IFT436 – Algorithmes et structures de données

Université de Sherbrooke

Examen final

Enseignant: Michael Blondin
Date: mercredi 9 décembre 2020
Durée: 3 heures

Directives:

- Vous devez répondre aux questions dans le **cahier de réponses**, et *non* sur ce questionnaire;
- **Une seule feuille** de notes manuscrites au format $8\frac{1}{2}'' \times 11''$ est permise;
- **Aucun matériel additionnel** (notes de cours, fiches récapitulatives, etc.) n'est permis;
- **Aucun appareil électronique** (calculatrice, téléphone, montre intelligente, etc.) n'est permis;
- Vous devez donner **une seule réponse** par sous-question;
- L'examen comporte **5 questions** sur **5 pages** valant un total de **50 points**;
- La correction se base notamment sur la **clarté**, l'**exactitude** et la **concision** de vos réponses, ainsi que sur la **justification** pour les questions qui en requièrent une;
- Les indices d'une séquence **débutent à 1**; autrement dit, $s = [s[1], s[2], \ldots, s[n]]$ si $n = |s|$.

Question 1: analyse d'algorithmes récursifs

Imaginons un système où on déverrouille une machine en entrant des opérations arithmétiques qui font évaluer une expression à zéro. Par exemple, « $(10 \blacksquare 5)+(2 \blacksquare 3)+(1 \blacksquare -1)$ » se déverrouille avec la combinaison $[\div, -, \times]$:

$$(10 \div 5) + (2 - 3) + (1 \times -1) = 2 + (-1) + (-1) = 0.$$

Plus formellement, nous disons qu'une séquence s de n entiers non nuls est *déverrouillable* s'il existe une façon de substituer les occurrences de « $\blacksquare$ » avec les opérateurs $\{+, -, \times, \div\}$ de telle sorte à satisfaire:

$$(s[1] \blacksquare s[2]) + (s[3] \blacksquare s[4]) + \ldots + (s[n-1] \blacksquare s[n]) = 0 \qquad \text{si } n \text{ est pair,}$$
$$s[1] + (s[2] \blacksquare s[3]) + (s[4] \blacksquare s[5]) + \ldots + (s[n-1] \blacksquare s[n]) = 0 \qquad \text{si } n \text{ est impair.}$$

Certaines séquences ne sont pas déverrouillables; par ex. $[10, 5, 2, 3, 1, -1]$ l'est (voir ci-dessus), mais $[4, 2, 1, 1]$ ne l'est *pas* car $4 \blacksquare 2 \in \{2, 6, 8\}$ et $1 \blacksquare 1 \in \{0, 1, 2\}$. Cet algorithme détermine si une séquence est déverrouillable:

Entrées: séquence s de $n \geq 0$ entiers non nuls
Résultat: s est déverrouillable?

```
déverrouillable(s):   // remarque: la séquence vide est trivialement déverrouillable
    chercher(i, accumulateur):
        si i ≤ 1 alors
            retourner (accumulateur = 0)
        sinon
            a ← chercher(i − 2, accumulateur + (s[i − 1] + s[i]))   // choix ■ = +
            b ← chercher(i − 2, accumulateur + (s[i − 1] − s[i]))   // choix ■ = −
            c ← chercher(i − 2, accumulateur + (s[i − 1] × s[i]))   // choix ■ = ×
            d ← chercher(i − 2, accumulateur + (s[i − 1] ÷ s[i]))   // choix ■ = ÷
            retourner (a ∨ b) ∨ (c ∨ d)
    si n est pair alors x ← 0 sinon x ← s[1]
    retourner chercher(n, x)
```

(a) Donnez une récurrence linéaire t où $t(n)$ dénote le nombre d'opérations élémentaires exécutées par l'appel initial « `chercher(`n`,` x`)` » par rapport à n. Considérez ces opérations élémentaires: 2,5 pts

— arithmétique: $+$, $-$, etc.; — affectations; — accès aux éléments d'une séquence;

— comparaisons: $=$, $\neq$, etc.; — opérations logiques: $\vee$, $\wedge$, etc.

Supposez également que *tous* les termes d'une opération logique sont évalués (contrairement à certains langages de programmation où, par exemple, q est seulement évalué dans $p \vee q$ lorsque $p =$ *faux*).

(b) Identifiez la forme close de t. Vous n'avez *pas* à identifier les valeurs des constantes, mais vous devez indiquer le système d'équations que vous auriez à résoudre pour les identifier. Laissez une trace de votre démarche. 3 pts

(c) On pourrait améliorer le temps d'exécution dans le meilleur cas en évitant d'évaluer tous les termes a, b, c et d dès que l'un des trois premiers mènerait à vrai. Cet élagage améliorerait-il aussi le temps de l'algorithme dans le *pire cas*? Justifiez. 2,5 pts

Question 2: diviser-pour-régner

Considérons ce problème:

ENTRÉE: séquence s de $n \geq 1$ séquences de k entiers triés
SORTIE: séquence $s[1] + s[2] + \cdots + s[n]$ triée (où « $+$ » est la concaténation)

Autrement dit, le problème consiste à combiner n séquences, chacune de k éléments triés, de telle sorte à obtenir une séquence de $n \cdot k$ éléments triés *globalement*. Par exemple:

sur entrée $s = [[3, 6, \ldots, 24], [2, 5, \ldots, 23], [1, 4, \ldots, 22]]$ la sortie devrait être $[1, 2, 3, \ldots, 24]$.

Supposons qu'on ait accès à une routine « `fusion`(x, y) » qui fusionne deux séquences triées x et y en exactement $|x| + |y|$ opérations élémentaires. Ces deux algorithmes résolvent le problème ci-dessus:

```
algo-itératif(s):
    n ← |s|
    z ← []
    pour i ∈ [1..n]
        z ← fusion(z, s[i])
    retourner z
```

```
algo-div-pour-régner(s):
    n ← |s|
    si n = 1 alors
        retourner s[1]
    sinon
        m ← n ÷ 2
        x ← algo-div-pour-régner(s[1 : m])
        y ← algo-div-pour-régner(s[m + 1 : n])
        retourner fusion(x, y)
```

Dans chacun des scénarios suivants, dites si « `algo-div-pour-régner` » est plus efficace que « `algo-itératif` » asymptotiquement, c.-à-d. en terme de $\mathcal{O}$. Vous pouvez supposer que n est une puissance de 2 si cela vous aide. Justifiez.

(a) Il y a n séquences, chacune de $k = 8$ éléments. 4 pts

(b) Il y a $n = 8$ séquences, chacune de k éléments. 4 pts

(c) Il y a n séquences, chacune de $k = n$ éléments. 4 pts

Question 3: force brute et programmation dynamique

Considérons un jeu où Tux récolte des objets en montant un escalier:

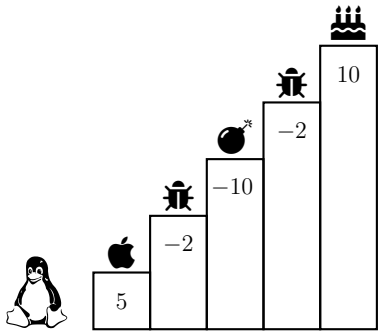
— Tux débute avec 0 point au bas d'un escalier de $n \geq 1$ marches;

— Chaque marche possède un objet de poids positif (gain de points) ou négatif (perte de points);

— Tux peut franchir *au plus* deux marches à la fois;

— Tux doit terminer sur la marche n (et peut terminer avec un pointage négatif).

Par exemple, sur l'escalier de 5 marches ci-dessous, Tux obtient $(-2) + (-2) + 10 = 6$ points en montant $[2, 2, 1]$ marches, et obtient un pointage maximal de $5 + (-2) + (-2) + 10 = 11$ en montant $[1, 1, 2, 1]$ marches.



Formellement, un escalier est représenté par une séquence s d'entiers, par ex. $s = [5, -2, -10, -2, 10]$ ci-dessus.

(a) Afin d'identifier le pointage maximal que Tux peut obtenir, nous pourrions explorer *tous* les parcours possibles. Cette procédure fonctionnerait-elle en temps polynomial par rapport à n? Justifiez. 3 pts

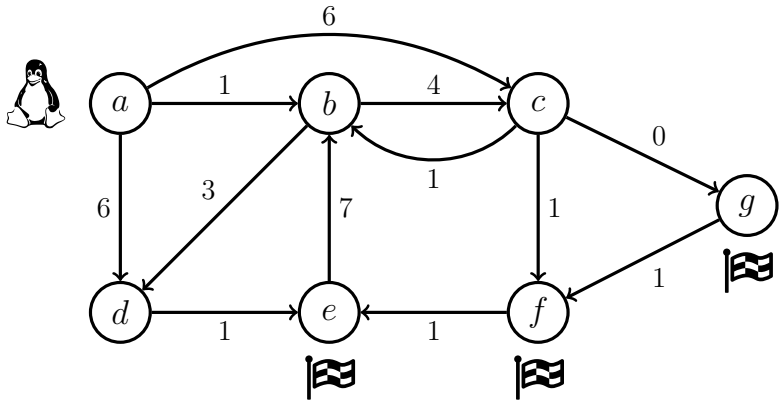
(b) Quel est le pointage maximal pour l'escalier $[5, 2, -3, -7, 3]$? Justifiez. 2 pts

(c) Donnez un algorithme, décrit sous forme de pseudocode, qui identifie le pointage maximal d'un escalier de n marches en temps $\mathcal{O}(n)$. Justifiez le temps d'exécution. 5 pts

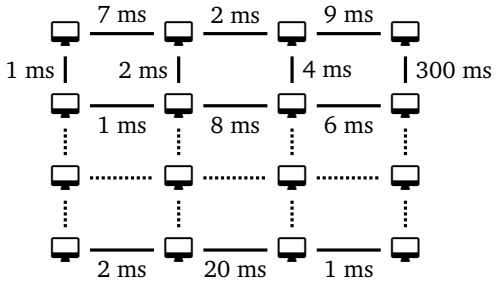
(d) Expliquez, en mots et/ou avec du pseudocode, comment adapter l'algorithme afin d'identifier une séquence de déplacements qui mène au pointage maximal; par exemple, « $[1, 1, 2, 1]$ » sur l'escalier illustré ci-dessus. 2 pts

Question 4: plus courts chemins

(a) Considérons le graphe pondéré $\mathcal{G}$ illustré ci-dessous. Tux débute au sommet de départ a et désire atteindre un des sommets d'arrivée e, f ou g (n'importe quel des trois). Aidez Tux à identifier un chemin de longueur *minimale* en exécutant l'algorithme de Dijkstra sur $\mathcal{G}$. Exécutez l'algorithme en marquant le *moins de sommets possible*. Laissez une trace des itérations effectuées en indiquant les sommets marqués et les distances partielles identifiées. Expliquez pourquoi vous pouvez arrêter à l'itération choisie. 3 pts



(b) Considérons un réseau de n^2 machines dont les connexions sont sous forme de grille $n \times n$, où une machine peut communiquer avec un voisin en un certain temps en millisecondes; c.-à-d. d'une telle forme: 2,5 pts



Une machine peut communiquer avec une autre via un chemin qui les relie dans le réseau. Afin de calculer le temps minimal de communication entre *chaque paire* de machines, est-ce plus efficace d'utiliser l'algorithme de Floyd–Warshall; l'algorithme de Dijkstra pour chaque machine; ou l'algorithme de Bellman–Ford pour chaque machine? Justifiez.

Rappel: leurs temps d'exécution, tels qu'analysés en classe, appartiennent à $\Theta(|V|^3)$, $\mathcal{O}(|V| \log |V| + |E|)$ et $\Theta(|V| \cdot |E|)$ respectivement.

(c) Le plus court chemin d'un sommet s vers lui-même est le chemin vide de longueur 0 (en supposant l'absence de cycle négatif). Dans certaines applications, on désire plutôt identifier un plus court chemin *non vide* où on quitte s et on y revient. Expliquez, en mots et/ou à l'aide de pseudocode, comment résoudre ce problème: 2,5 pts

ENTRÉE: graphe dirigé $\mathcal{G} = (V, E)$ pondéré par une séquence p de poids positifs, sommet $s \in V$

SORTIE: longueur d'un plus court chemin *non vide* de s vers s

Vous pouvez invoquer des algorithmes couverts en classe.

Question 5: algorithmes probabilistes

Il n'est pas nécessaire d'évaluer vos résultats numériquement; des expressions symboliques suffisent.

(a) Les deux algorithmes ci-dessous simulent un dé à six faces. L'algorithme de gauche est celui vu en classe, et celui de droite est une légère variation. Lequel a le plus petit nombre espéré de *pièces lancées*? Justifiez. 5 pts

Entrées: —
Résultat: nombre $x \in [1, 6]$ choisi de façon aléatoire et uniforme

```
dé-A():
    faire
        choisir un bit y_2 à pile ou face
        choisir un bit y_1 à pile ou face
        choisir un bit y_0 à pile ou face
    tant que y_2 = y_1 = y_0
    retourner 4 · y_2 + 2 · y_1 + y_0
```

Entrées: —
Résultat: nombre $x \in [1, 6]$ choisi de façon aléatoire et uniforme

```
dé-B():
    choisir un bit y_2 à pile ou face
    faire
        choisir un bit y_1 à pile ou face
        choisir un bit y_0 à pile ou face
    tant que y_2 = y_1 = y_0
    retourner 4 · y_2 + 2 · y_1 + y_0
```

(b) Cet algorithme cherche à déterminer si une séquence d'une certaine forme contient un nombre impair: 5 pts

Entrées: séquence s dont les éléments sont des entiers non négatifs, dont la taille est paire, et dont la moitié des éléments sont égaux à $a \in \mathbb{N}$ et l'autre moitié à $b \in \mathbb{N}$ où $a \neq b$
Résultat: s contient un nombre impair?

```
parité(s):
    faire 3 fois
        choisir i ∈ [1..|s|] aléatoirement de façon uniforme
        choisir j ∈ [1..|s|] aléatoirement de façon uniforme
        si (s[i] mod 2 = 1) ∨ (s[j] mod 2 = 1) alors
            retourner vrai
    retourner (s[1] mod 2 = 1)
```

Dites, avec justification, si l'algorithme est de Las Vegas ou de Monte Carlo. Expliquez comment utiliser l'amplification afin d'obtenir une probabilité d'erreur inférieure ou égale à $1/2^{300}$. Ne modifiez *pas* le pseudocode de « `parité` »; décrivez plutôt une nouvelle routine qui l'utilise en boîte noire. Justifiez.