

IFT436 – Algorithmes et structures de données
Université de Sherbrooke

Examen final

Enseignant: Michael Blondin
Date: mercredi 22 décembre 2021
Durée: 3 heures

Directives:

- Vous devez répondre aux questions dans le **cahier de réponses**, et *non* sur ce questionnaire;
- **Une seule feuille** de notes au format $8\frac{1}{2}'' \times 11''$ est permise;
- Les **fiches récapitulatives** des chapitres 5 à 8 se trouvent à la fin de ce questionnaire;
- **Aucun matériel additionnel** 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, $s = [s[1], s[2], \dots, s[n]]$ si $n = |s|$.

Question 1: analyse d'algorithmes récursifs

Un artiste a produit quatre exemplaires de n œuvres. Un exemplaire de l'œuvre i vaut $v[i]$ dollars et possède une aire de $a[i]$ cm². L'artiste désire minimiser la superficie de son kiosque du marché de Noël de Sherbrooke, mais en étant en mesure d'obtenir au moins b dollars en vendant tout ce qu'il y exposera. Formellement, il cherche donc à identifier des valeurs $x[1], \dots, x[n] \in [0..4]$ qui minimisent $a[1] \cdot x[1] + \dots + a[n] \cdot x[n]$ tout en satisfaisant $v[1] \cdot x[1] + \dots + v[n] \cdot x[n] \geq b$. Une informaticienne lui propose cet algorithme:

Entrées: deux séquences v et a de $n \geq 0$ entiers positifs, et un entier positif b

Résultat: aire minimale

aire-min(v, a, b):

```

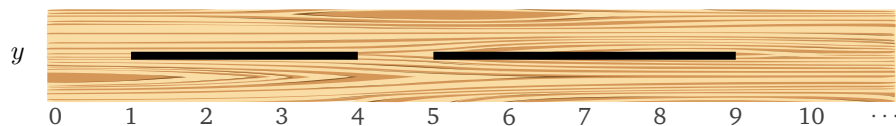
résoudre( $i, aire, valeur$ ):
  si  $i = n + 1$  alors
    si  $valeur \geq b$  alors retourner  $aire$ 
    sinon retourner  $\infty$ 
  sinon
     $m \leftarrow \infty$ 
    pour  $q \in [0..4]$ 
       $m \leftarrow \min(m, \text{résoudre}(i + 1, aire + a[i] \cdot q, valeur + v[i] \cdot q))$ 
    retourner  $m$ 
retourner résoudre(1, 0, 0)

```

- (a) Donnez une récurrence linéaire t où $t(n)$ dénote le nombre d'opérations élémentaires exécutées par l'appel initial « résoudre(1, 0, 0) » par rapport à n . Considérez ces opérations élémentaires: 2,5 pts
- arithmétique: +, -, min, etc.;
 - affectations;
 - comparaisons: =, \geq , etc.;
 - accès aux éléments d'une séquence.
- (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) Expliquez comment améliorer l'algorithme à l'aide d'une forme d'élagage. 2,5 pts

Question 2: diviser-pour-régner

Considérons une machine-outil à commande numérique. Sa librairie logicielle permet notamment de découper une séquence de traits dans une planche de bois. Formellement, un *trait* est une paire $(g, d) \in \mathbb{N}^2$ où $g < d$ indiquent le côté gauche et droite. Par exemple, sur entrée $s = [(1, 3), (5, 7), (2, 4), (6, 9)]$, la machine effectue cette découpe, où y est la position verticale actuelle de l'outil:



Sur cette entrée s , la machine effectue des découpages superflus, par ex. après avoir découpé le trait $(1, 3)$, il n'est pas nécessaire de découper $(2, 4)$ en entier car $(2, 3)$ a déjà été retiré de la planche. Comme la machine est lente, nous cherchons à implémenter un algorithme réduire qui retire les chevauchements d'une séquence de traits (avant de les envoyer à la machine). Par exemple, $\text{réduire}(s)$ peut retourner $[(1, 4), (5, 9)]$.

- (a) Complétez le pseudocode ci-dessous: 7 pts

Entrée: une séquence s de traits

Résultat: une séquence de traits sans chevauchements équivalente à s

réduire(s):

```

si  $|s| \leq 1$  alors
  | retourner  $s$ 
sinon
  |  $m \leftarrow |s| \div 2$ 
  |  $x \leftarrow \text{réduire}(s[1:m])$ 
  | /* Complétez ce segment */
  | /* de pseudocode (sur autant de lignes que nécessaire) */

```

Afin de rendre votre code moins cryptique, utilisez au besoin ces fonctions auxiliaires, où t est un trait:

```

gauche( $t$ ):
  | retourner  $t[1]$ 
droite( $t$ ):
  | retourner  $t[2]$ 

```

- (b) Nous disons qu'un trait (a, b) contient un point x si $x \in [a..b]$. Par exemple, le trait $(5, 7)$ contient 5, 6 et 7. Complétez cet algorithme: 2 pts

Entrées: une séquence s de traits triés et sans chevauchements, et $x \in \mathbb{N}$

Résultat: vrai s'il y a un trait de s qui contient x , faux sinon

contient(s, x):

```

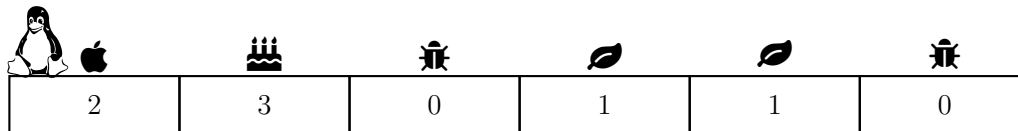
contient'(lo, hi):
  si lo = hi alors
    /* à compléter */
  sinon
    i ← (lo + hi) ÷ 2
    si x ≤ droite(s[i]) alors
      retourner contient'(lo, i)
    sinon
      /* à compléter */
  retourner contient'(1, |s|)

```

- (c) Quel est le temps d'exécution asymptotique des algorithmes obtenus en (a) et (b) dans le pire cas? Vous pouvez supposer que $|s|$ est une puissance de deux. Justifiez. 3 pts

Question 3: force brute et programmation dynamique

Considérons un jeu où Tux se déplace vers la droite sur une banquise de n blocs. Le bloc i lui permet de sauter jusqu'au bloc $i + s[i]$ (grâce à de la nourriture qui lui offre momentanément une dose d'énergie). Par exemple, considérons la banquise ci-dessous. Sur le premier bloc, Tux peut se rendre en un seul saut au bloc 2 ou au bloc 3 (car $i = 1$ et $s[1] = 2$). Tux débute à la case 1 et désire atteindre la case n avec le moins de sauts possible.

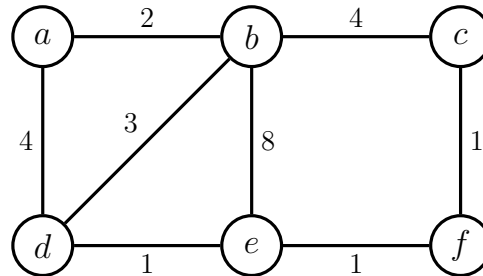


Formellement, une banquise est représentée par une séquence s d'entiers non négatifs, par ex. $s = [2, 3, 0, 1, 1, 0]$ ci-dessus. Remarquons qu'il est possible que Tux ne puisse pas se rendre à la case n , par ex. si $s[1] = 0$. De plus, la valeur du dernier bloc n'influence pas le parcours puisque Tux a terminé (c'est de la pure gourmandise).

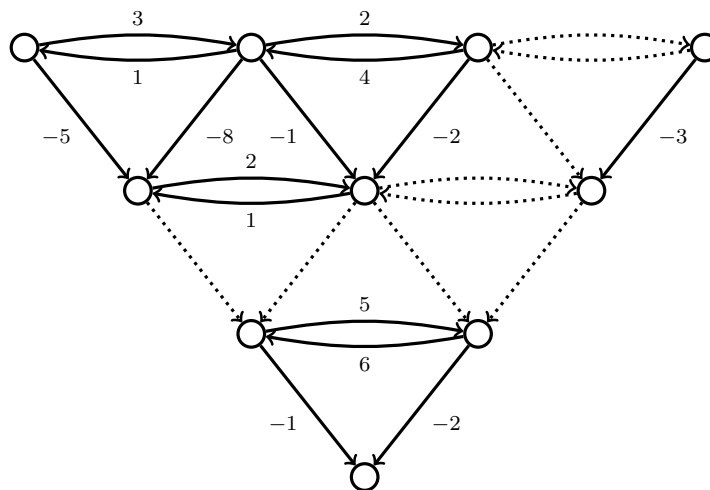
- (a) Afin d'identifier le nombre de sauts minimal que Tux doit effectuer pour se rendre à la case n , nous pourrions explorer exhaustivement tous les parcours possibles. Cette procédure fonctionnerait-elle en temps polynomial dans le pire cas par rapport à n ? Justifiez. 3 pts
- (b) Quel est le nombre de sauts minimal pour la banquise $[2, 3, 0, 1, 1, 0]$? Justifiez. 2 pts
- (c) Donnez un algorithme, décrit sous forme de pseudocode, qui identifie le nombre de sauts minimal pour une banquise de n blocs en temps polynomial. 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 sauts qui mène à une solution minimale. 2 pts

Question 4: plus courts chemins

- (a) Expliquez comment traduire le problème de la question 3 en un problème de plus court chemin. Indiquez, en mots, comment obtenir le graphe résultant et les sommets de départ et d'arrivée. 2,5 pts
- (b) Exécutez l'algorithme de Dijkstra sur le graphe pondéré ci-dessous afin de déterminer la longueur d'un plus court chemin de a vers f . Arrêtez l'exécution *dès que possible*. Laissez une trace des itérations effectuées en indiquant les sommets marqués et les distances partielles identifiées. 3 pts



- (c) Considérons une carte triangulaire, avec n cases de départ et une case d'arrivée, organisée de cette façon: 2,5 pts



Plus précisément, l'étage du haut possède n cases, et chaque étage subséquent en possède une de moins. Sur chaque étage on peut se déplacer librement de gauche à droite avec un coût positif. De plus, on peut descendre vers une case adjacente avec un coût négatif (mais pas remonter). On cherche à identifier une case de départ sur l'étage du haut qui minimise le coût d'atteindre la case tout en bas. Afin d'y arriver, est-ce plus efficace d'utiliser l'algorithme de Floyd–Warshall; l'algorithme de Dijkstra pour chaque case de départ; ou l'algorithme de Bellman–Ford pour chaque case de départ? 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.

Question 5: algorithmes probabilistes

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

Nous disons qu'une séquence s de n entiers est *triangulaire* si:

- n est un multiple de 6;
- s contient $(1/6)n$ occurrences d'un nombre a , $(2/6)n$ occurrences d'un nombre b , et $(3/6)n$ occurrences d'un nombre c ,
- $a \neq b$, $a \neq c$ et $b \neq c$.

Par exemple, la séquence $s = [20, 10, 15, 20, 20, 10]$ est triangulaire. Considérons ces deux algorithmes probabilistes qui cherchent à identifier le minimum d'une séquence triangulaire:

Entrée: séquence triangulaire s de taille $n \in \mathbb{N}_{\geq 6}$

Résultat: $\min(s)$

triangulaireA(s):

faire

choisir $i \in [1..n]$ aléatoirement de façon uniforme

choisir $j \in [1..n]$ aléatoirement de façon uniforme

choisir $k \in [1..n]$ aléatoirement de façon uniforme

tant que $|\{s[i], s[j], s[k]\}| \neq 3$

 // trois valeurs distinctes?

retourner $\min(s[i], s[j], s[k])$

Entrée: séquence triangulaire s de taille $n \in \mathbb{N}_{\geq 6}$

Résultat: $\min(s)$

triangulaireB(s):

faire 100 fois

choisir $i \in [1..n]$ aléatoirement de façon uniforme

choisir $j \in [1..n]$ aléatoirement de façon uniforme

si $|\{s[1], s[i], s[j]\}| = 3$ **alors**

retourner $\min(s[1], s[i], s[j])$

 // trois valeurs distinctes?

retourner $s[1]$

Pour chacun des algorithmes triangulaireA et triangulaireB:

(a) Dites s'il est de Las Vegas ou de Monte Carlo. Expliquez pourquoi.

3 pts

(b) Selon votre réponse en (a), identifiez la probabilité d'erreur ou le temps espéré de l'algorithme. Justifiez.

7 pts

Annexe:

Fiches récapitulatives

5. Algorithmes récursifs et approche diviser-pour-régner

Diviser-pour-régner

- ▶ A) découper en sous-problèmes disjoints
- ▶ B) obtenir solutions récursivement
- ▶ C) s'arrêter aux cas de base (souvent triviaux)
- ▶ D) combiner solutions pour obtenir solution globale
- ▶ Exemple: tri par fusion $\mathcal{O}(n \log n)$

Réurrences linéaires

- ▶ Cas homogène: $\sum_{i=0}^d a_i \cdot t(n-i) = 0$
- ▶ Polynôme caractéristique: $\sum_{i=0}^d a_i \cdot x^{d-i}$
- ▶ Forme close: $t(n) = \sum_{i=1}^d c_i \cdot \lambda_i^n$ où les λ_i sont les racines
- ▶ Constantes c_i : obtenues en résolvant un sys. d'éq. lin.
- ▶ Cas non homo.: si $s = c \cdot b^n$, on multiplie poly. par $(x-b)$
- ▶ Exemple:
 - Récurrence: $t(n) = 3 \cdot t(n-1) + 4 \cdot t(n-2)$
 - Poly. carac.: $x^2 - 3x - 4 = (x-4)(x+1)$
 - Forme close: $t(n) = c_1 \cdot 4^n + c_2 \cdot (-1)^n$

Autres méthodes

- ▶ Substitution: remplacer $t(n), t(n-1), t(n-2), \dots$ par sa déf. jusqu'à deviner la forme close
- ▶ Arbres: construire un arbre représentant la récursion et identifier le coût de chaque niveau

Quelques algorithmes

- ▶ Hanoi: $src[1:n-1] \rightarrow tmp, src[n] \rightarrow dst, tmp[1:n-1] \rightarrow dst$ $\mathcal{O}(2^n)$
- ▶ Exp. rapide: exploiter $b^n = (b^{n \div 2})^2 \cdot b^{n \bmod 2}$ $\mathcal{O}(\log n)$
- ▶ Mult. rapide: calculer $(a+b)(c+d)$ en 3 mult. $\mathcal{O}(n^{\log 3})$
- ▶ Horizon: découper blocs comme tri par fusion $\mathcal{O}(n \log n)$

Théorème maître (allégé)

- ▶ $t(n) = c \cdot t(n \div b) + f(n)$ où $f \in \mathcal{O}(n^d)$:
 - $\mathcal{O}(n^d)$ si $c < b^d$
 - $\mathcal{O}(n^d \cdot \log n)$ si $c = b^d$
 - $\mathcal{O}(n^{\log_b c})$ si $c > b^d$

6. Force brute

Approche

- ▶ Exhaustif: essayer toutes les sol. ou candidats récursivement
- ▶ Explosion combinatoire: souvent # solutions $\geq b^n, n!, n^n$
- ▶ Avantage: simple, algo. de test, parfois seule option
- ▶ Désavantage: généralement très lent et/ou avare en mémoire

Techniques pour surmonter explosion

- ▶ Élagage: ne pas développer branches inutiles
- ▶ Contraintes: élaguer si contraintes enfreintes
- ▶ Bornes: élaguer si impossible de faire mieux
- ▶ Approximations: débiter avec approx. comme meilleure sol.
- ▶ Si tout échoue: solveurs SAT ou d'optimisation

Problème des n dames

- ▶ But: placer n dames sur échiquier sans attaques
- ▶ Algo.: placer une dame par ligne en essayant colonnes dispo.

Sac à dos

- ▶ But: maximiser valeur sans excéder capacité
- ▶ Algo.: essayer sans et avec chaque objet
- ▶ Mieux: élaguer dès qu'il y a excès de capacité
- ▶ Mieux++: élaguer si aucune amélioration avec somme valeurs

Retour de monnaie

- ▶ But: rendre montant avec le moins de pièces
- ▶ Algo.: pour chaque pièce, essayer d'en prendre 0 à # max.

7. Programmation dynamique

Approche

- ▶ *Principe d'optimalité*: solution optimale obtenue en combinant solutions de sous-problèmes qui se chevauchent
- ▶ *Descendante*: algo. récursif + mémoïsation (ex. Fibonacci)
- ▶ *Ascendante*: remplir tableau itér. avec solutions sous-prob.

Retour de monnaie

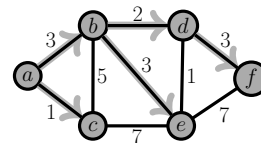
- ▶ *Sous-question*: # pièces pour rendre j avec pièces 1 à i ?
- ▶ *Identité*: $T[i, j] = \min(T[i-1, j], T[i, j-s[i]] + 1)$
- ▶ *Exemple*: montant $m = 10$ et pièces $s = [1, 5, 7]$

	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10
2	0	1	2	3	4	1	2	3	4	5	2
3	0	1	2	3	4	1	2	1	2	3	2

Sac à dos

- ▶ *Sous-question*: val. max. avec capacité j et les objets 1 à i ?
- ▶ *Identité*: $T[i, j] = \max(T[i-1, j], T[i-1, j-p[i]] + v[i])$

Plus courts chemins



- ▶ *Déf.*: chemin simple de poids minimal
- ▶ *Bien défini*: si aucun cycle négatif
- ▶ *Approche générale*: raffiner distances partielles itérativement
- ▶ *Dijkstra*: raffiner en marquant sommet avec dist. min.
- ▶ *Floyd-Warshall*: raffiner via sommet intermédiaire v_k
- ▶ *Bellman-Ford*: raffiner avec $\geq 1, 2, \dots, |V| - 1$ arêtes
- ▶ *Sommaire*:

	Dijkstra	Bellman-Ford	Floyd-Warshall
Types de chemins	d'un sommet vers les autres	paires de sommets	
Poids négatifs?	✗	✓	✓
Temps d'exécution	$\mathcal{O}(V \log V + E)$	$\Theta(V \cdot E)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(1)$)	$\mathcal{O}(V \log V)$	$\Theta(V)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(V)$)	$\mathcal{O}(V \log V)$	$\Theta(V ^2)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(V ^2)$)	$\mathcal{O}(V ^2)$	$\Theta(V ^3)$	$\Theta(V ^3)$

8. Algorithmes et analyse probabilistes

Modèle probabiliste

- ▶ *Modèle*: on peut tirer à pile ou face (non déterministe)
- ▶ *Aléa*: on peut obtenir une loi uniforme avec une pièce
- ▶ *Idéalisé*: on suppose avoir accès à une source d'aléa parfaite (en pratique: source plutôt pseudo-aléatoire)

Algorithmes de Las Vegas

- ▶ *Temps*: varie selon les choix probabilistes
- ▶ *Valeur de retour*: toujours correcte
- ▶ *Exemple*: tri rapide avec pivot aléatoire
- ▶ *Temps espéré*: dépend de $\mathbb{E}[Y_x]$ où $Y_x = \#$ opér. sur entrée x

Algorithmes de Monte Carlo

- ▶ *Temps*: borne ne varie pas selon les choix probabilistes
- ▶ *Valeur de retour*: pas toujours correcte
- ▶ *Exemple*: algorithme de Karger
- ▶ *Prob. d'erreur*: dépend de $\Pr(Y_x \neq \text{bonne sortie sur } x)$

Coupe minimum: algorithme de Karger

- ▶ *Coupe*: partition (X, Y) des sommets d'un graphe non dirigé
- ▶ *Taille*: # d'arêtes qui traversent X et Y
- ▶ *Coupe min.*: identifier la taille minimale d'une coupe
- ▶ *Algorithme*: contracter itérativement une arête aléatoire en gardant les multi-arêtes, mais pas les boucles



- ▶ *Prob. d'erreur*: $\leq 1 - 1/|V|^2$ (Monte Carlo)
- ▶ *Amplification*: on peut réduire (augmenter) la prob. d'erreur (de succès) arbitrairement (en général: avec min., maj., \vee , etc.)

Temps moyen

- ▶ *Temps moyen*: $\sum(\text{temps instances de taille } n) / \#$ instances
- ▶ *Attention*: pas la même chose que le temps espéré
- ▶ *Hypothèse*: entrées distribuées uniformément (\pm réaliste)
- ▶ *Exemple*: $\Theta(n^2)$ pour le tri par insertion