IFT436 – Algorithmes et structures de données

Université de Sherbrooke

Examen final

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

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*).

$$t(n) \coloneqq \begin{cases} 2 & \text{si } n = 0 \text{ ou } n = 1, \\ 4 \cdot t(n-2) + \underbrace{32}_{1+4\cdot 7+3} & \text{si } n \geq 2. \end{cases}$$

(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

La récurrence linéaire non homogène $t(n) - 4 \cdot t(n-2) = 32 \cdot 1^n$ mène au polynôme $(x^2 - 4)(x-1) = (x-2)(x+2)(x-1)$. La forme close est donc $t(n) = c_1 \cdot 2^n + c_2 \cdot (-2)^n + c_3$. On obtient le système:

$$\begin{aligned} t(0) = \ \ 2 = \ \ c_1 + \ \ c_2 + c_3 \\ t(1) = \ \ 2 = 2c_1 - 2c_2 + c_3 \\ t(2) = 40 = 4c_1 + 4c_2 + c_3 \end{aligned}$$

(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

Non, par ex. les entrées de la forme $[4, 2, \ldots, 4, 2]$ ne sont pas déverrouillables car toutes les combinaisons sont strictement positives. Ainsi, tous les appels évalueront à « *faux* » et rien ne sera élagué.

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.

L'algorithme itératif a un temps d'exécution de $\Theta(k + 2k + \ldots + nk) = \Theta(k \cdot n(n+1)/2) = \Theta(kn^2)$.

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

Le temps d'exécution d'`algo-div-pour-régner` est décrit par $t(n) = 2 \cdot t(n/2) + \mathcal{O}(n)$. Par le théorème maître, on a $c = 2 = 2^1 = b^d$ et ainsi $t \in \mathcal{O}(n \log n)$, qui est plus efficace que $\Theta(8n^2) = \Theta(n^2)$.

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

Il y a 7 fusions d'au plus $8k$ éléments. Ainsi, le temps d'exécution d'`algo-div-pour-régner` appartient à $\Theta(k)$. La complexité est la même pour les deux algorithmes car $\Theta(k \cdot 8^2) = \Theta(k)$.

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

Sol. 1: Comme `algo-div-pour-régner` coupe en deux, l'arbre de récursion est de hauteur $\mathcal{O}(\log n)$. Chaque étage a un coût total de $\mathcal{O}(nk) = \mathcal{O}(n^2)$ car on fusionne toutes les séquences. Ainsi, son temps d'exécution appartient à $\mathcal{O}(n^2 \log n)$, ce qui est plus efficace que $\Theta(kn^2) = \Theta(n^3)$.

Sol. 2: Le temps est décrit par $t(n) = 2 \cdot t(n/2) + nk$. En substituant, on a: $2 \cdot t(n/2) + nk = 4 \cdot t(n/4) + 2nk = 8 \cdot t(n/8) + 3nk = \cdots = \log(n) \cdot nk = n^2 \log n$, qui est plus efficace que $\Theta(kn^2) = \Theta(n^3)$.

Question 3: force brute et programmation dynamique

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

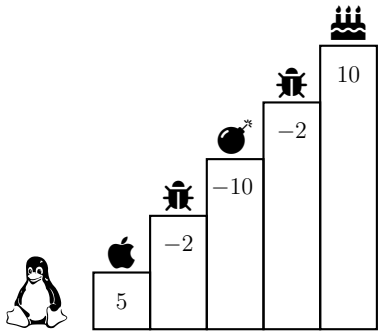
— Tux débute avec 0 point au bas d'un escalier de $n \geq 1$ marches;

— 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

Sol. 1: Non, il y a 2 façons de monter 2 marches: $[1,1]$ ou $[2]$. Il y a donc $\geq 2^{n/2}$ parcours.

Sol. 2: Non, il y a au moins 2 façons de monter 3 marches: $[1,2]$ ou $[2,1]$. Il y a donc $\geq 2^{n/3}$ parcours.

Sol. 3: Non, le nombre de parcours est décrit par la suite de Fibonacci $f(n)=f(n-1)+f(n-2)$.

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

Sol. 1: Soit $M[i]$ le pointage max. du départ vers la marche i. On a $M[i]=s[i]+\max(M[i-1],\ M[i-2])$ où on considère les positions invalides comme valant 0. Le pointage max. est 7 car $M=[5,7,4,0,7]$.

Sol. 2: On doit forcément prendre 3 au sommet. Si on y arrive via -7, on peut améliorer la solution en passant directement par -3. Comme les marche précédentes sont positives, on augmente la solution en les prenant toutes. On obtient donc $5+2+(-3)+3=7$.

(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

Sol. 1:

```
pointage(s):
    initialiser un tableau M[1..n] avec 0
    pour i ∈ [1..n]
        si i > 1 alors a ← M[i − 1] sinon a ← 0
        si i > 2 alors b ← M[i − 2] sinon b ← 0
        M[i] ← s[i] + max(a, b)
    retourner M[n]
```

L'initialisation se fait en temps $\Theta(n)$. Comme les autres opérations sont élémentaires et qu'on exécute le corps n fois, on obtient $\Theta(n)$ au total.

Sol. 2: Même chose, mais vers l'arrière avec l'identité $M[i] = s[i] + \max(M[i+1], M[i+2])$.

Sol. 3–4: Mêmes idées, mais en procédant récursivement et en utilisant la mémoïsation.

Sol. 5: Comme la solution 1 mais avec peu de mémoire (à la Fibonacci):

```
pointage(s):
    si n = 1 alors retourner s[1]
    sinon
        a, b ← 0, s[1]
        pour i ∈ [2..n]
            a, b ← b, s[i] + max(a, b)
        retourner b
```

Sol. 6: Même chose, mais vers l'arrière.

(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

Sol. 1: Lors du calcul de $M[i]$, on mémorise une marche $j \in \{i-1, i-2\}$ telle que $M[j] = \max(a, b)$. Cela permet ensuite de rebrousser chemin de $M[n]$ et d'identifier les bonds vers le départ.

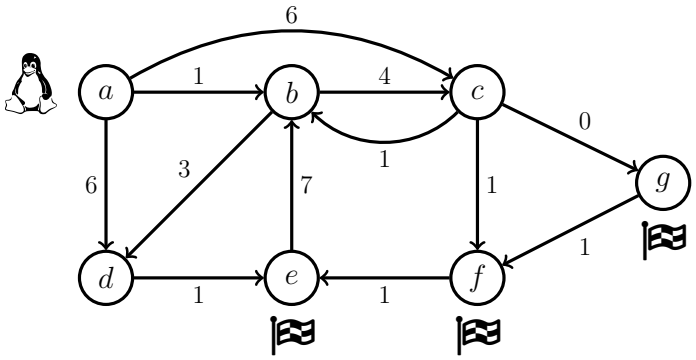
Sol. 2: Après le calcul de M, on rebrousse chemin à partir de $M[n]$ en choisissant une marche $j \in \{i-1, i-2\}$ telle que $T[j] = \max(a, b)$.

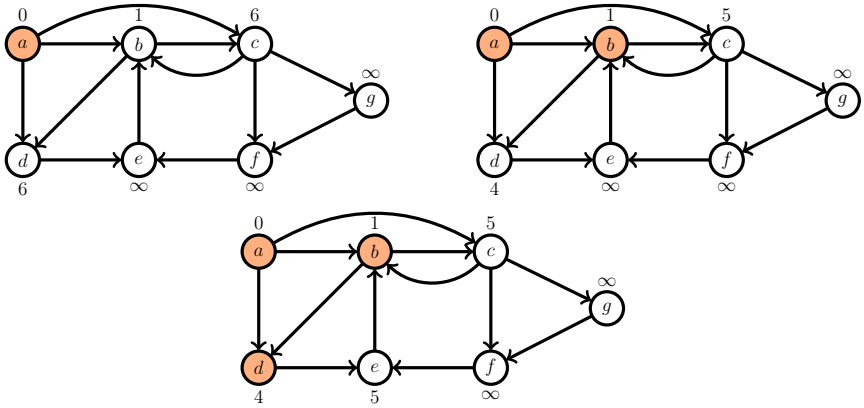
Sol. 3: Lors du calcul de $M[i]$, on mémorise 1 si $\max(a, b) = a$ et 2 sinon. Cela permet ensuite de rebrousser chemin à partir de $M[n]$ et de suivre les bonds vers le départ.

Sol. 4–6: Mêmes solutions pour l'algo. vers l'arrière en reversant tout dans l'autre sens.

Question 4: plus courts chemins

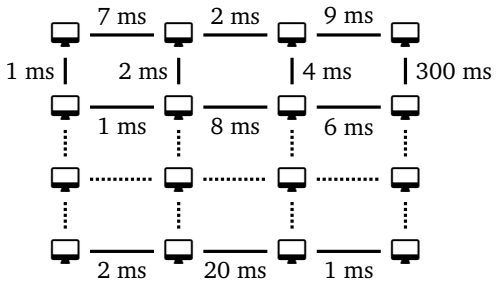
(a) Considérons le graphe pondéré $\mathcal{G}$ illustré ci-dessous. Tux 🐧 débute au sommet de départ a et désire atteindre un des sommets d'arrivée e, f ou g (n'importe quel des trois). Aidez Tux à identifier un chemin de longueur *minimale* en exécutant l'algorithme de Dijkstra sur $\mathcal{G}$. Exécutez 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





À la prochaine itération, on peut marquer c ou e car ils sont minimaux. On peut donc s'arrêter ici en marquant e, qui est un sommet d'arrivé (on n'améliorera plus jamais les distances).

(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.

Sol. 1: On a $|V| = n^2$ et $2n^2 \leq |E| \leq 4n^2$. Les temps d'exécution sont de $\Theta(n^6)$, $\mathcal{O}(n^2 \cdot n^2 \log n)$ et $\Theta(n^2 \cdot n^4)$. L'algo. de Dijkstra est donc préférable. (on peut l'utiliser car le temps n'est pas négatif.)

Sol. 2 (même chose avec autre notation): On a $2|V| \leq |E| \leq 4|V|$. Les temps d'exécution sont de $\Theta(|V|^3)$, $\mathcal{O}(|V| \cdot |V| \log |V|)$ et $\Theta(|V| \cdot |V|^2)$. L'algo. de Dijkstra est donc préférable.

(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.

Sol. 1: On lance l'algo. de Dijkstra sur chaque successeur v de s et on obtient une longueur minimale $d[v]$ de v vers s. On retourne $\min\{p[s, v] + d[v] : v \text{ est un successeur de } s\}$.

Sol. 2: Même chose avec l'algo. de Bellman-Ford.

Sol. 3: On lance l'algo. de Floyd-Warshall qui retourne une matrice d. On retourne $\min\{p[s, v]+d[v, s] : v \text{ est un successeur de } s\}$.

Sol. 4: On ajoute un nouveau sommet s' avec une arête de poids 0 vers chaque successeur de s. On lance l'algo. de Dijkstra sur s' et on retourne la longueur minimale de s' vers s.

Sol. 5–6: Même chose avec l'algo. de Bellman-Ford ou de Floyd-Warshall.

Sol. 7: On lance l'algo. de Dijkstra sur s qui donne une séq. d et on retourne $\min\{d[v]+p[v, s] : v \neq s\}$.

Sol. 8: Même chose avec l'algo. de Bellman-Ford.

Sol. 9: On lance l'algo. de Floyd-Warshall qui retourne d. On retourne $\min\{d[s, v] + d[v, s] : v \neq s\}$.

Sol. 10: On lance l'algo. de Floyd-Warshall avec la diagonale initialisée à ∞, puis on retourne $d[s, s]$.

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
```

On quitte les boucles avec probabilité $6/8$ et $3/4$, respectivement. Le nombre espéré de pièces lancées est donc de $3 \cdot 8/6 = 4$ et $1 + 2 \cdot 4/3 = 11/3$ (loi géométrique) L'algo. `dé-B` en lance donc moins.

(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.

Monte Carlo car le temps est constant, et il y a erreur lorsque $s[1]$ est pair et qu'on choisit 6 nombres pairs. La probabilité d'erreur est donc de $1/2^6$. Si l'algo. retourne « *vrai* », alors sa sortie est correcte. En l'exécutant 50 fois et en prenant le $\vee$ des sorties, on réduit la probabilité d'erreur à $(1/2^6)^{50} = 1/2^{300}$.

Sol. alternative: On peut arrêter avant la fin des 50 itérations dès qu'on obtient « *vrai* ».