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 (réminiscence DS 1) : Donner la définition d'un type somme et d'un type enregistrement, exemples à l'appui.

Question de cours 2 (réminiscence DS 2) : Expliquer le rôle de `>`, et la différence avec `»`, **dans une commande écrite au niveau d'un terminal Unix**.

Question de cours 3 (réminiscence DS 3) : Donner la définition d'un type option, exemple à l'appui (ne pas écrire de long programme).

Question de cours 4 (réminiscence DS 4) : Donner la définition de l'ordre structurel, exemple à l'appui (dans un langage précis ou non, au choix).

Question de cours 5 (réminiscence DS 5) : Montrer que dans un graphe non orienté non vide, n'importe laquelle des trois propriétés suivantes est impliquée par la conjonction des deux autres : (i) le graphe n'admet pas de cycle non trivial, (ii) le graphe est connexe, (iii) le graphe a une arête de moins que le nombre de sommets.

Question de cours 6 : Démontrer le théorème d'interpolation de Craig.

Question de cours 7 : Démontrer le théorème de substitution.

Question de cours 8 : Montrer que toute formule de la logique propositionnelle est équivalente à une formule en forme normale disjonctive (la syntaxe de la logique propositionnelle est au choix pour la preuve, si tant est qu'on veut en tenir compte).

Question de cours 9 : La formule $(a \vee b) \wedge (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c)$ est-elle satisfaisable ? Si oui, donner une interprétation qui la satisfait et déterminer s'il existe une clause distincte des six déjà présentes et qui rendrait la formule insatisfaisable en l'ajoutant. Si non, le prouver et déterminer si elle devient satisfaisable en retirant une clause.

Question de cours 10 : Définir les notions de clé, de clé primaire et de clé étrangère.

Question de cours 11 : Dans une requête sans sous-requête utilisant les mots-clés `FROM`, `GROUP BY`, `HAVING`, `JOIN`, `SELECT` et `WHERE` en SQL, dans quel ordre apparaissent ces mots-clés ?

Problème 1 - Bases de données et relations sur un ensemble

Nous allons faire le lien entre la théorie des ensembles et les bases de données.

On considère un ensemble fini D et une table `T` avec un attribut noté `Elt` dont les enregistrements parcourent l'ensemble D. On considère aussi une relation R entre éléments de D, qu'on représentera par une fonction $f : D \times D \to \{0, 1\}$ telle que $f(x, y) = 1$ ssi xRy. La fonction f est l'indicatrice de la relation, et on suppose qu'elle est implémentée en SQL de sorte qu'un appel à `f(x,y)` renvoie `1` ou `0` suivant le cas, dès lors que `x` et `y` seront des valeurs pour l'attribut `Elt`.

Exercice P1.1 : Écrire une requête dont le résultat est la table dérivée décrivant la relation R.

On suppose maintenant que la table qu'on souhaitait obtenir existe déjà et s'appelle `TT`. Comme la relation n'est pas forcément symétrique, l'ordre des attributs est important, et on les appellera `Elt1` et `Elt2`, tous les deux de domaine D.

Exercice P1.2 : Écrire une requête qui permet de récupérer le nombre d'éléments y de D tels que xRy, à x fixé et utilisable dans la requête.

Exercice P1.3 : Écrire une requête qui permet de récupérer le nombre d'éléments y de D tels que yRy.

Exercice P1.4 : Écrire une requête qui permet de récupérer le nombre d'éléments y de D tels que xRy ou yRx, à x fixé et utilisable dans la requête. Attention, on demande bien Card$(\{y \in D \mid xRy \vee yRx\})$.

Problème 2 - Bases de données et gestion de vos notes

Ce problème a été inspiré par un souci que j'ai eu cette année en tentant de saisir des notes : les identifiants étaient incohérents (une sombre histoire d'accents dans un fichier et absents d'un autre). Une requête rapide m'a permis de tout arranger, et le reste sera justifié par l'intérêt pédagogique.

Sur mon serveur, deux tables régissent la gestion des notes pour mes classes, mais le schéma ci-après ne présente que les attributs qui seront utiles (si le programme était plus étendu, on pourrait s'amuser sur les caractères composant les mots de passe, par exemple...). Une table existe aussi pour les rendus et elle est également liée à la table des comptes, mais elle ne nous intéressera pas cette fois-ci.

- Table `Comptes`, avec pour attributs `Login` (l'identifiant qui est saisi pour se connecter, typiquement), `Nom` (avec le prénom) et `Classe` (sous forme de chaînes de caractères).
- Table `Notes`, avec pour attributs `Etudiant` (le login), `Date` (chaîne de caractères en clair au format `AAAA-MM-JJ` pour respecter l'ordre chronologique), `Titre_DS` (chaîne de caractères) et `Note` (flottant, en particulier en cas d'absence il n'y a pas d'enregistrement correspondant).

Bon, en pratique l'attribut `Note` est une chaîne de caractères, mais ce serait pénible de voir qu'un 5 est meilleur qu'un 19, heureusement, aucun tri n'est effectivement fait avec ces données, et quoi qu'il en soit pas par SQL...

Les exercices suivants commencent tous par `Écrire une requête en SQL permettant d'obtenir....`

Exercice P2.1 : ... la date de la première épreuve.

Exercice P2.2 : ... le nombre de classes comportant au moins un étudiant ayant un compte.

Exercice P2.3 : ... la meilleure note d'un étudiant dont l'identité est donnée (écrire `id` pour la valeur en question).

Exercice P2.4 : ... la liste des enregistrements correspondant à une note qui aurait été associée à un étudiant dont le nom de compte était mal écrit (autrement dit à un compte inexistant) [ceci est la question qui a justifié l'écriture du problème].

Exercice P2.5 : ... le nombre de notes de chaque étudiant ayant un compte (on donnera également l'identifiant de l'étudiant pour clarifier). **Attention, il est difficile de trouver la réponse correcte sans bien réfléchir ici !**

Exercice P2.6 : ... les éventuelles paires d'épreuves ayant eu lieu le même jour, indépendamment de la classe.

Pour l'exercice précédent, on ne mettra qu'une fois les paires éventuelles (donc pas DS 1 en face de DM 2 puis DM 2 en face de DS 1). Hypothèse simplificatrice : les épreuves de deux classes différentes ont des noms différents et ceci ne fait en particulier pas manquer de paires en raison d'une quelconque homonymie.

Exercice P2.7 : ... le nombre d'étudiants de chaque classe (la classe est également à préciser), en tant qu'attribut renommé `Effectif` et par effectif décroissant (égalités gérées comme on souhaite).

Exercice P2.8 : ... la pire assiduité (en pourcentage) d'une épreuve concernant des étudiants de PC (on admet que l'effectif est constant sur l'année).

Problème 3 - Autour du code Gray

Toutes les fonctions de ce problème sont à écrire en C.

La représentation des entiers en binaire se fera par des tableaux en *little endian*, ce qui signifie que le chiffre des unités sera le plus à droite, conformément à l'écriture usuelle que nous employons mais différent des conventions suggérées précédemment cette année pour les représentations de polynômes, par exemple.

Ce problème s'inspire du sujet d'option informatique de Centrale en 2019 (sans la mutation des listes en OCaml à propos de laquelle mieux vaut ne pas faire de commentaire) et présente le code Gray. Il s'agit d'une méthode d'énumération des entiers naturels, ici représentés en binaire en tant que tableaux dont les éléments seront tous zéro ou un, limitant les états transitoires lors du passage d'un entier à l'autre. Pour clarifier, en passant de la représentation binaire de 3 à celle de 4, trois bits changent de valeur, et si ce n'est pas simultané, d'autres entiers sont représentés pendant une courte période, ce qui peut avoir des conséquences néfastes du point de vue électronique.

Exercice P3.1 : Écrire une fonction prenant en argument un tableau d'entiers valant tous `0` ou `1` et sa taille et retournant le tableau correspondant à la représentation en binaire de l'entier représenté par le tableau en argument plus un. Si le tableau en argument ne contient que des 1, on déclenchera une erreur par `exit(1)`. **Il est interdit de passer par l'entier représenté !**

Exercice P3.2 : Déterminer la somme des nombres de bits différents entre i et $i+1$ écrits en binaire pour i de 0 à $2^n - 2$.

La méthode qui semble la plus populaire pour produire tous les codes binaires sur n bits selon l'ordre du code Gray revient à partir d'un tableau de deux tableaux de taille un, contenant respectivement `0` et `1`, et à chaque étape de prendre tous les éléments du tableau de tableaux actuel précédés d'un zéro puis tous les éléments du tableau de tableaux actuel précédés d'un un et en commençant par le dernier (donc le dernier indice du tableau de tableaux, on ne change pas l'ordre des codes au sein des tableaux). On remarquera que l'ordre des entiers représentés demeure en particulier le même en passant d'une taille à la suivante. **Il est garanti que la valeur de `n` sera strictement positive, inutile de le vérifier.**

Exercice P3.3 : Lister les seize premiers entiers naturels selon l'ordre indiqué.

Exercice P3.4 : Écrire une fonction renvoyant tous les codes binaires sur n bits dans l'ordre engendrés par la méthode précédente. On renvera un tableau de tableaux d'entiers.

Exercice P3.5 : Prouver que la méthode renvoie effectivement un tableau contenant une et une seule fois tous les tableaux de n bits possibles et que passer de l'un à l'autre se fait en changeant exactement un bit.

Exercice P3.6 : Prouver que le dernier élément du tableau est la représentation en binaire de 2^{n-1}, où n correspond au nombre de bits de chaque élément.

Une deuxième méthode (je n'ai en pratique appliqué que celle-ci pendant des années) revient à partir d'un tableau composé uniquement de zéros et pour chaque indice ultérieur `i` produire le tableau obtenu à partir du tableau d'indice `i-1` en modifiant le bit à la position qui est la valuation dyadique de l'entier `i` (donc le plus grand entier `k` tel que deux à la puissance `k` divise `i`) **en comptant les positions à partir de la fin, le dernier étant à la position** `0`.

Exercice P3.7 : Écrire une fonction prenant en argument un entier strictement positif et renvoyant sa valuation dyadique.

Exercice P3.8 : Écrire une fonction renvoyant tous les codes binaires sur n bits dans l'ordre engendrés par la méthode précédente. On renvera aussi un tableau de tableaux d'entiers.

Exercice P3.9 : Prouver que la méthode renvoie effectivement un tableau contenant une et une seule fois tous les tableaux de n bits possibles et que passer de l'un à l'autre se fait en changeant exactement un bit. Cela peut se faire directement ou en prouvant que les résultats des deux fonctions engendrant les codes binaires sur n bits sont les mêmes.

Exercice P3.10 : Discuter de l'utilité de compter en binaire selon le code Gray, par exemple dans le cadre d'une exploration exhaustive, en considérant le problème de la sous-somme traité récemment.

Problème 4 - Dominosa

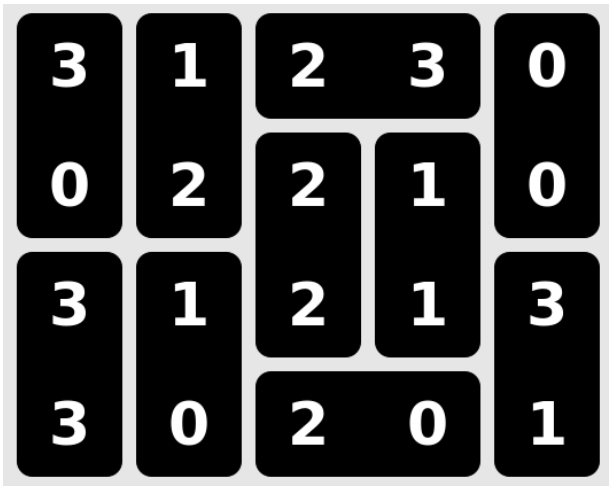
Tous les programmes de ce problème seront à écrire en OCaml.

L'énigme de logique intitulée « Dominosa » (nom trouvé à divers endroits) consiste à reconstituer un set complet de dominos caché dans une grille de nombres. **Les grilles seront toujours rectangulaires mais de dimensions arbitraires et à récupérer à la main dans les programmes.**

On définit un set complet de dominos allant jusqu'à n comme l'ensemble des couples (i, j) pour lesquels $0 \leq i \leq j \leq n$. Cacher un tel set dans une grille revient à découper cette grille en rectangles de longueur deux cases et de largeur une case (orientés horizontalement ou verticalement) de sorte que chaque domino soit représenté une et une seule fois en tant que rectangle. **Toutes les complexités à calculer dans ce problème seront exprimées en fonction de cette valeur n.** On admettra que n sera positif et que les grilles auront au moins une ligne et au moins une colonne.

Un exemple de grille et sa solution (allant jusqu'à 3) figure ci-après.

3	1	2	3	0
0	2	2	1	0
3	1	2	1	3
3	0	2	0	1



Exercice P4.1 : Combien y a-t-il de cases dans une grille allant jusqu'à n ?

Exercice P4.2 : Écrire une fonction `n_grille` prenant en argument le nombre de lignes et le nombre de colonnes d'une grille de Dominosa et qui renvoie la valeur n associée (en particulier, la grille ne sera pas en argument et la complexité doit être constante). Cette fonction sera à appeler par la suite pour ne pas que son résultat soit un argument superflu des fonctions à écrire.

Comme pour toutes les énigmes de logique, des raisonnements peuvent être mis en place pour ne pas recourir à la seule force brute. Nous allons voir comment mettre ces raisonnements en œuvre à la main, mais aussi et surtout en termes de programmation.

Dans un premier temps, on peut chercher un démarrage qui se voit particulièrement bien : les dominos avec deux fois la même valeur. Puisqu'il faut pouvoir les placer, cela veut dire qu'au moins une fois sur la grille deux voisins sont égaux. Idéalement, cela ne se produit qu'une fois et un domino est trouvé.

Exercice P4.3 : Écrire une fonction qui prend en argument une grille de Dominosa ainsi qu'un entier `k` et qui détermine si le domino « double `k` » peut être placé selon le principe ci-avant. La valeur de retour sera un triplet `(i, j, b)` formé de deux entiers et d'un booléen de sorte que le domino « double `k` » soit sur les cases `(i, j)` et sa voisine, de droite si `b` est vrai et du bas sinon. Cependant, si le domino « double `k` » ne peut pas être placé directement en raison d'un doute possible, la valeur de retour sera `(-1, -1, false)`. Donner également la complexité de cette fonction.

Exercice P4.4 : Signaler comment adapter la fonction précédente afin de déterminer si n'importe quel domino (deux entiers en argument, dont on peut supposer que le premier est inférieur ou égal au deuxième pour simplifier) peut être placé de manière certaine dans une grille.

À présent, il est temps de chercher une structure de données suffisamment avancée pour supporter les raisonnements ultérieurs.

En parallèle de la grille de nombres, une matrice de booléens de même dimensions signalera si une case a déjà été identifiée comme faisant partie d'un domino (le booléen sera alors évidemment `true`), et ne sera donc plus à considérer comme la voisine d'une autre dans le cadre de l'exploration. On appellera cette matrice *progression*.

Une autre matrice précise à chaque ligne `i` inférieure ou égale à `n` et à chaque colonne `j` inférieure ou égale à `i` si le domino `(j, i)` a déjà été placé et si oui où (représentation au choix, pourvu qu'elle soit pertinente). On appellera cette matrice *checklist*.

Exercice P4.5 : Écrire une fonction qui prend en argument une grille de Dominosa et qui fait faire tous les tests induits par la fonction de l'exercice précédent une seule fois, pour le moment sans tenir compte des dominos trouvés (sinon il faudrait adapter la fonction précédente). On renverra l'état actuel des choses en tant que le couple formé par la progression et la checklist. Donner aussi la complexité de cette fonction.

Exercice P4.6 : Écrire une fonction qui prend en argument une grille de Dominosa ainsi que la progression et la checklist, et qui construit une matrice de dictionnaires (là aussi, un autre choix était possible) de mêmes dimensions que la grille, de sorte qu'à chaque position où le booléen de la progression est `true`, le dictionnaire sera vide, et à chaque autre position (notée `(i, j)`), le dictionnaire contiendra comme clés l'ensemble des valeurs des cases voisines de la case `(i, j)` pour lesquelles le booléen de la progression est aussi `false`, à condition que cette case et la case `(i, j)` forment un couple de valeurs représentant un élément de la checklist qui est encore à `false` ; les valeurs associées à ces clés sont la liste des cases voisines pour lesquelles le booléen de la progression est `false` et qui contiennent la valeur en question.

La matrice de dictionnaires permettant de déduire la progression, elle la remplacera désormais et s'appellera *avancée*. Le choix d'utiliser la liste des cases voisines plutôt que leur nombre est dicté par des algorithmes plus avancés à écrire ultérieurement : les mises à jour de l'avancée élimineront parfois une liaison sans placer de domino.

Exercice P4.7 : Écrire une fonction qui prend en argument une grille de Dominosa ainsi que l'avancée et la checklist obtenues par la fonction précédente et qui tente de placer tous les dominos une fois à la lumière de toutes les informations disponibles. À chaque fois qu'on place un domino, les structures annexes sont à modifier en fonction. On se posera la question de la valeur de retour au vu de la suite de l'énoncé.

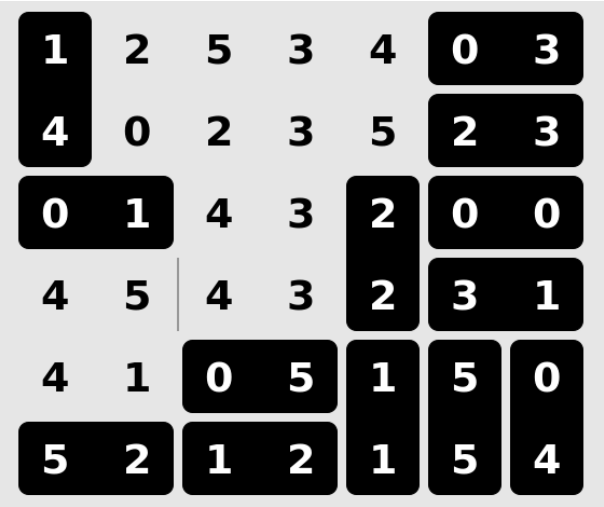
On pourra aussi commencer par énoncer les règles de raisonnement utilisées, cela facilitera l'écriture (et surtout la lecture) des programmes.

Exercice P4.8 : Écrire une fonction qui répète la fonction précédente jusqu'à ce que plus rien ne soit trouvé par ces moyens (éventuellement parce que la grille est résolue). Donner une majoration pertinente de la complexité de cette fonction.

Un raisonnement particulièrement élégant peut être mis en œuvre dans le jeu de Dominosa : si par hasard on se retrouvait dans une situation où placer un domino augmenterait d'un le nombre de « composantes connexes » des cases libres de la grille, ce domino devrait laisser invariante cette propriété sur toute grille partiellement remplie : le nombre de cases de chaque composante connexes de cases libres doit être pair.

Exercice P4.9 : Mettre en œuvre ce raisonnement sous la forme d'une fonction prenant en argument une grille de Dominosa, l'avancée et la checklist et tentant de mettant à jour l'avancée à la lumière du raisonnement ci-avant. La fonction renverra un booléen déterminant si une modification a été faite. On pourra en profiter pour faire le lien avec la théorie des graphes.

Illustration pour la question précédente (noter qu'un autre raisonnement pouvait s'appliquer à ce stade, mais chercher plus longtemps une instance où ce fait était crucial aurait demandé plus de temps) :



Ici, une barre a été placée pour signifier que sur le 4 et le 5 concernés il était impossible de placer un domino.

Exercice P4.10 : Après l'application des fonctions précédentes, il est probable que la grille ne soit toujours pas résolue. Proposer des solutions, par exemple à base de raisonnements qui peuvent s'implémenter en des algorithmes. [Ceci est une question ouverte donc le barème n'est pas fixé et les points seront accordés à la mesure des contributions.]

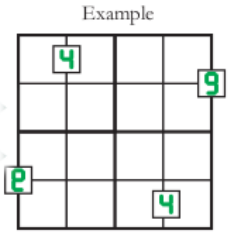
Exercice P4.11 : On termine par une grille à résoudre à la main, cela aurait été inconcevable de ne pas en faire une! Détailler le raisonnement au maximum pour avoir tous les points (inutile de faire des phrases tout le long, pour autant).

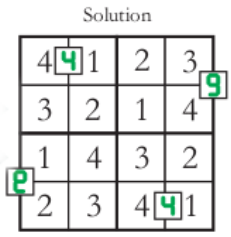
Pour être sûr que personne ne termine le DS dans les temps : un sudoku

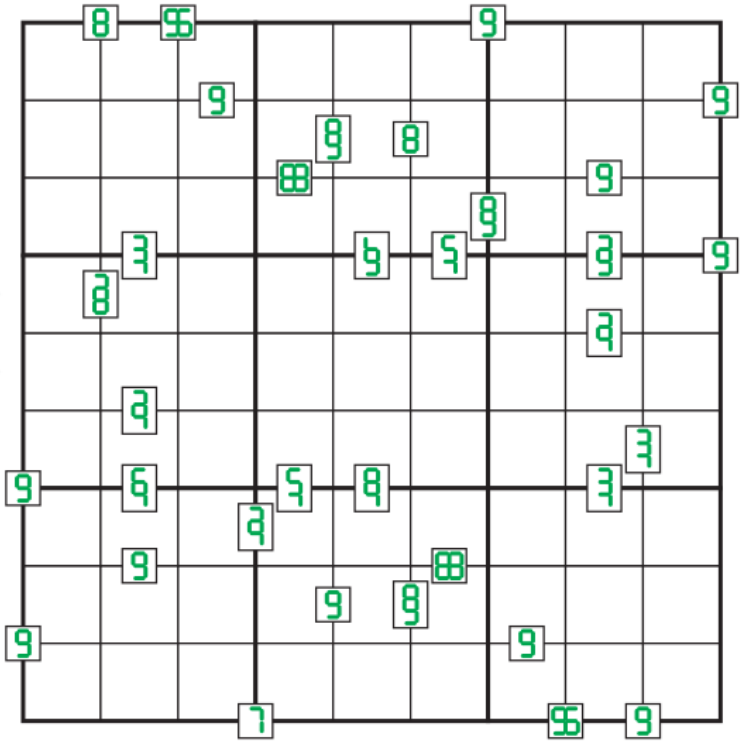
Le sudoku ci-après est tiré d'un concours de puzzle (avril 2024, auteur Riad Khanmagomedov). Le temps de résolution se compte en heures.

En plus de la règle usuelle du sudoku (s'il faut la rappeler : placer dans chaque ligne, chaque colonne et chaque bloc de trois cases par trois délimité par un bord en gras) une occurrence et une seule de chaque chiffre de 1 à 9 pour la taille ici considérée.

Ce sudoku a pour particularité que des rectangles figurent entre deux cases (parfois sur le bord du sudoku pour limiter l'encombrement) et le contenu du rectangle est une superposition des chiffres des deux cases, sans rotation ni symétrie, mais éventuellement translatés l'un par rapport à l'autre. Un exemple est donné à côté de la grille. L'affichage des chiffres est rappelé au-dessus de la grille.







Dans la mesure où une résolution complète est inimaginable dans les temps alloués pour le DS, c'est surtout la présentation des raisonnements qui sera évaluée et valorisée. En tout état de cause, il est peu probable que s'attaquer à cet exercice soit rentable en termes de points. Compléter la grille avant la correction du DS fera cependant office de dernière quête secondaire.

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).

Théorème de Craig : Soient n un entier naturel non nul, $p_1, p_2, \ldots, p_n$ des variables propositionnelles deux à deux distinctes, et φ, ψ deux formules ayant $p_1, p_2, \ldots, p_n$ comme variables propositionnelles communes. Les deux propriétés suivantes sont équivalentes : (i) La formule $(\varphi \Rightarrow \psi)$ est une tautologie. (ii) Il existe au moins une formule θ, ne contenant aucune variable propositionnelle en dehors de $p_1, \ldots, p_n$, appelée interpolante entre φ et ψ, et telle que les formules $\varphi \Rightarrow \theta$ et $\theta \Rightarrow \psi$ soient des tautologies.

Théorème de substitution : Soient I une interprétation, n un entier naturel, $\varphi, \psi_1, \ldots, \psi_n$ des formules et $p_1, \ldots, p_n$ des variables propositionnelles deux à deux distinctes et n'apparaissant dans aucune des formules ψ_k. La valeur de vérité de $\varphi[p_1/\psi_1, \ldots, p_n/\psi_n]$ avec l'interprétation I est la même que la valeur de vérité de φ avec une interprétation qui correspond à I pour les variables propositionnelles hors $p_1, \ldots, p_n$ et qui affecte à chaque p_i la valeur de vérité avec I du ψ_i correspondant.

Pour la troisième fois de suite, un taux de succès supérieur à un seuil fixé dans les questions de cours sera aussi gratifiant que l'inscription au tableau des side quests. Le seuil dépendra de la réussite globale de la classe mais il serait agréable de voir une demi-douzaine de personnes au-delà des trois quarts des points.