UE LU3IN003 - Algorithmique.
Licence d'informatique.

Examen de seconde chance du 14 juin 2021. Durée : 1h30

Documents non autorisés. Seule une feuille A4 portant sur les cours et les TD est autorisée.
Téléphones portables éteints et rangés dans vos sacs
Le barème est indicatif et est susceptible d'être modifié.
Toutes les réponses doivent être justifiées.

Exercice 1 (6 points)

Pour chaque question, cocher la bonne réponse (1.5 point/bonne réponse, -0.5 point si incorrect, 0 si pas de réponse).

Question 1 (1.5/6) — Soit un algorithme diviser pour régner dont la complexité est caractérisée par $T(n)=16T(n/4)+\Theta(n^2)$. Quelle est la complexité de l'algorithme ?

◯ $\Theta(\log n)$ ◯ $\Theta(n)$ ◯ $\Theta(n^2)$ ⊗ $\Theta(n^2 \log n)$

Question 2 (1.5/6) — Supposons que l'on encode un texte comportant uniquement les caractères A, B, C, D en utilisant le codage de Huffman pour les fréquences indiquées ci-dessous. Combien de bits comporte l'encodage ainsi obtenu du texte ?

A	B	C	D
1	2	3	4

◯ 15 ⊗ 19 ◯ 24 ◯ 29

Question 3 (1.5/6) — Quel invariant de boucle est valide au terme de chaque itération de la boucle `while` dans l'algorithme ci-dessous ?

```
int fun (int n)
   i:=0;
   x:=2;
   while (i<n)
      i:=i+1;
      x:=3*x+2;
   return x
```

◯ $i \leq n$ et $x=3^i+1$ ◯ $i \leq n-1$ et $x=3^i+1$ ⊗ $i \leq n$ et $x=3^{i+1}-1$ ◯ $i \leq n-1$ et $x=3^i-1$

Question 4 (1.5/6) — Quel est l'ordre des complexités croissantes pour $f_1(n)=n^{\log_2 n}$, $f_2(n)=2^n$, $f_3(n)=\log_2 n$ et $f_4(n)=\sqrt{n}$?

◯ f_4, f_3, f_1, f_2 ◯ f_4, f_3, f_2, f_1 ⊗ f_3, f_4, f_1, f_2 ◯ f_3, f_4, f_2, f_1

Exercice 2 (6 points)

Vous souhaitez concevoir une projection de photos pour vos ami(e)s. Supposons que vous avez déjà classé les n photos $\{1, \ldots, n\}$ dans l'ordre où vous souhaitez les projeter. Vous avez maintenant la possibilité de mettre une ou deux photos par affichage lors de la projection. Mettre deux photos côté à côte peut en effet faire un joli effet, par exemple si les photos ont le même sujet. Pour $i \in \{1, \ldots, n-1\}$, vous notez $v_i \in \mathbb{N}$ la valeur de l'effet attendu si les photos i et $i+1$ sont projetées côte à côte. Si $v_i < 0$, cela signifie qu'il est préférable de passer les photos i et $i+1$ sur une page chacune plutôt que côte à côte. Si $v_i = 0$, il vous paraît équivalent de passer ces photos chacune sur une page ou les deux côte à côte. Si $v_i > 0$, alors placer les photos côte à côte provoque un joli effet (et plus v_i est important, plus vous estimez l'effet joli).

Vous souhaitez concevoir un algorithme de programmation dynamique permettant de déterminer quelles paires de photos présenter côte à côte (en respectant l'ordre initial choisi) de manière à maximiser la somme des effets attendus : $\sum_{i \in I} v_i$, où $I = \{i \mid i \text{ et } i+1 \text{ sont placées côte à côte}\}$.

Dans cet exercice, nous chercherons uniquement à calculer la somme maximale des effets attendus – on notera cette valeur OPT –, et pas à savoir quelles photos placer côte à côte (une fois OPT calculée, on pourra retrouver cette information en utilisant la méthode du *backtracking*).

Pour tout $i \in \{2, \ldots, n\}$, notons $S(i)$ la valeur maximale des effets que l'on peut obtenir en considérant les photos de 1 à i uniquement (ces photos étant placées dans l'ordre $1, \ldots, n$). On fixera $S(1) = 0$ (il n'y a pas d'effet si l'on considère une photo uniquement).

Question 1 (1/6) — Exprimer OPT en fonction des valeurs $S(i)$.

$OPT = S(n)$.

Question 2 (1/6) — Exprimer $S(2)$ en fonction des valeurs v_i .

$S(2) = \max\{0, v_1\}$.

Question 3 (2/6) — Donner une formule de récurrence permettant de calculer la valeur $S(i)$, pour tout $i \in \{3, \ldots, n\}$.

Considérons la photo i : soit on la projette seule, auquel cas la valeur maximale des effets correspond à $S(i-1)$, soit la projette côte à côte avec la photo $i-1$, auquel cas la valeur maximale des effets est $S(i-2) + v_{i-1}$. Ainsi, pour tout $i > 2$:

$$S(i) = \max\{S(i-1); S(i-2) + v_{i-1}\}.$$

Question 4 (2/6) — Ecrire un algorithme de programmation dynamique calculant OPT. Quelle est la complexité de votre algorithme ?

$Calcule\_effets(n$, tableau v indiquant les valeurs $v_i)$
Créer tableau S de n cases.
$S[1] = 0$
$S[2] = \max(0, v[1])$
Pour i allant de 3 à n
    $S[i] = \max(S[i-1], S[i-2] + v[i-1])$
Retourner $S[n]$

Complexité : $O(n)$.

Exercice 3 (8 points)

On s'intéresse dans cet exercice à deux algorithmes de pré-traitement dans la résolution d'un problème de réseau, nommé problème de routage contraint.

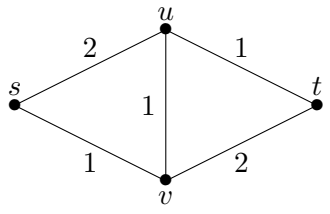
Le réseau considéré est constitué d'antennes qui peuvent, pour certaines, communiquer entre elles. Le réseau est donc modélisé par un graphe non-orienté $G = (S, A)$ où S est l'ensemble des antennes et A est l'ensemble des paires d'antennes qui peuvent communiquer directement, et par une distance $d : A \to \mathbb{R}^+ \setminus \{0\}$. On notera $n = |S|$ et $m = |A|$.

Une **demande** de connexion est donnée sous la forme d'un triplet $(s, t, l) \in S \times S \times \mathbb{R}^+ \setminus \{0\}$. Pour satisfaire cette demande, il faut trouver comment router un message de l'antenne s à l'antenne t par une chaîne de longueur inférieure ou égale à l. Autrement dit, il faut trouver une chaîne de s à t dans G de longueur inférieure ou égale à l, où la longueur d'une chaîne $(s_1, \ldots, s_p)$ ($s_1 = s$ et $s_p = t$ pour une chaîne de s à t) est la somme $\sum_{i=1}^{p-1} d(\{s_i, s_{i+1}\})$ des distances entre les antennes le long de la chaîne. Le problème de routage contraint consiste à satisfaire simultanément le plus de demandes possibles en respectant, outre la contrainte sur la longueur des chaînes, certaines contraintes supplémentaires que l'on ne détaillera pas ici.

Une façon de résoudre ce problème consiste à le formuler comme un programme linéaire en nombre entiers [1]. Afin de réduire la taille de ce programme (et le résoudre plus efficacement), on cherche à identifier pour chaque demande des arêtes interdites. Pour une demande (s, t, l), une arête $a \in A$ est **interdite** si toute chaîne de s à t qui emprunte a est trop longue, *i.e.* de longueur strictement supérieure à l.

Le but de cet exercice est d'identifier les arêtes interdites.

Question 1 (1/8) — Pour le graphe G représenté ci-contre, indiquer toutes les chaînes élémentaires de s à t et leurs longueurs. L'arête $\{u, v\}$ est-elle interdite pour la demande $(s, t, 3)$? Justifier.



1. Ce n'est pas grave si vous ne savez pas ce que c'est.

$s \to u \to t$ de longueur 3 $s \to v \to t$ de longueur 3 $s \to u \to v \to t$ de longueur 5 $s \to v \to u \to t$ de longueur 3	L'arête $\{u, v\}$ n'est pas interdite pour la demande $(s, t, 3)$ puisqu'il existe au moins une chaîne de longueur 3 qui l'emprunte, la chaîne $s \to v \to u \to t$.

On propose l'algorithme suivant.

`est_interdite_version1`

Entrées : $G = (S, A)$ graphe non-orienté,
(s, t, l) une demande sur G,
$u \in S$, $v \in S$ tels que $\{u, v\} \in A$

Sortie : Vrai si $\{u, v\}$ est une arête interdite pour la demande (s, t, l), Faux sinon

Calculer l_1 la longueur d'une plus courte chaîne de s à u
Calculer l_2 la longueur d'une plus courte chaîne de v à t
Retourner Vrai si $l_1 + d(\{u, v\}) + l_2 > l$, retourner Faux sinon

Question 2 (1.5/8) — Que valent l_1 et l_2 lorsque l'on applique `est_interdite_version1` sur le graphe de la question 1 pour la demande $(s, t, 3)$ et les sommets u et v ? Quelle est alors la conclusion de cet algorithme ? (L'arête $\{u, v\}$ est-elle interdite ?) L'algorithme proposé est-il correct ?
Si non, corriger l'algorithme dans le cadre ci-après (nommé `est_interdite_version2`).

$l_1 = d_{\{s,u\}} = 2$, $l_2 = d_{\{v,t\}} = 2$, donc $l_1 + d_{\{u,v\}} + l_2 = 5 > 3$ et l'algorithme `est_interdite_version1` conclut que l'arête est interdite, alors que l'on a vu en Q1 que non. Cela vient du fait que la chaîne de longueur 3 qui passe par l'arête $\{u, v\}$ passe par v puis u et non le contraire. L'algorithme `est_interdite_version1` n'est pas correct et peut être corrigé comme proposé ci-dessous.

`est_interdite_version1`

Entrées : $G = (S, A)$, (s, t, l), $u \in S$, $v \in S$ tels que $\{u, v\} \in A$
Sortie : Vrai si $\{u, v\}$ est une arête interdite pour la demande (s, t, l)

Calculer l_1 la longueur d'une plus courte chaîne de s à u
Calculer l'_1 la longueur d'une plus courte chaîne de s à v
Calculer l_2 la longueur d'une plus courte chaîne de v à t
Calculer l'_2 la longueur d'une plus courte chaîne de u à t
Retourner Vrai si $l_1 + d(\{u, v\}) + l_2 > l$ et $l'_1 + d(\{u, v\}) + l'_2 > l$, Faux sinon

Question 3 (1/8) — On suppose dans la suite que chaque calcul d'une plus courte chaîne est réalisé à l'aide de l'algorithme de Dijkstra (adapté au cas des graphes non-orientés en examinant les

voisins plutôt que les *successeurs*). Donner la complexité de l'algorithme `est_interdite_version2`. *(Si vous n'avez pas répondu à la question précédente, vous donnerez celle de l'algorithme proposé* `est_interdite_version1`.)
Quelle serait la complexité de l'algorithme naïf qui appellerait cet algorithme pour chaque arête afin de déterminer la liste des arêtes interdites pour une demande fixée ?

La complexité de `est_interdite_version2` est en $O((m+n)\log(n))$, car il y a quatre calculs de plus courtes chaînes (et donc autant d'appels à Dijkstra), puis deux tests en $O(1)$.
En répétant cet algorithme pour chaque arête, on a alors une complexité en $O((m+n)m\log(n))$.

Question 4 (2.5/8) — On remarque que certaines plus courtes chaînes entre deux mêmes sommets sont calculées à plusieurs reprises lorsque l'on appelle la fonction `est_interdite_version1` ou `est_interdite_version2` sur toutes les arêtes. Proposer un algorithme `liste_interdites` plus efficace pour déterminer la liste des arêtes interdites pour une demande fixée. Préciser sa complexité temporelle.

L'algorithme proposé ci-dessous a une complexité temporelle en $O((m+n)\log(n))$ pour les deux appels à l'algorithme de Dijkstra, la boucle qui suit est en $O(m)$, donc la complexité temporelle globale est $O((m+n)\log(n))$.

`liste_interdites`

Entrées : $G = (V, E)$ graphe non-orienté, (s, t, l) une demande sur G
Sortie : La liste des arêtes interdites pour la demande (s, t, l)

$D_s \leftarrow$ tableau indexé par $[1..n]$ où $n = |V|$
$D_t \leftarrow$ tableau indexé par $[1..n]$ où $n = |V|$
Appliquer l'algorithme de Dijkstra à partir de s, et enregistrer les distances à s dans D_s
Appliquer l'algorithme de Dijkstra à partir de t, et enregistrer les distances à t dans D_t

$L \leftarrow$ liste vide
pour chaque *arête* $\{u, v\}$ *de* E **faire**
  si $D_s[u] + d(\{u, v\}) + D_t[v] > l$ ***et*** $D_t[u] + d(\{u, v\}) + D_s[v] > l$ **alors**
    Ajouter $\{u, v\}$ à L
  fin
fin
retourner L

Question 5 (2/8) — L'algorithme de Floyd-Warshall permet de calculer les plus courtes chaînes entre toutes les paires de sommets d'un graphe non orienté (S, A) avec une complexité temporelle en $O(|S|^3)$. Cet algorithme retourne un tableau T de taille $n \times n$ tel que $T[i, j]$ indique la longueur d'une plus courte chaîne entre les sommets i et j.
Dans le cas où l'on cherche à identifier la liste des arêtes interdites pour k demandes différentes, on peut alors appeler `liste_interdites` pour chaque demande ou bien utiliser l'algorithme de

Floyd-Warshall avant de tester pour chaque demande et pour chaque arête si l'arête est interdite pour cette demande. Pour les deux cas suivants, dire quel algorithme est préférable (en terme de complexité temporelle). Justifier la réponse.

- k est en $\Theta(1)$ et m en $\Theta(n^2)$
- k est en $\Theta(n)$ et m en $\Theta(n^2)$

En utilisant l'algorithme de Floyd-Warshall, la complexité temporelle est $C_{FW} = O(n^3 + k\,m)$ car une fois tous les plus courts chemins calculés en $O(n^3)$, la détection des arêtes interdites se réduit à un test en $O(1)$ pour chaque arête pour chaque demande.

En utilisant `liste_interdites` pour chaque demande, la complexité temporelle est $C_{LI} = O(k \times (m + n)\log(n))$ (Cf question précédente).

- **Si k est en $\Theta(1)$ et m en $\Theta(n^2)$,**
alors $C_{FW} = O(n^3 + 1 \times n^2) = O(n^3)$, tandis que $C_{LI} = O(1 \times (n^2 + n)\log(n)) = O(n^2 \log(n))$.
On préférera donc répéter `liste_interdites` qui est meilleur en complexité temporelle.

- **Si k est en $\Theta(n)$ et m en $\Theta(n^2)$,**
alors $C_{FW} = O(n^3 + n \times n^2) = O(n^3)$, tandis que $C_{LI} = O(n \times (n^2 + n)\log(n)) = O(n^3 \log(n))$.
On préférera par conséquent ici utiliser F-W.