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 : Donner, en détaillant les étapes, la liste des codes émis par l'algorithme LZW sur la chaîne de caractères suivante : "never gonna give you up never gonna let you down never gonna run around and desert you never gonna make you cry never gonna say goodbye never gonna tell a lie and hurt you" (sans retour à la ligne, seulement des espaces). Les codes ASCII utiles sont en annexe.

Question de cours 2 : Énoncer le théorème de lecture unique pour la syntaxe usuelle de la logique propositionnelle.

Question de cours 3 : Définir la notion de formule satisfaisable.

Question de cours 4 : Prouver le théorème de substitution, dont l'énoncé est rappelé en annexe.

Question de cours 5 : Prouver que $(\Rightarrow, \perp)$ et (NAND) sont des systèmes complets. On pourra partir du fait que $(\neg, \vee)$ et $(\neg, \wedge)$ en sont, sans démontrer ce dernier résultat.

Question de cours 6 : Prouver que, pour deux formules propositionnelles φ et ψ quelconques, $(\varphi \Rightarrow \psi) \vee (\psi \Rightarrow \varphi)$ est une tautologie et que $(\varphi \Rightarrow \psi) \Rightarrow \varphi$ implique φ .

Question de cours 7 : Définir le problème SAT.

Question de cours 8 : Définir les opérateurs relationnels de sélection et de projection.

Question de cours 9 : Donner la différence entre une association 1 – 1, une association 1 – * et une association * – *.

Question de cours 10 : Donner la différence entre **WHERE** et **HAVING**.

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 le tableau des positions où le motif est trouvé dans l'autre chaîne.

Exercice T2 : Écrire en OCaml une fonction prenant en argument une chaîne de caractères et construisant une séquence (type au choix) de couples formés de chaque caractère et de son nombre d'occurrences. L'ordre n'a pas d'importance.

Exercice T3 : En utilisant les types ci-après, écrire en OCaml la fonction `evalue v arbre` qui évalue la formule propositionnelle représentée par `arbre` avec le type défini ci-avant, suivant l'interprétation `v`. Écrire aussi une fonction d'évaluation pour le type `f_b`. Dans les deux cas, les variables utilisées dans les formules seront les premiers entiers naturels et les interprétations seront des tableaux de booléens.

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

type feuille_formule = V | F | Vb of int;;
type operateur = And | Or | Not;;
type arbre_formule = Feuille of feuille_formule | Noeud of operateur * arbre_formule list;;
```

Problème 0 - Boyer - Karp - Moore - Rabin [intégralement en C]

Pour la beauté du geste, mélangeons les algorithmes de Boyer-Moore et de Rabin-Karp (et le choix du langage permet de ne pas faire un DS avec à peine plus de programmation en C qu'il n'y aura eu de programmation en OCaml aux Mines cette année...).

Le principe sera de comparer le haché d'une tranche avec celui du motif, comme pour Rabin-Karp, et si les hachés sont identiques, en raison de l'ambiguïté possible via les opérations de modulo, de comparer la tranche au motif en commençant par la droite, comme pour Boyer-Moore.

En cas d'échec de la comparaison des hachés, on décale d'un cran. En cas d'échec durant la comparaison entre la tranche et le motif, on décale d'autant de crans que la table de redirection l'indique (dans la version simple de Boyer-Moore). Cela implique de calculer successivement le passage au haché suivant pour toutes les étapes intermédiaires.

Puisque l'on programme en C, il ne s'agira pas d'utiliser des dictionnaires mais des tableaux de taille 256 contenant chaque caractère de la table ASCII. Cela simplifiera les choses au prix de la complexité en temps et en espace.

Exercice P0.1 : Écrire une fonction prenant en argument un motif et renvoyant le tableau des tableaux de redirection.

Exercice P0.2 : Écrire une fonction prenant en argument un motif et renvoyant son haché. Il s'agit de simuler la fonction suivante, en notant $m_0, m_1, \dots, m_{k-1}$ les caractères successifs du motif : $\sum_{i=0}^{k-1} m_i 31^{k-1-i}$, le tout modulo 97 (on calculera les restes des divisions euclidiennes à chaque fois qu'il y a un risque de dépassement arithmétique). En vue d'éviter le copier-coller, la fonction aura un argument supplémentaire qui est le nombre de caractères à considérer en vue du calcul du haché, ce qui sera utile pour la première tranche du texte.

Exercice P0.3 : Écrire avec les arguments nécessaires la fonction de passage du haché (selon la fonction de l'exercice précédent) d'une tranche du texte au haché de la tranche suivante. Tout ce qui est cohérent avec la fonction finale peut être accepté.

Exercice P0.4 : Écrire une fonction réalisant la combinaison entre l'algorithme de Boyer-Moore et l'algorithme de Rabin-Karp mentionnée en début de problème. La fonction de hachage à utiliser est la même que précédemment. **On renverra le premier indice du motif dans le texte**, avec la valeur par défaut de -1 si le motif n'est pas présent dans le texte.

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

La base de données utilisée pour ce problème est :

- Table PARTICIPANTS, avec les attributs `id_participant`, `identite` et `genre`.
- Table OBJETS, avec les attributs `id_objet`, `categorie` et `prix`.
- Table CADEAUX, avec les attributs `participant`, `destinataire`, `cadeau` et `satisfaction`.

L'attribut `categorie` de la table OBJETS permet de rassembler des objets similaires mais de nom différent. L'attribut `satisfaction` est un booléen précisant si le destinataire était content de son cadeau. Le reste est transparent.

Exercice P1.1 : Écrire une requête permettant d'obtenir le prix le plus élevé d'un objet disponible.

Exercice P1.2 : Écrire une requête permettant d'obtenir le prix moyen des cadeaux effectivement offerts (en comptant un objet offert plusieurs fois à concurrence du nombre de fois qu'il aurait été offert).

Exercice P1.3 : Écrire une requête permettant d'obtenir le prix minimal d'un objet ayant plu à au moins une personne.

Exercice P1.4 : Écrire une requête permettant d'obtenir la catégorie d'objets pour laquelle le prix total des cadeaux est le plus élevé (même remarque que pour l'exercice P1.2, et une seule suffit en cas d'égalité).

Exercice P1.5 : Écrire une requête permettant d'obtenir le nombre de fois qu'une fille a offert un cadeau à un garçon (les valeurs possibles de `genre` sont "M" et "F").

Exercice P1.6 : Écrire une requête permettant d'obtenir le nombre de fois qu'une personne a offert un cadeau de la même catégorie que celui qu'elle a reçu.

Exercice P1.7 : Écrire une requête permettant d'obtenir la plus grande différence de prix entre le cadeau offert et le cadeau reçu sous la forme de deux attributs (plus grande différence entre reçu et offert, et plus grande différence entre offert et reçu).

Problème 2 - Spectacles et satisfaisabilité

Mise en situation de la logique dans la vie réelle : on considère un ensemble de spectacles noté $\mathcal{S}$, dont les éléments sont représentés comme ensembles de dates où les spectacles ont lieu et notés $S_1, S_2, \dots, S_n$. Les dates sont représentées comme des entiers naturels, à comprendre comme le nombre de jours à partir du premier jour de la période considérée.

Mon but est d'assister à tous les spectacles, mais un conflit de date est possible. Ainsi, à une certaine date, je peux me rendre à un spectacle au plus. Ceci étant, une fois que j'ai vu un spectacle, je n'y retourne pas.

On considère que les dates possibles sont les entiers de 1 à k inclus, et on note k_i le nombre de dates auxquelles le spectacle S_i est joué, ces dates étant notées $s_{i,1}, \dots, s_{i,k_i}$. Pour un numéro de spectacle i (avec $1 \leq i \leq n$) et une date d (avec $1 \leq d \leq k$) appartenant à S_i , on considère la variable $v_{i,d}$ signifiant que je suis allé au spectacle i à la date d .

Exercice P2.1 : Écrire une clause qui est vraie si, et seulement si, je suis allé voir le spectacle numéro i au moins une fois.

Exercice P2.2 : Écrire une formule propositionnelle en forme normale conjonctive qui est vraie si, et seulement si, je suis allé voir au moins un spectacle à la date d , puis une formule propositionnelle en forme normale conjonctive qui est vraie si, et seulement si, je suis allé voir au plus un spectacle à la date d .

Exercice P2.3 : Écrire une formule propositionnelle en forme normale conjonctive qui est satisfaisable si, et seulement si, je suis allé voir tous les spectacles exactement une fois sans conflit de date.

Exercice P2.4 : Rappeler comment s'appelle le problème qui ressemblerait à une modélisation de la question du nombre maximal de spectacles que je peux voir sans conflit de date.

Problème 3 - Mosaic [intégralement en OCaml]

La collection d'énigmes `sgtpuzzles` s'est enrichie depuis environ un an d'un nouveau jeu auquel j'ai accroché, intitulé Mosaic. Le principe est de colorier en noir et blanc une grille dont certaines cases sont marquées d'un indice correspondant au nombre de cases en noir dans le voisinage comprenant la case marquée de l'indice, ses jusqu'à quatre voisines latérales et les jusqu'à quatre voisines en diagonale.

Ci-dessous un exemple et sa solution.

2		4	2	
5		9		5
			6	
	4	4		

2	■	4	2	□
□	■	■	■	□
5	■	9	■	5
■	■	■	6	■
□	4	4	□	□

Comme on peut le constater, la case marquée de l'indice 2 dans le coin en haut à gauche a deux voisines noires à droite et en bas à droite, la case centrale marquée de l'indice 9 est noire comme ses huit voisines, etc.

Le but de ce problème est de procéder à des raisonnements sur ce jeu de logique et d'implémenter des fonctions de résolution.

Exercice P3.1 : Supposons qu'une case n'ait aucune case avec indice dans son voisinage. Peut-il y avoir unicité de la solution ? (Si oui, donner un exemple ; si non, le prouver.)

Exercice P3.2 : Supposons que toutes les cases soient marquées d'un indice. Y a-t-il forcément unicité de la solution ? (Si oui, le prouver ; si non, donner un contre-exemple.)

Exercice P3.3 : Supposons qu'aucune case ne soit marquée d'un 0 ni du nombre de cases du voisinage (donc 4 pour un coin, 6 pour un bord ou 9 ailleurs). Peut-il y avoir unicité de la solution ? (Si oui, donner un exemple ; si non, le prouver.)

Bien entendu, la résolution est plus simple à démarrer avec des cases marquées d'un 0 où du nombre de cases du voisinage, que ce soit pour l'humain ou pour les algorithmes que nous allons mettre en œuvre.

La grille de jeu est représentée dans ce problème par un tableau de tableaux d'entiers, avec la valeur `-1` pour signaler qu'il n'y a pas d'indice.

Exercice P3.4 : Écrire une fonction `demarrage` prenant en argument une grille de jeu et renvoyant la liste des indices des cases (en tant que couples formés par la ligne et la colonne) où l'indice est 0 ou le nombre de cases du voisinage.

La résolution implique une autre grille, de mêmes dimensions que la grille de jeu, et dont les valeurs seront 1 pour une case noire, 0 pour une case blanche et `-1` pour une case de couleur non encore déterminée.

Mis à part les cases détectées par la fonction `demarrage`, on peut faire progresser la résolution au niveau des cases ayant un indice noté k , si une telle case a dans la grille de résolution déjà k cases noires dans le voisinage (et au moins une case de couleur non encore déterminée), ou si elle a dans la grille de résolution déjà $n - k$ cases blanches dans le voisinage, où n est le nombre de cases du voisinage (avec la même restriction). Ceci peut être déterminé en analysant les deux grilles. Dans le premier cas, toutes les cases de couleur non encore déterminée peuvent être mises en blanc, et dans le deuxième cas elles peuvent toutes être mises en noir.

Exercice P3.5 : Écrire une fonction `completable` prenant en argument une grille de jeu et la grille de résolution en cours d'élaboration et renvoyant une liste permettant de mettre à jour la grille de résolution selon le principe ci-avant, la liste contenant des triplets (i, j, b) pour lesquels les **cases indéterminées** du voisinage de la case à la ligne i , colonne j peuvent être remplies par des b .

Exercice P3.6 : Écrire une fonction `miseajour` prenant en argument la grille de résolution en cours d'élaboration et la liste renvoyée par la fonction précédente et mutant cette grille selon le protocole établi.

Exercice P3.7 : Écrire une fonction `resoudre_simple` prenant en argument une grille de jeu et construisant la grille de résolution associée et va aussi loin que possible dans la résolution en ne reprenant que le principe ci-avant. La fonction doit renvoyer la grille de résolution ainsi qu'un booléen précisant si la résolution est terminée.

Exercice P3.8 : Proposer des raisonnements plus élaborés qu'un humain peut mettre en œuvre. Inutile de tenter de les implémenter cependant.

Exercice P3.9 : Proposer d'autres méthodes algorithmiques de résolution.

Exercice P3.10 : Résoudre la grille ci-après à la main (on fera apparaître le détail du raisonnement). La manière de marquer les couleurs est au choix, mais à clarifier. On pourra travailler directement sur l'énoncé et le rendre.

			2	2					
	6	4			6	5	4		2
5		6	4	4		5	5	4	
	7			6					3
		5		6	5	5		4	
0			4				3	3	
0		2			4	4	3		
	5		7		4		2		2
			5			5			3
	4	4		4					3

Exercice P3.11 : Résoudre la grille ci-après à la main (on fera apparaître le détail du raisonnement). Attention, cela peut durer bien plus longtemps.

	1			3	1				1		3			4	4	4				4	4			1
		5	3				2													5				
		4	3	4		6		4		4	4		6		3		5		4		1	4		5
4					4	5		4				6					6	8						
	3		2	1	3	3			3	4	2		3	3	3					5	4	4		
3			2			4		6					3			3	5					4		3
1					5	6	5					5	6		4	2			8		5	3		
		2	1	3		8		5		6		7		4					8		3	2	2	
2			2			7	5			6		7	6		4	4	5		6		4	3		
			3		3			3		4	5		6	7	6		7		7	5				
5			6			2		5			5				9	7	6		5					5
				6				5	6			7		7	9						7	4		3
4		5	6			5	6			6									7	6		5	5	
			5	6	5		5		7		8	5	6					7	7	7				5
	5				5			5			7	6		3		3				4		6		5
3		2		4		2		5			6				5						5			5
			4		5		3		7	7	5	6	4					2	1			7	8	
	2						2			7	4	5	4	5		4	4	2				6	5	
2				4		5				5	4		4	4		3		4	4		5			
3		2	1		4	5		4	2		3					5	6	6			3		4	
				3	4		4	4	3	4			3	2	3		5		5			4		
	3	2					5	6		5			4			5	6	5		2		5		6
	3		5		5		5	5			7	8	6	4		5		4	3		3	5		5
			6							5					4		6	4	4			6		
1		3		5	5		2	3	3			5	5								2	3		4

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.

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