Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml suivant l'exercice. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *L'accès aux enfers*

Dans la mythologie grecque, l'accès aux Enfers est gardé par le Cerbère, un terrible loup à trois têtes. Celui-ci se trouve devant trois couloirs qui, soit permettent de rejoindre le monde des vivants, soit conduisent directement aux Enfers.

Lorsque Cerbère accueille un nouvel arrivant, il est contraint de lui dire la vérité. Par la suite, il peut mentir ou dire la vérité à sa guise mais il respecte toujours les règles qu'il s'est fixées.

Après avoir bu la coupe de ciguë, Socrate se retrouve face à Cerbère. Celui-ci, honoré de rencontrer le grand philosophe, veut lui offrir une chance d'éviter la damnation éternelle. Il lui dit alors : « *Je vais t'indiquer un des couloirs qui mène au monde des vivants mais, pour mettre à l'épreuve ta grande sagesse, j'énoncerai trois indications logiques qui seront, soit toutes vraies, soit toutes fausses et tu en déduiras le couloir que tu devras suivre* ».

Nous noterons I_1 , I_2 et I_3 les propositions associées aux indications de la première, la deuxième et la troisième tête de Cerbère.

Q1– Représenter l'énoncé de Cerbère sous la forme d'une formule du calcul des propositions dépendant de I_1, I_2 et I_3

$(I_1 \wedge I_2 \wedge I_3) \vee (\neg I_1 \wedge \neg I_2 \wedge \neg I_3)$

La première tête dit ensuite : « *Le premier couloir ainsi que le troisième mènent au monde des vivants* ». La deuxième tête dit : « *Si le deuxième couloir mène au monde des vivants, alors le troisième n'y mène pas* ». La troisième tête conclut par : « *Le premier couloir mène au monde des vivants, par contre le deuxième n'y mène pas* ».

Nous noterons, C_1, C_2 et C_3 les variables propositionnelles correspondant au fait que le premier, le deuxième et le troisième couloir mène au monde des vivants.

Q2– Exprimer I_1, I_2 et I_3 sous la forme du calcul des propositions dépendant de C_1, C_2 et C_3

- $I_1 : C_1 \wedge C_3$
- $I_2 : C_2 \rightarrow \neg C_3$
- $I_3 : C_1 \wedge \neg C_2$

Q3– En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer le couloir que Socrate doit suivre pour rejoindre le monde des vivants.

C_1	C_2	C_3	I_1	I_2	I_3	$(I_1 \wedge I_2 \wedge I_3) \vee (\neg I_1 \wedge \neg I_2 \wedge \neg I_3)$
0	0	0	0	1	0	0
0	0	1	0	1	0	0
0	1	0	0	1	0	0
0	1	1	0	0	0	1
1	0	0	0	1	1	0
1	0	1	1	1	1	1
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Donc comme I_1, I_2 et I_3 sont soient toutes vraies soient toutes fausses, seules deux lignes du tableau conviennent et dans les deux cas, C_3 est vraie, donc pour être sur de rejoindre le monde des vivants, Socrate doit prendre le couloir 3.

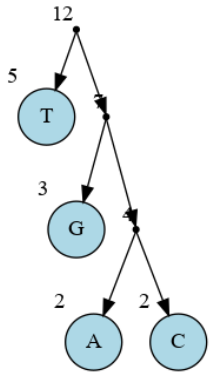
Q4– En admettant que Cerbère ait menti en donnant les trois indications, Socrate pouvait-il suivre d'autres couloirs ? Si oui, le ou lesquels ?

Si on suppose que Cerbère a menti alors on se trouve sur la ligne où I_1, I_2 et I_3 sont fausses et donc deux couloirs permettent de rejoindre le monde des vivants le couloir 2 et le couloir 3.

❑ **Exercice 2** : *Séquence génétique*

On s'intéresse dans cet exercice à la compression de chaines de caractères représentant des séquences génétique codés sur l'alphabet `{A, C, G, T}`. Et on souhaite compresser la séquence S=`ATGTGATGTCCT`. On supposera qu'initialement la séquence est codée en ASCII et que donc chacun des caractères occupe 1 octet (donc 8 bits).

Q5– Construire l'arbre de Huffman associé à la compression de S.



Q6– Donner les codes obtenus pour chacun des quatre caractères A, T, C et G. En déduire le taux de compression de l'algorithme, sans tenir compte de la taille de l'arbre (on pourra conserver une écriture fractionnaire.)

— Code pour `T` = `0`
— Code pour `G` = `10`
— Code pour `A` = `110`
— Code pour `C` = `111`

Comme indiqué sur l'arbre `T` apparaît 5 fois, `G` 3 fois et `A` et `C` chaun deux fois. La taille finale du code est donc : $5 \times 1 + 3 \times 2 + 2 \times 3 + 2 \times 3 = 23$ bits, alors le code initial contenait 12 caractères codés sur 8 bits chacun donc 96 octets, le taux de compression est donc de $\frac{23}{96} \simeq 24\%$.

On rappelle que le principe de la compression LZW est d'attribuer un code aux préfixes rencontrés lors de la lecture du texte à compresser de façon à disposer d'un code compact si ce code se présente à nouveau. On

attribue initialement les codes `A` $\to 0$, `C` $\to 1$, `G` $\to 2$ et `T` $\to 3$ Le début de l'algorithme va donc consister à attribuer un nouveau code au premier préfixe non encore codé qui apparaît lors de la lecture du texte. Et donc, ici, on attribue le code `4` au prefixe `AT` et on émettra le code de `A` c'est à dire 0.

Q7– Poursuivre le déroulement de cet algorithme en complétant le tableau suivant :

Position dans le texte	Code émis	Nouveau préfixe ajouté
ATGTGATGTCCT	0	AT $\to 4$

Position dans le texte	Code émis	Nouveau préfixe ajouté
ATGTGATGTCCT	0	AT $\to 4$
ATGTGATGTCCT	3	TG $\to 5$
ATGTGATGTCCT	2	GT $\to 6$
ATGTGATGTCCT	5	TGA $\to 7$
ATGTGATGTCCT	4	ATT $\to 8$
ATGTGATGTCCT	6	GTT $\to 9$
ATGTGATGTCCT	1	CC $\to 10$
ATGTGATGTCCT	1	CT $\to 11$
ATGTGATGTCCT	3	

On obtient donc la suite de code : `[0; 3; 2; 5; 4; 6; 1; 1; 3]`

Q8– Quel est le taux de compression obtenu (la taille d'un code est un octet) ?

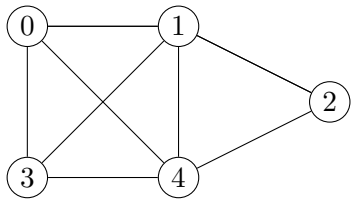
12 sur 9

Q9– Décompresser le texte T codé par suite de codes `[3; 1; 4; 6; 5; 2; 0; 2 ]` sur ce même alphabet en expliquant comment est reconstruit le dictionnaire de décompression.

$T =$ `TCTCTCTCTGAG`

❑ **Exercice 3** : *Triangle dans un graphe*

On considère un graphe *non orienté* $G = (S, A)$ où $A = \{0, \dots, n-1\}$. On dit qu'un sous ensemble de V à trois éléments $\{x, y, z\}$ est un *triangle* de G lorsque $\{x, y\}, \{y, z\}$ et $\{x, z\}$ appartiennent à A. Par exemple, dans le graphe g suivant, $\{0, 1, 3\}$ est un triangle car $\{0, 1\}, \{1, 3\}$ et $\{0, 3\}$ sont des arcs mais $\{0, 1, 2\}$ n'est pas un triangle car $\{0, 2\} \notin A$.



Q10– Donner tous les triangles du graphe g.

On obtient : `{ 0, 1, 3}, { 0, 1, 4}, { 0, 3, 4}, { 1, 2, 4}` et `{ 1, 3, 4}`

Q11– Rappeler la défintion d'un graphe complet. Donner le nombre de triangle d'un graphe complet à n sommets.

Un graphe complet est un graphe dans lequel pour toute paire de sommets (i, j), on a ij qui est un arc c'est à dire $ij \in A$. Donc tous les triplets de sommets sont des triangles, donc il y a autant de triplets de sommets que de triangle, c'est à dire $\binom{|S|}{3}$ triangles

Q12– On rappelle qu'un graphe est *bipartite* lorsqu'on il existe une partition de l'ensemble des sommets S en deux ensembles S_1 et S_2 tel que tout arête ait un élément dans S_1 et l'autre dans S_2. Montrer qu'un graphe bipartite ne contient pas de triangles.

Soit $ij \in A$, et quitte à échanger leur rôle supposons $i \in S_1$ et $j \in S_2$, alors pour tout autre sommet k du graphe, soit $k \in S_1$ et donc $ik \notin A$, soit $k \in S_2$ et donc $kj \notin A$. Donc ijk n'est pas un triangle.

Q13– Afin de lister les triangles d'un graphe, on propose de tester si chaque partie de A à trois éléments est un triangle. Indiquer la complexité d'un tel algorithme en fonction du nombre de sommets $|S|$ du graphe.

Il faut tester les $\binom{|S|}{3} = \frac{|S|(|S|-1)(|S|-2)}{6}$ parties possibles à 3 éléments, cet algorithme a donc une complexité quadratique $\mathcal{O}(|S|^3)$ en nombre de sommets du graphes.

Q14– Un autre algorithme possible pour lister les triangles d'un graphe consiste pour chaque arête $\{x, y\}$ à chercher l'intersection de l'ensemble des sommets z adjacent à x et à y. Donner la complexité de cet algorithme (en fonction de $|A|$ et de $|S|$) si on suppose que l'intersection de deux listes est calculée en temps linéaire du minimum de leur nombre de sommet, c'est à dire que le temps de calcul de l'intersion de `l1` (longueur `n1`) et `l2` (longueur `n2` est un $\mathcal{O}(\min(n1, n2))$).

Cette fois on parcourt l'ensemble des arêtes, et on doit pour chaque arête calculer l'intersection de deux listes contenant au plus $|S|$ éléments et ce calcul est supposé se faire en $\mathcal{O}(|S|)$, donc on obtient une complexité en $\mathcal{O}(|S||A|)$.

Q15– Ecrire en OCaml, une fonction `intersection int list -> int list -> int list` qui calcule en temps linéaire l'intersection de deux listes *en les supposant triées*.

```
let rec intersection l1 l2 =
  match l1, l2 with
  | _, [] -> []
  | [], _ -> []
  | h1::t1, h2::t2 -> if h1=h2 then h1::intersection t1 t2 else
    if h1<h2 then intersection t1 l2 else
      intersection l1 t2;;
```

Q16– Dans la suite de l'exercice, on utilise des graphes représentés par liste d'adjacence en OCaml avec le type :

```
type graphe = {
  taille : int;
  ladj : int list array};;
```

Ecrire une fonction `ajoute graphe -> int -> int -> ()` permettant d'ajouter une arête dans un graphe en *maintenant triée* les listes d'adjacence. On pourra supposer qu'on ajoute toujours un arc qui n'existe pas encore dans le graphe.

```
let ajoute_arete g i j =
  let rec insere elt lst =
    match lst with
    | [] -> [elt]
    | h::t -> if h<elt then h::insere elt t else elt::h::t in
  g.ladj.(i) <- insere j g.ladj.(i);
  g.ladj.(j) <- insere i g.ladj.(j);;
```

Q17– Implémenter l'algorithme décrit à la question **Q5** pour lister les triangles d'un graphe en écrivant une fonction `triangle : graphe -> (int*int*int) list` qui renvoie la liste des triangles du graphe.