

IFT436 – Algorithmes et structures de données  
Université de Sherbrooke

## Examen final

Enseignant: Michael Blondin  
Date: mardi 19 décembre 2023  
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 du 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 **4 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,  $s = [s[1], s[2], \dots, s[n]]$  où  $n = |s|$ ;
- Si la **base d'un logarithme** n'est pas spécifiée, il s'agit de la base 2: « log » dénote «  $\log_2$  »;
- Les symboles  $/$  et  $\div$  désignent la division exacte et entière, par ex.  $7/2 = 3,5$  et  $7 \div 2 = 3$ .

### Question 1: analyse d'algorithmes récursifs

Nous disons qu'une séquence  $s$  de  $n$  entiers est **satisfaisable** s'il existe une séquence  $x \in \{-1, 0, 1\}^n$  telle que

$$x[1] \cdot s[1] + x[2] \cdot s[2] + \dots + x[n] \cdot s[n] = 1.$$

Par exemple,  $[4, 10, 3, 5]$  est satisfaisable car  $-4 + 10 - 5 = 1$ , mais  $[3, 5]$  n'est pas satisfaisable car les seules valeurs possibles sont  $\{0, \pm 2, \pm 3, \pm 5, \pm 8\}$ . Cet algorithme récursif détermine si une séquence est satisfaisable:

---

**Entrées:** séquence  $s$  de  $n \in \mathbb{N}_{\geq 1}$  entiers

**Résultat:**  $s$  est satisfaisable?

résoudre( $s$ ):

```

chercher-solution( $k, valeur$ ):
  si  $k = n + 1$  alors
    retourner ( $valeur = 1$ )
  sinon
     $p \leftarrow$  chercher-solution( $k + 1, valeur - s[k]$ )           // sat. avec  $x[k] = -1?$ 
     $q \leftarrow$  chercher-solution( $k + 1, valeur$ )                 // sat. avec  $x[k] = 0?$ 
     $r \leftarrow$  chercher-solution( $k + 1, valeur + s[k]$ )         // sat. avec  $x[k] = 1?$ 
    retourner  $p \vee q \vee r$ 
retourner chercher-solution(1,0)

```

---

- (a) Donnez une récurrence linéaire  $t$  où  $t(n)$  est le nombre d'opérations élémentaires effectuées par la procédure « résoudre » par rapport à  $n$ . Considérez ces opérations élémentaires: 2,5 pts
- arithmétique: +, −, etc.; — affectations; — accès au  $k^{\text{ème}}$  élément d'une séquence.
  - comparaisons: =, ≠, etc.; — opérations logiques: ∨, ∧, 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  $p = \text{faux}$ ).
- (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) Peut-on rendre l'algorithme plus rapide dans le meilleur cas avec une forme d'élagage? Justifiez. 2,5 pts

### Question 2: diviser-pour-régner

Rappelons qu'une séquence  $s$  de  $n$  éléments possède une **valeur majoritaire**  $x$  si  $s$  contient  $> n/2$  occurrences de  $x$ . Par exemple,  $[42, 0, 42, 42, 0]$  possède la valeur majoritaire 42, mais  $[42, 0, 42, 0]$  n'en possède pas.

- (a) Complétez la procédure ci-dessous dont le but est d'identifier une valeur majoritaire. Identifiez le temps d'exécution de l'algorithme complet. Au besoin, vous pouvez supposer que  $n$  est une puissance de 2. Justifiez. 5 pts

---

**Entrées:** séquence  $s$  de  $n \in \mathbb{N}_{\geq 1}$  éléments comparables

**Résultat:** la valeur majoritaire de  $s$  si elle en contient une, « aucune » sinon

maj-div( $s$ ):

```

si  $n = 1$  alors
  | /* Complétez cette ligne */
sinon
  |  $a \leftarrow \text{maj-div}(s[1:n \div 2])$ 
  | /* Complétez ce segment */
  | /* de pseudocode (sur autant de lignes que nécessaire) */

```

---

- (b) Supposons que  $s$  soit triée en ordre croissant, et que vous ayez accès à une sous-routine première-occ( $s, x$ ) qui retourne la position de la première occurrence de  $x$  dans  $s$ . Complétez cette procédure: 2 pts

---

**Entrées:** séquence  $s$  de  $n \in \mathbb{N}_{\geq 1}$  éléments comparables, triée en ordre croissant

**Résultat:** la valeur majoritaire de  $s$  si elle en contient une, « aucune » sinon

maj-triée( $s$ ):

```

 $i \leftarrow \text{première-occ}(s, \text{????})$  // 1ère occ. de la seule valeur possiblement maj.
si  $s[i + \text{????}] = s[i]$  alors retourner  $s[i]$  // Suffisamment d'occ. de cette valeur?
sinon retourner aucune

```

---

- (c) Implémentez première-occ afin que son temps d'exécution appartienne à  $\mathcal{O}(\log n)$ . Justifiez son temps. 5 pts

---

**Entrées:** séquence  $s$  de  $n \in \mathbb{N}_{\geq 1}$  éléments comparables, triée en ordre croissant, valeur  $x$

**Résultat:** position de la première occurrence de  $x$  dans  $s$ , « aucune » si aucune occurrence

première-occ( $s, x$ ):

```

fouille( $i, j$ ):
  | /* Complétez le pseudocode (sur autant de lignes que nécessaire) */
retourner fouille(1,  $n$ );

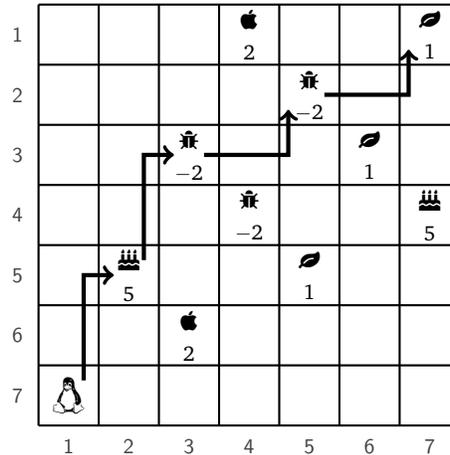
```

---

**Question 3: force brute et programmation dynamique**

Considérons un jeu où Tux  bondit sur une grille  $n \times n$  selon ces deux types de mouvements:

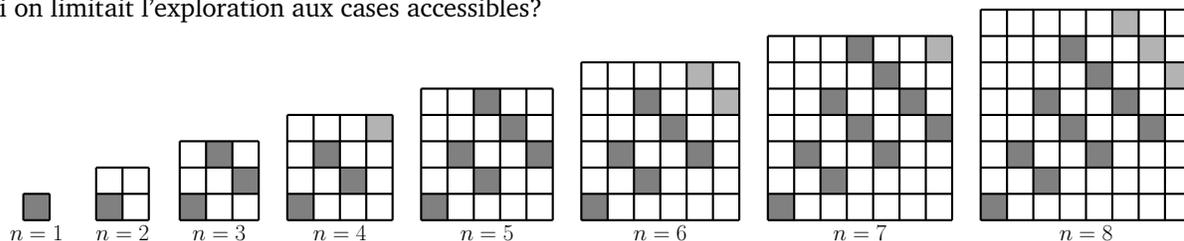
Tux débute sur la case tout en bas à gauche. Il obtient  $T[i, j]$  points lorsqu'il tombe sur la case  $(i, j)$ . Il cherche à se rendre à la case en haut à droite tout en maximisant son pointage. Par exemple, sur la grille ci-dessous, Tux peut suivre le chemin illustré, ce qui lui donne  $5 + (-2) + (-2) + 1 = 2$  points.



Clarifications:

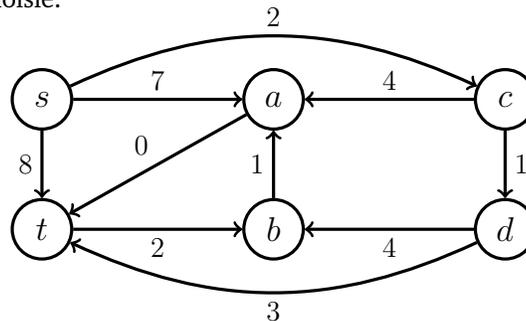
- La case de départ vaut toujours 0 point;
- Dans l'exemple ci-dessus, les cases vides valent 0 point; ce n'est pas écrit pour éviter de polluer l'illustration;
- Il est possible que Tux soit incapable d'atteindre la case de fin, auquel cas le pointage maximal est  $-\infty$ .

- (a) Quel est le pointage maximal pour la grille ci-dessus? (Aucune justification n'est requise.) 1 pt
- (b) Afin d'identifier le pointage maximal, 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
- (c) Donnez un algorithme, sous forme de pseudocode, qui identifie le pointage maximal en temps  $\mathcal{O}(n^2)$ . 6 pts
- (d) Plusieurs cases d'une grille  $n \times n$  ne sont pas accessibles. Par exemple, les cases accessibles pour  $n \in \{1, \dots, 8\}$  sont illustrées ci-dessous. Pourrions-nous obtenir un temps d'exécution inférieur à  $\mathcal{O}(n^2)$ , dans le pire cas, si on limitait l'exploration aux cases accessibles? 2 pts

**Question 4: plus courts chemins**

- (a) Reconsidérons le problème de la question 3. Soit  $\mathcal{G}$  le graphe dirigé où chaque sommet correspond à une case de la grille, et où une arête  $(u, v)$  apparaît ssi on peut se déplacer de la case  $u$  à la case  $v$  en un déplacement.
- (i) Comment pouvons-nous utiliser l'algorithme de Bellman-Ford afin d'obtenir le pointage maximal? 2 pts
- (ii) Quel est le temps d'exécution dans le pire cas, par rapport à  $n$ ? 1,5 pts
- (iii) Dans quel ordre devrait-on considérer les sommets pour réduire le temps d'exécution? 1,5 pts

- (b) Exécutez l'algorithme de Dijkstra sur le graphe pondéré ci-dessous afin de déterminer la longueur d'un plus court chemin de  $s$  vers  $t$ . Arrêtez l'exécution dès que possible. Laissez une trace des itérations effectuées en indiquant les sommets marqués et les distances partielles identifiées. Expliquez brièvement pourquoi vous pouvez arrêter à l'itération choisie. 3 pts



Remarque: ce n'est pas nécessaire de recopier le graphe plusieurs fois, vous pouvez numéroté l'ordre des calculs/marquages.

### Question 5: algorithmes probabilistes

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

- $n \geq 2$  et  $n$  est pair,
- $s$  contient  $n/2$  occurrences d'un même nombre  $a$ ,
- $s$  contient  $n/2$  occurrences d'un même nombre  $b$  tel que  $b \neq a$ .

Par exemple, la séquence  $s = [42, 1, 1, 42, 1, 42]$  de taille  $n = 6$  est balancée. Considérons les deux algorithmes ci-dessous qui cherchent à vérifier si la valeur maximale d'une séquence balancée est paire. Par exemple, sur la séquence  $s$  précédente, on cherche à obtenir *vrai* car 42 est pair.

---

**Entrée:** séquence balancée  $s$  de taille  $n$

**Résultat:**  $\max(s)$  est pair?

max-paritéA( $s$ ):

```

choisir  $i \in [1..n]$  aléatoirement de façon uniforme
choisir  $j \in [1..n]$  aléatoirement de façon uniforme
retourner  $\max(s[i], s[j]) \bmod 2 = 0$  //  $\max(s[i], s[j])$  est pair?
  
```

---

**Entrée:** séquence balancée  $s$  de taille  $n$

**Résultat:**  $\max(s)$  est pair?

max-paritéB( $s$ ):

```

faire
  | choisir  $i \in [1..n]$  aléatoirement de façon uniforme
  | choisir  $j \in [1..n]$  aléatoirement de façon uniforme
tant que  $s[i] = s[j]$ 
retourner  $\max(s[i], s[j]) \bmod 2 = 0$  //  $\max(s[i], s[j])$  est pair?
  
```

---

Pour chacun des algorithmes max-paritéA et max-paritéB:

- (a) Dites s'il est de Las Vegas ou de Monte Carlo. Expliquez pourquoi. 3 pts
- (b) Selon votre réponse en (a), identifiez le temps espéré ou la probabilité d'erreur de l'algorithme. Justifiez. 7 pts

Si la probabilité d'erreur est non nulle, alors expliquez comment la réduire en appelant l'algorithme trois fois (en boîte noire). Calculez la nouvelle probabilité d'erreur.

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

## 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  $\lambda_i$  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

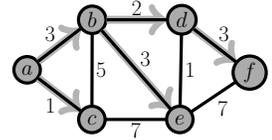
- ▶ *Sous-question*: # pièces pour rendre  $j$  avec pièces 1 à  $i$ ?
- ▶ *Identité*:  $T[i, j] = \min(T[i-1, j], T[i, j-s[i]] + 1)$
- ▶ *Exemple*: montant  $m = 10$  et pièces  $s = [1, 5, 7]$

	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