DS 6

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé le cas échéant.

Questions de cours

Question de cours 1 : Construire l'arbre de Huffman associé à la chaîne `"si six cent scies scient six cent cigares, six cent six scies scieront six cent six cigares"`.

Question de cours 2 : Décoder la chaîne dont l'encodage par l'algorithme LZW a donné la séquence `110; 101; 118; 101; 114; 32; 103; 111; 110; 110; 97; 261; 105; 258; 32; 121; 111; 117; 32; 117; 112; 44; 32; 256; 258; 260; 262; 264; 266; 103; 268; 101; 270; 272; 32; 102; 114; 101; 287; 112; 111; 105; 110; 116; 115`.

Question de cours 3 : Définir la notion de tautologie.

Question de cours 4 : Définir la notion de système complet de connecteurs.

Question de cours 5 : Définir la notion de forme normale conjonctive.

Question de cours 6 : Définir la jointure à droite.

Question de cours 7 : Définir les notions de clé, de clé primaire et de clé étrangère (de manière à bien distinguer les différences).

Exercices issus des TD et TP

Exercice T1 : Écrire en C une fonction prenant en argument deux chaînes de caractères, la deuxième étant le motif, et retournant la liste des positions où le motif est trouvé dans l'autre chaîne. On utilisera l'algorithme naïf.

Exercice T2 : Écrire en OCaml une fonction sans argument permettant de remplir un carré magique normal de neuf cases (rappel : contenant les entiers de `1` à `9` une et une seule fois et dont toutes les sommes sur les lignes, colonnes et diagonales sont les mêmes). La méthode est au choix mais il faudra procéder à un moment par retour sur trace. En particulier, donner la solution à la main est exclu.

Exercice T3 : Écrire une fonction d'évaluation pour le type `f_b` défini ci-après. L'argument supplémentaire est une liste de chaînes de caractères représentant une interprétation et correspondant aux variables s'évaluant à vrai.

```
type f_b = Vrai | Faux | Var of string | Non of f_b | Et of f_b list
| Ou of f_b list;;
```

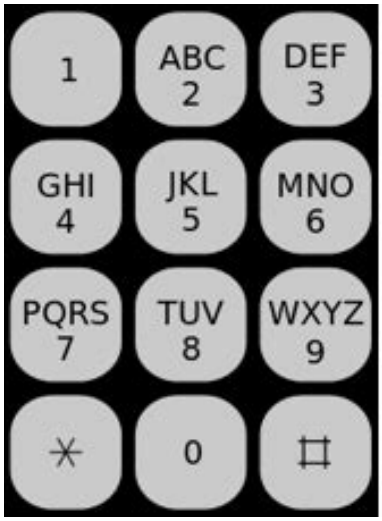
Exercices

Exercice 1 : Écrire un « tri backtracking » en OCaml, dont le principe est le suivant : on construit à partir d'une liste `l` la version triée de la liste en extrayant de `l` un élément et en continuant si l'extraction peut encore donner un résultat croissant (on peut optimiser l'exploration exhaustive ou non, au choix). Sinon, on procède par retour sur trace.

Exercice 2 : On considère un clavier de téléphone (ancien...) dont les touches sont associées à des lettres comme ci-après. La saisie T9 permet d'écrire un mot à l'aide de la séquence de chiffres correspondant aux lettres du mot, par exemple `"ELIXIR"` s'obtient par la séquence `354947`. Cette saisie est certes non injective a priori... Écrire en C une fonction prenant en entrée une liste de mots représentée par un trie (la structure est donnée ci-après) et une chaîne de chiffres correspondant à une saisie et renvoyant le tableau des mots possibles associés à cette chaîne de chiffres.

```
struct trie { char lettre ; bool fini ;
struct trie* successeurs ; int nombre_successeurs ; };
// La lettre à la racine est \0

typedef struct trie trie;
```



Exercice 3 : Écrire en OCaml une fonction récursive calculant l'inverse d'une matrice en utilisant le complément de Schur, selon le principe donné ci-après. Il s'agira d'une part de faire l'inversion d'une matrice définie positive comme décrite plus bas, et ensuite d'utiliser le résultat annoncé dans le premier paragraphe pour inverser une matrice générique. On pourra supposer sans le vérifier que la taille de la matrice est une puissance de deux.

Soit M une matrice carrée supposée inversible. D'après les propriétés de la transposition, on a $M^{-1} = ({}^tMM)^{-1} \times {}^tM$ et donc on peut ramener le calcul de l'inverse d'une matrice au calcul de l'inverse d'une matrice symétrique.

Soit donc M symétrique de taille une puissance de 2 (pour simplifier). M étant inversible dès qu'elle est définie positive, on va supposer ceci (c'est le cas pour une matrice de la forme ${}^tNN$, ce qui tombe bien).

On décompose M en blocs de taille identique, en notant que le bloc en haut à droite et le bloc en bas à gauche sont la transposée l'un de l'autre, d'où les blocs (dans l'ordre de lecture) $A, {}^tB, B, C$.

On appelle complément de Schur de C la matrice $S = C - B \times A^{-1} \times {}^tB$.

Il s'avère que l'inverse de M est exactement

$$\begin{pmatrix} A^{-1} + A^{-1} \times {}^tB \times S^{-1} \times B \times A^{-1} & -A^{-1} \times {}^tB \times S^{-1} \\ -S^{-1} \times B \times A^{-1} & S^{-1} \end{pmatrix}$$

Dans ce cas, l'inversion d'une matrice de taille $2n$ se fait récursivement (les matrices en question sont elles aussi symétriques), et on en déduit l'inverse de M à l'aide de deux inversions de matrices de taille n, ainsi qu'un nombre restreint de multiplications matricielles, là aussi sur des matrices de taille n.

Exercice 4 : Prouver que la complexité de l'inversion matricielle par la méthode de l'exercice précédent est dominée par la complexité du produit (en sachant que la complexité du produit domine n^2, ce qui n'est pas à justifier).

Problème 1 - Base de données « voyages » [intégralement en SQL]

Pour recenser les voyages faits depuis un an, une possibilité est de construire la base de données suivante :

— Table `VOYAGE`, avec les attributs `idvoyage` (entier), `transport` (chaîne de caractères donnant le moyen de transport), `depart` (date de départ au format AAAA-MM-JJ) et `retour` (date de retour au même format).
— Table `ETAPE`, avec les attributs `idvoyage` (entier), `arrivee` (date d'arrivée sur place au format AAAA-MM-JJ), `localisation` (chaîne de caractères, nom d'un hôtel par exemple), `ville` (chaîne de caractères), `pays` (chaîne de caractères), `ppj` (flottant, prix par jour pour l'hébergement), `satisfaction` (entier, note de zéro à dix).

La correspondance entre les tables est intuitive. On considère que le temps passé sur une étape est un nombre entier de jours, soit l'écart entre la date d'arrivée et la date d'arrivée à l'étape suivante, soit le cas échéant l'écart entre la date d'arrivée et la date du retour (pour simplifier). En particulier, une valeur de l'attribut `arrivee` est forcément unique.

Exercice P1.1 : La relation entre `VOYAGE` et `ETAPE` est-elle `1-1`, `1-*` ou `*-*` ? Justifier.

Exercice P1.2 : Écrire une requête affichant (une seule fois...) le pays où se situe la ville de Lermoos (ce pays est unique).

Exercice P1.3 : Écrire une requête affichant toutes les villes où nous avons fait une étape dans notre voyage ayant débuté le 22 juillet 2025 (ce voyage est unique et on peut se servir de cette information) dans l'ordre chronologique.

Exercice P1.4 : Écrire une requête affichant toutes les localisations ayant maximisé notre satisfaction (ce maximum peut être strictement inférieur à `10`).

Exercice P1.5 : Écrire une requête affichant le nombre de voyages par moyen de transport (en précisant celui-ci à chaque fois, bien entendu).

Exercice P1.6 : Écrire une requête affichant le nombre total de jours qu'ont duré les voyages enregistrés dans la base de données.

Exercice P1.7 : Écrire une requête affichant, s'il en existe, les informations sur une étape dont la date d'arrivée est incohérente avec les dates du voyage.

Exercice P1.8 : Écrire une requête affichant, s'il en existe, les informations sur un voyage dont la date de départ n'est pas la date d'arrivée à la première étape.

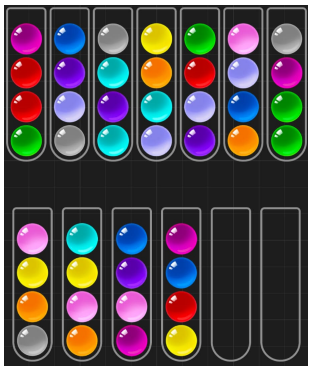
Exercice P1.9 : Écrire une requête affichant le nombre maximal de pays différents visités au cours d'un même voyage.

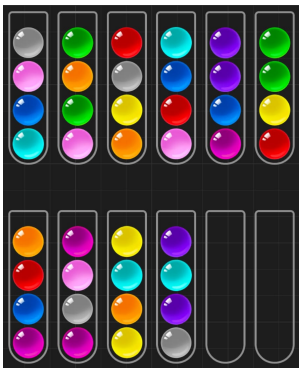
Exercice P1.10 : Écrire une requête affichant le coût total d'hébergement du voyage d'identifiant `1`.

Attention, cette dernière requête est **très** difficile. On a en particulier *très probablement* besoin de la fonction `DATEDIFF`, issue de SQL Server et fonctionnant légèrement différemment avec MySQL, offerte avec un exemple de fonctionnement : `DATEDIFF(day, '2025-07-22', '2025-08-01')` renvoie `10` car il y a dix jours d'écart entre les deux dates mentionnées.

Problème 2 - Ball sort [intégralement en C]

Le jeu `Ball sort` (à ne pas confondre avec le bubble sort) est un jeu mobile de type casse-tête où un joueur doit réorganiser des balles contenues dans des tubes de façon à ce que chaque tube non vide contienne toutes les balles d'une même couleur et seulement elles. Deux exemples de positions de départ figurent ci-contre.





Les règles de déplacement sont les suivantes :

- On peut déplacer uniquement la balle au sommet d'un tube, et la mettre uniquement dans un tube vide ou dont la balle au sommet est de la même couleur mais ayant au moins une place libre.
- Si on déplace une balle et qu'il existe juste en-dessous de cette balle une balle de la même couleur, elle est déplacée avec (sous réserve qu'il reste suffisamment de place dans le tube d'arrivée), et ainsi de suite. Ainsi, dans l'exemple de droite, si on déplace la balle verte du tube en haut à droite vers un tube vide, celle du dessous est déplacée avec.

On se limite dans cette partie aux règles classiques : quatre balles par couleur, quatre places par tube, toutes les balles sont visibles, il y a deux tubes vides pour les déplacements. On peut simplifier avec la variante suivante : si on ne peut pas déplacer toutes les balles d'une couleur d'un tube vers l'autre, le déplacement est interdit (cf. remarque sur la subtilité ci-après).

La modélisation du jeu se fait ainsi : un jeu est un tableau de tableaux d'entiers, représenté en pratique par un pointeur de pointeurs, et une variable annexe précise le nombre de tableaux. Tout tableau qui n'est pas plein contient un `-1` après la dernière balle (ou en position `0` s'il est vide), et les balles sont représentées par les premiers entiers naturels (une par couleur, et le nombre de couleurs se déduit du nombre de tableaux, comme annoncé ci-avant).

Exercice P2.1 : Créer une variable correspondant à l'exemple **de droite**, en énonçant les tubes depuis leur fond (indice zéro), l'ordre des tubes étant de gauche à droite et toute la rangée du haut étant énoncée avant la rangée du bas. Les couleurs sont à numéroter à partir de zéro dans l'ordre de leur première apparition.

Exercice P2.2 : Écrire une fonction de prototype `bool fini(int** plateau, int nbtubes)` déterminant si le jeu est terminé.

Exercice P2.3 : Écrire une fonction de prototype `int** copie_plateau(int** plateau, int nbtubes)` renvoyant une copie du plateau en argument.

Exercice P2.4 : Écrire une fonction de prototype `int* coups_possibles(int** plateau, int nbtubes)` renvoyant un tableau d'entiers organisé de la manière suivante : en premier indice figure le nombre de coups possibles, ensuite chaque coup est donné dans trois indices consécutifs, avec dans l'ordre le tube de départ, le nombre de balles à déplacer et le tube d'arrivée. En déduire une fonction de prototype `bool perdu(int** plateau, int nbtubes)` déterminant si le joueur est bloqué.

Attention à exclure les coups transférant l'intégralité d'un tube dans un autre, sinon la résolution dans l'exercice P2.6 aura des boucles / récursions infinies ! Une autre subtilité est encore plus technique et n'entrera pas dans l'évaluation...

Exercice P2.5 : Écrire une fonction de prototype `void imprime_coup(int depart, int nbballes, int arrivee)` imprimant un message précisant l'action en argument de manière claire.

Exercice P2.6 : Écrire une fonction de prototype `int* resout(int** plateau, int nbtubes)` renvoyant une solution au jeu si elle existe (sinon `return NULL;`) sous la forme d'un tableau d'entiers organisé de la manière suivante : en premier indice figure le nombre de coups à faire, ensuite chaque coup est donné dans trois indices consécutifs comme dans l'exercice P2.4. En déduire une fonction de prototype `void imprime_solution(int** plateau, int nbtubes)` imprimant la solution trouvée s'il y en a une.

Problème 3 - Two truths one lie [intégralement en OCaml]

Activité pédagogique (fondée sur un jeu de pyjama party) utilisée en début d'année pour faire pratiquer une langue tout en apprenant à se connaître dans une classe, "Two truths one lie" consiste à annoncer trois faits sur soi-même, dont seulement deux sont des vérités et un autre (pas forcément le troisième) est un mensonge.

Ce contexte est l'occasion idéale pour étudier des 3-DNF ! Nous allons certes nous éloigner du principe de l'activité...

Une formule 2T1L est une formule utilisant trois variables propositionnelles différentes que l'on notera a, b et c, écrite sous forme de 3-DNF avec trois « clauses conjonctives », chacune ayant un seul littéral négatif qui est différent pour chaque clause conjonctive.

Exercice P3.1 : Écrire une formule (en tant que formule logique, pas sous forme d'une valeur en OCamll) 2T1L (unique à l'ordre près des littéraux et des clauses conjonctives) pour les variables a, b et c.

Exercice P3.2 : Écrire une variable correspondant à la formule de l'exercice P3.1, avec le type `(string * bool) array` pour les clauses conjonctives (le nom de chaque variable et la précision du fait qu'elle apparaisse normale ou niée) et une formule 2T1L étant représentée comme un tableau de telles clauses conjonctives.

Exercice P3.3 : On considère $n > 3$ variables propositionnelles. Combien de formules 2T1L faut-il au minimum pour qu'une seule interprétation puisse satisfaire la conjonction de toutes ces formules ?

Exercice P3.4 : Écrire une fonction prenant en argument une formule du type des formules 2T1L et déterminant s'il s'agit bien d'une 2T1L.

Exercice P3.5 : Écrire une fonction prenant en argument une formule 2T1L (pas à vérifier) et renvoyant la liste des noms des variables y apparaissant.

Exercice P3.6 : Écrire une fonction prenant en argument une liste de formules 2T1L et cherchant à satisfaire l'ensemble des formules y apparaissant. Si c'est possible, en considérant une interprétation arbitraire parmi celles qui satisfont l'ensemble (donc la conjonction) des formules, on renverra la liste des variables que l'interprétation rend vraies et uniquement elles. Sinon, on déclenchera une erreur avec `failwith`.

Exercice P3.7 : Même exercice mais on déclenchera une erreur si au moins deux interprétations satisfont la conjonction des formules et sinon la valeur de retour est comme dans l'exercice P3.6.

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 la manipulation des chaînes en C, on propose les fonctions suivantes :

- `atoi(chaine)` renvoie l'entier représenté dans la chaîne en argument (ne pas l'utiliser si la chaîne ne correspond pas exactement à un entier), la fonction est `atof` pour renvoyer un flottant de type `double` ;
- `sprintf(chaine, "%d", n)` écrit dans la chaîne en premier argument l'entier en troisième argument, ce qui écrase la chaîne et suppose que sa taille soit suffisante, le format devient `"%f"` pour un flottant ;
- `strcat(chaine1, chaine2)` écrit à la suite de la chaîne en premier argument la chaîne en deuxième argument (même remarque sur la taille) ;
- `strcpy(chaine1, chaine2)` écrit au début de la chaîne en premier argument la chaîne en deuxième argument (idem).

Quelques codes ASCII utiles : 32 pour l'espace, 46 pour le point, 65 pour le A, puis l'alphabet dans l'ordre jusqu'à 90 pour le Z, 97 pour le a puis l'alphabet dans l'ordre jusqu'à 122 pour le z.

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 $c_n = \Theta(n^\alpha \log_b(n))$.