :

```
let sommet1 = { ligne = 0 ; colonne = 1 ; est_tete = false ; taille = 1 }
let sommet2 = { ligne = 2 ; colonne = 3 ; est_tete = true ; taille = 1 }
let sommet3 = { ligne = 2 ; colonne = 4 ; est_tete = false ; taille = 1 }
let sommet4 = { ligne = 3 ; colonne = 2 ; est_tete = false ; taille = 1 }
let sommet5 = { ligne = 4 ; colonne = 0 ; est_tete = false ; taille = 0 }
let sommet6 = { ligne = 4 ; colonne = 4 ; est_tete = false ; taille = 1 }
let graphe1 = { sommets = [ sommet1 ; sommet2 ; sommet3 ; sommet4 ; sommet5 ; sommet6 ] ;
arcs = [(sommet2, sommet3)] }
```

Question P3.1 : Donner les quatre représentations sous forme de graphe des configurations intermédiaires lors de la résolution du problème en exemple, avec par exemple un dessin rapide (lisible sans être fignolé) de l'état du plateau pour bien délimiter les graphes créés.

Dans la question précédente, on pourra être concis dans la représentation en OCaml.

Question P3.2 : Donner la représentation sous forme de matrice de codes du problème en exemple.

Question P3.3 : Écrire une fonction prenant en argument une matrice de codes et déterminant s'il s'agit d'une configuration finale.

Question P3.4 : Écrire à l'aide d'un parcours en largeur une fonction prenant en argument une matrice de codes, un indice de ligne et un indice de colonne et renvoyant la matrice de mêmes dimensions telle qu'à chaque position où figure autre chose que `Vide` dans la matrice en argument, la distance (au sens des règles du jeu) entre cette position et la position représentée en argument figure au même endroit dans la matrice renvoyée. Les valeurs aux autres indices peuvent être quelconques, on pourra mettre des `-1` ou la distance tout de même, par exemple.

Le résultat de cette fonction sur la matrice de codes attendue dans la question P3.2 et avec l'indice de ligne `2` et l'indice de colonne `3` est le suivant (en ne mettant que les valeurs importantes en lumière) :

```
[|
[| ? ; 4 ; ? ; ? ; ? |];
[| ? ; ? ; ? ; ? ; ? |];
[| ? ; ? ; ? ; 0 ; 1 |];
[| ? ; ? ; 2 ; ? ; ? |];
[| 5 ; ? ; ? ; ? ; 3 |]
|]
```

Question P3.5 : Écrire une fonction prenant en argument une matrice de codes et renvoyant le graphe correspondant ainsi que la fonction réciproque.

Question P3.6 : Écrire une fonction prenant en argument une matrice de codes et deux couples de coordonnées (un indice de ligne puis un indice de colonne) et renvoyant la même matrice à la modification près que la figurine ou l'empilement au premier couple de coordonnées est déplacé au deuxième couple de coordonnées.

Dans la question précédente, la fonction n'a pas besoin de déclencher d'erreur si le déplacement ainsi suggéré ne respecte pas les règles, car la vérification passant par le graphe serait redondante.

On prendra garde à ne pas muter l'argument.

Question P3.7 : Écrire une fonction prenant en argument une matrice de codes et renvoyant la liste des couples correspondant aux mouvements qui respectent les règles.

Pour encoder un problème, on utilise une chaîne de caractères de taille vingt-cinq (le découpage en lignes étant non ambigu) dont les caractères possibles sont un point pour le vide, un `T` pour la tête (forcément de taille 1), un `H` pour le haut-de-forme et un nombre pour un élément ou empilement dont la taille est le nombre en question (forcément 1 ou 2 dans le jeu).

Question P3.8 : Écrire une fonction de désérialisation d'une telle chaîne de caractères en une matrice de codes.

Question P3.9 : Écrire à l'aide d'un algorithme d'exploration exhaustive une fonction résolvant un problème donné en tant que matrice de codes.

Le format de la valeur de retour de la fonction précédente aura tout intérêt à être une liste de couples au sens de la question P3.7 décrivant dans le bon ordre les mouvements effectués. Ceci permet de rendre la question suivante plus agréable.

Question P3.10 : Écrire finalement une fonction prenant en argument la description d'un problème en tant que chaîne de caractères et imprimant les instructions pour résoudre le problème associé.

L'impression devra être formée de lignes sous la forme « Déplacer de (i, j) vers (k, l). » avec les indices correspondants. Pour l'exemple, ce serait donc :

```
Déplacer de (2, 3) vers (2, 4).
Déplacer de (2, 4) vers (4, 4).
Déplacer de (4, 4) vers (3, 2).
Déplacer de (3, 2) vers (0, 1).
Déplacer de (0, 1) vers (4, 0).
```

Question P3.11 : Résoudre à la main (sans nécessairement passer par tous ces algorithmes) le problème représenté ci-après, avec l'encodage présenté avant la question P3.8 mais organisé en lignes et en colonnes avec de l'espacement, pour la lisibilité.

1	1	.	.	.
2	T	2	H	.
.	.	.	.	.
.	1	.	.	.
.	.	.	.	.

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.