

# IFT436 – Algorithmes et structures de données

## Université de Sherbrooke

### Examen final

Enseignant: Michael Blondin  
 Date: jeudi 15 décembre 2022  
 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 dernière page 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 **10 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

Considérons un ordinateur, constitué de 4 processeurs, sur lequel nous cherchons à exécuter  $n$  tâches. La tâche  $i$  se complète en  $t[i]$  millisecondes. Par exemple, si  $t = [5, 1, 3, 2, 8, 4]$  et que l'assignation des temps de tâches aux processeurs est  $p = [8, 5, 3 + 4, 1 + 2]$ , alors toutes les tâches sont complétées en  $\max(8, 5, 3 + 4, 1 + 2) = 8$  millisecondes. Cet algorithme identifie le temps minimal d'exécution des tâches parmi toutes les assignations:

---

**Entrées:** une séquence  $t$  de  $n \geq 0$  entiers positifs

**Résultat:** temps minimal pour compléter toutes les tâches sur un ordinateur à 4 processeurs

temps-min( $t$ ):

```

aux( $i, p$ ):
  si  $i = n + 1$  alors
    retourner  $\max(\max(p[1], p[2]), \max(p[3], p[4]))$  // temps du processeur le + chargé
  sinon
     $m \leftarrow \infty$ 
    pour  $j \in [1..4]$ 
       $p[j] \leftarrow p[j] + t[i]$  // assigner tâche  $i$  au processeur  $j$ 
       $m \leftarrow \min(m, \text{aux}(i + 1, p))$  // compléter assignation récursivement
       $p[j] \leftarrow p[j] - t[i]$  // rétablir processeur  $j$ 
    retourner  $m$ 
retourner aux(1, [0, 0, 0, 0])

```

---

(a) Donnez une récurrence linéaire  $t$  où  $t(n)$  dénote le nombre d'opérations élémentaires exécutées par l'appel initial «  $\text{aux}(1, [0, 0, 0, 0])$  » par rapport à  $n$ . Considérez ces opérations élémentaires: 2,5 pts

- Arithmétique: +, −, max, min, etc.
- Affectations de la forme «  $\text{var} \leftarrow \dots$  » ou «  $\text{seq}[k] \leftarrow \dots$  »
- Comparaisons: =, ≥, etc.
- Accès aux éléments d'une séquence (en lecture).

$$t(n) = \begin{cases} 9 & \text{si } n = 0, \\ 4 \cdot t(n-1) + \underbrace{47}_{3+4 \cdot 11} & \text{sinon.} \end{cases}$$

(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  $t(n) - 4 \cdot t(n-1) = 47 \cdot 1^n$  mène au polynôme  $(x^1 - 4)(x - 1)$ . La forme close est donc  $t(n) = c_1 \cdot 4^n + c_2 \cdot 1^n$ . On obtient le système:

$$\begin{aligned} t(0) &= 9 = c_1 + c_2 \\ t(1) &= 83 = 4c_1 + c_2 \end{aligned}$$

(c) Expliquez comment améliorer l'algorithme à l'aide d'une forme d'élagage. 2,5 pts

Sol. 1: on stocke la meilleure valeur obtenue; et on cesse d'explorer dès que  $\max(p)$  l'excède.

Sol. 2: on cesse d'explorer si  $\max(p) > \lceil n/4 \rceil \cdot \max(t)$ , car la solution optimale est forcément meilleure.

## Question 2: diviser-pour-régner

Considérons des séquences d'éléments de  $\mathbb{N} \times \{a, b\}$ , par ex.  $s = [(4, a), (3, b), (1, b), (2, a)]$ . Nous cherchons à trier de telles séquences sous l'ordre défini par  $(x, y) \preceq (x', y')$  ssi  $(y = a \wedge y' = b) \vee (y = y' \wedge x \leq x')$ . En mots: nous priorisons les paires qui contiennent «  $a$  » avant celles avec «  $b$  ». Par exemple, sur entrée  $s$ , la sortie attendue est  $[(2, a), (4, a), (1, b), (3, b)]$ .

(a) Complétez ce pseudocode:

7 pts

**Entrée:** une séquence  $s$  d'éléments de  $\mathbb{N} \times \{a, b\}$ **Résultat:**  $s$  triée sous  $\preceq$ trier( $s$ ):

```

si  $|s| \leq 1$  alors
  | retourner  $s$ 
sinon
  |  $m \leftarrow |s| \div 2$ 
  |  $g \leftarrow \text{trier}(s[1:m])$ 
  | /* Complétez ce segment */
  | /* (nécessite plusieurs lignes de pseudocode) */

```

**Entrée:** une séquence  $s$  d'éléments de  $\mathbb{N} \times \{a, b\}$ **Résultat:**  $s$  triée sous  $\preceq$ trier( $s$ ):

```

si  $|s| \leq 1$  alors
  | retourner  $s$ 
sinon
  |  $m \leftarrow |s| \div 2$ 
  |  $g \leftarrow \text{trier}(s[1:m])$ 
  |  $d \leftarrow \text{trier}(s[m+1:|s|])$ 
  |  $i \leftarrow 1; j \leftarrow 1$ 
  |  $t \leftarrow []$ 
  | tant que  $i \leq |g| \wedge j \leq |d|$  // fusion de  $g$  et  $d$  dans  $t$ 
  |   |  $(x, y) \leftarrow g[i]$ 
  |   |  $(x', y') \leftarrow d[j]$ 
  |   | si  $(y = a \wedge y' = b) \vee (y = y' \wedge x \leq x')$  alors //  $(x, y) \preceq (x', y')$ ?
  |   |   | ajouter  $(x, y)$  à  $t$ 
  |   |   |  $i \leftarrow i + 1$ 
  |   | sinon
  |   |   | ajouter  $(x', y')$  à  $t$ 
  |   |   |  $j \leftarrow j + 1$ 
  |   | retourner  $t + g[i:|g|] + d[j:|d|]$  // ajouter éléments restants

```

(b) Complétez ce pseudocode:

2 pts

---

**Entrées:** une séquence  $s$  d'éléments de  $\mathbb{N} \times \{a, b\}$  déjà triée sous  $\preceq$ , et  $(x, y) \in \mathbb{N} \times \{a, b\}$   
**Résultat:** vrai si  $s$  contient  $(x, y)$ , faux sinon

```

contient( $s, (x, y)$ ):
  contient'( $lo, hi$ ):
    si  $lo = hi$  alors
      retourner ( $s[lo] = (x, y)$ )
    sinon
       $i \leftarrow (lo + hi) \div 2$ 
       $(x', y') \leftarrow s[i]$ 
      si ...ligne 1... alors /* à compléter */
        retourner contient'( $lo, i$ )
      sinon
        ...ligne 2... /* à compléter */
  retourner contient'(1,  $|s|$ )

```

---



---

**Entrées:** une séquence  $s$  d'éléments de  $\mathbb{N} \times \{a, b\}$  déjà triée sous  $\preceq$ , et  
 $(x, y) \in \mathbb{N} \times \{a, b\}$   
**Résultat:** vrai si  $s$  contient  $(x, y)$ , faux sinon

```

contient( $s, (x, y)$ ):
  contient'( $lo, hi$ ):
    si  $lo = hi$  alors
      retourner ( $s[lo] = (x, y)$ )
    sinon
       $i \leftarrow (lo + hi) \div 2$ 
       $(x', y') \leftarrow s[i]$ 
      si  $(y = a \wedge y' = b) \vee (y = y' \wedge x \leq x')$  alors
        retourner contient'( $lo, i$ )
      sinon
        retourner contient'( $i + 1, hi$ )
  retourner contient'(1,  $|s|$ )

```

---

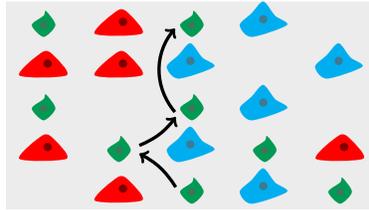
(c) Quel est le temps d'exécution asymptotique des algorithmes obtenus en (a) et (b) dans le pire cas? Vous pouvez supposer que  $|s|$  est une puissance de deux. Justifiez. 3 pts

(a) Le temps s'exprime par  $t(n) = 2 \cdot t(n \div 2) + f(n)$  où  $f \in \mathcal{O}(n)$ . Par le théorème maître, avec  $b = c = 2$  et  $d = 1$ , on obtient  $t \in \mathcal{O}(n \log n)$ .

(b) Le temps s'exprime par  $t(n) = t(n \div 2) + f(n)$  où  $f \in \mathcal{O}(1)$ . Par le théorème maître, avec  $b = 2$ ,  $c = 1$  et  $d = 0$ , on obtient  $t \in \mathcal{O}(\log n)$ .

### Question 3: force brute et programmation dynamique

Lassé de descendre des montagnes, Tux  s'intéresse maintenant à la grimpe. Un mur d'escalade est formé de prises colorées posées sur une grille  $n \times n$ . Tux débute sur une prise de son choix au bas du mur et doit se rendre à une prise au sommet du mur en utilisant *toujours des prises d'une même couleur*. En raison de sa morphologie, Tux bondit vers le haut: d'une case en diagonale ou d'exactement deux cases verticalement (pas moins). Tux aimerait compléter une grimpe qui minimise le nombre de bonds. Par exemple, celle-ci utilise trois bonds:



Remarque: cet exemple illustre les trois seuls types de bonds possibles.

Formellement, un mur est représenté par un tableau bidimensionnel  $T$ , où l'entrée  $T[i, j]$  contient un caractère qui représente une couleur, ou  $\perp$  s'il n'y a pas de prise. Par exemple, le mur ci-dessus est représenté par:

$$T = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & V & R & V & B & \perp \\ 2 & R & R & B & \perp & B \\ 3 & V & \perp & V & B & \perp \\ 4 & R & V & B & V & R \\ 5 & \perp & R & V & B & V \end{array}$$

- (a) Afin d'identifier la quantité minimale de bonds, 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

Non.

*Sol. 1:* Il y a au moins 2 façons de monter de deux cases: verticalement, ou en combinant deux bonds en diagonale. Ainsi, s'il y a des prises rouges partout, il y a  $\geq 2^{n/2}$  parcours pour chaque colonne.

*Sol. 2:* Il y a 3 façons de monter de deux cases sur une colonne qui n'est pas sur les côtés: verticalement, ou en combinant deux bonds en diagonale. Ainsi, s'il y a des prises rouges partout, il y a  $\geq 3^{n/2}$  parcours pour chaque telle colonne.

*Sol. 3:* Sur un mur de la forme ci-dessous, le nombre de parcours est décrit par la suite de Fibonacci  $f(n) = f(n-1) + f(n-2)$ , qui est exponentielle.



*Sol. 4:* Sur un mur de la forme ci-dessous, où  $n = 6k - 1$ , le nombre de parcours est  $2^k = 2^{(n+1)/6}$ .



- (b) Quel est le nombre de bonds minimal pour le mur ci-dessus? Justifiez.

2 pts

Tux peut monter en 2 bonds verticaux sur la colonne du centre. Comme il faut franchir 4 lignes et qu'on ne peut pas franchir plus de 2 lignes à la fois, c'est une montée optimale.

(c) Donnez un algorithme, sous forme de pseudocode, qui identifie le nombre de bonds minimal en temps  $\mathcal{O}(n^2)$ . 7 pts

*Remarque: dans le doute, donnez au moins une description textuelle pour m'aider à vous donner des points.*

Sol. 1:

```

initialiser un tableau  $B[1..n, 1..n]$  avec  $\infty$ 
pour  $j \in [1..n]$ 
| si  $T[n, j] \neq \perp$  alors  $B[n, j] \leftarrow 0$ 
pour  $i \in [n - 1..1]$ 
| pour  $j \in [1..n]$ 
| |  $a \leftarrow \infty; b \leftarrow \infty; c \leftarrow \infty$ 
| | si  $(j - 1 \geq 1) \wedge (T[i, j] = T[i + 1, j - 1])$  alors // bond ↗
| | |  $a \leftarrow B[i + 1, j - 1]$ 
| | si  $(i + 2 \leq n) \wedge (T[i, j] = T[i + 2, j])$  alors // bond ↑
| | |  $b \leftarrow B[i + 2, j]$ 
| | si  $(j + 1 \leq n) \wedge (T[i, j] = T[i + 1, j + 1])$  alors // bond ↘
| | |  $c \leftarrow B[i + 1, j + 1]$ 
| |  $B[i, j] \leftarrow 1 + \min(a, b, c)$ 
retourner  $\min B[1]$  // min. de la 1ère ligne de  $B$ 

```

Sol. 2: Même chose de façon plus concise:

```

initialiser un tableau  $B[1..n, 1..n]$  avec  $\infty$ 
pour  $j \in [1..n]$ 
| si  $T[n, j] \neq \perp$  alors  $B[n, j] \leftarrow 0$ 
pour  $i \in [n - 1..1]$ 
| pour  $j \in [1..n]$ 
| | pour  $(x, y) \in [(1, -1), (2, 0), (1, 1)]$  // bonds [↗, ↑, ↘]
| | | si  $(1 \leq i + x \leq n) \wedge (1 \leq j + y \leq n) \wedge (T[i, j] = T[i + x, j + y])$  alors
| | | |  $B[i, j] \leftarrow \min(B[i, j], 1 + B[i + x, j + y])$ 
retourner  $\min B[1]$  // min. de la 1ère ligne de  $B$ 

```

Sol. 3:

initialiser un tableau associatif  $B$

aux( $i, j, c$ ):

```

  si  $\neg(1 \leq i \leq n \wedge 1 \leq j \leq n)$  alors retourner  $\infty$ 
  sinon si  $T[i, j] \neq c$           alors retourner  $\infty$ 
  sinon si  $i = 1$                 alors retourner 0
  sinon si  $B$  contient  $(i, j)$    alors retourner  $B[i, j]$ 
  sinon
     $B[i, j] \leftarrow 1 + \min(\text{aux}(i-1, j-1, c), \text{aux}(i-2, j, c), \text{aux}(i-1, j+1, c))$ 
    retourner  $B[i, j]$ 

```

$m \leftarrow \infty$

pour  $j \in [1..n]$

```

  si  $T[n, j] \neq \perp$  alors
     $m \leftarrow \min(m, \text{aux}(n, j, T[n, j]))$ 

```

retourner  $m$

Sol. 4–6: Même chose, mais dans l'autre sens du mur.

#### Question 4: plus courts chemins

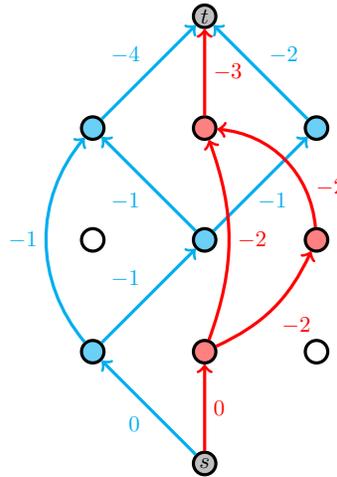
- (a) Reconsidérons le problème de la question 3. Supposons que chaque prise possède une valeur entière positive qui représente sa difficulté. Par exemple, si Tux utilise trois prises rouges de difficultés  $[2, 2, 3]$ , alors il obtient un score de 7. Tux aimerait compléter une grimpe qui maximise son score (sans se soucier de minimiser le nombre de bonds). 5 pts

Expliquez comment identifier le score maximal en construisant un graphe, puis en invoquant l'algorithme de Dijkstra, Bellman-Ford ou Floyd-Warhsall. Vous pouvez illustrer votre approche à l'aide de l'exemple ci-dessous (sans nécessairement le résoudre). Quel est le temps d'exécution asymptotique de votre approche dans le pire cas par rapport à  $n$ ?

		1	2	3
$T =$	1	B 4	R 3	B 2
	2	$\perp$ $\perp$	B 1	R 2
	3	B 1	R 2	$\perp$ $\perp$

On associe un sommet à chaque case  $(i, j)$ . On relie chaque sommet  $(i, j)$  vers les cases  $(i', j')$  qu'il peut atteindre, avec un poids  $-T[i, j]$ . On ajoute deux sommets  $s$  et  $t$ . Il y a une arête de poids nul de  $s$  vers toutes les cases de la forme  $(n, j)$ . Il y a une arête de poids  $-T[1, j]$  de chaque case de la forme  $(1, j)$  vers  $t$ . On utilise l'algorithme de Bellman-Ford (car poids négatifs) afin d'identifier un plus court chemin de  $s$  vers  $t$ . Le temps appartient à  $\mathcal{O}(n^2 \cdot n^2) = \mathcal{O}(n^4)$ .

Sur l'exemple précédent, on construit:



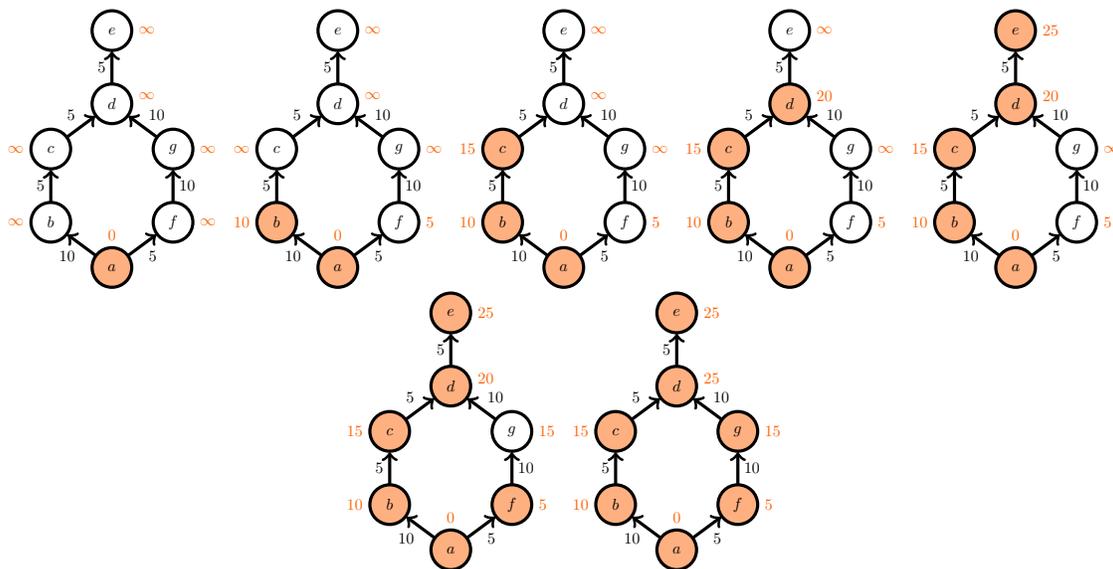
*Commentaire 1:* Dans un langage bien plus formel qu'attendu, le graphe de la solution précédente est ce graphe dirigé  $\mathcal{G} := (V, E)$  pondéré par  $p$ :

$$V := \{v_{i,j} : i, j \in [1..n]\} \cup \{s, t\},$$

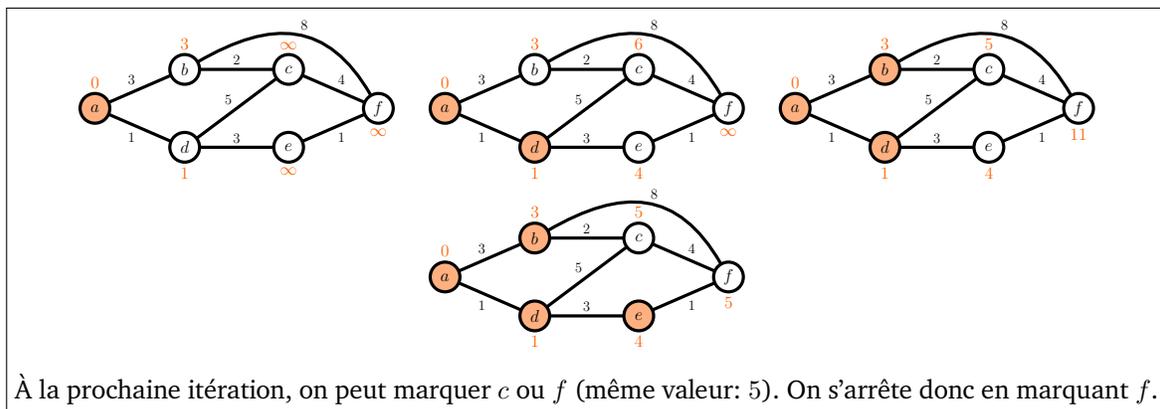
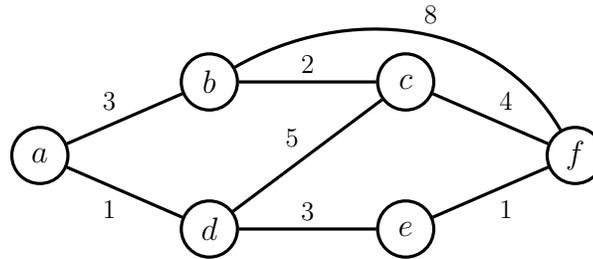
$$E := \{(v_{i,j}, v_{i',j'}) : \text{permis de bondir de } (i, j) \text{ à } (i', j')\} \cup \{(s, v_{n,j}) : j \in [1..n]\} \cup \{(v_{1,j}, t) : j \in [1..n]\},$$

$$p := \{(v_{i,j}, v_{i',j'}) \mapsto -T[i, j]\} \cup \{(s, v_{n,j}) \mapsto 0\} \cup \{(v_{1,j}, t) \mapsto -T[1, j]\}.$$

*Commentaire 2:* L'algorithme de Dijkstra modifié avec « max » plutôt que « min » ne fonctionne pas. Voici un contre-exemple (possible en tant que mur d'escalade), où l'algorithme modifié identifie une distance de 25 de  $a$  vers  $e$ , alors qu'il existe un chemin de poids 30:



- (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



### 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 *débalancée* si:

- $n \geq 3$  et  $n$  est impair,
- $s$  contient  $(n + 1)/2$  occurrences d'un même nombre,
- $s$  contient  $(n - 1)/2$  autres nombres distincts qui apparaissent chacun exactement une fois.

Par exemple, la séquence  $s = [2, 2, 1, 2, 3, 2, 4]$  de taille  $n = 7$  est débalancée. Dans une séquence débalancée, il y a toujours un élément majoritaire, par ex. 2 dans la séquence précédente. Considérons ces deux algorithmes probabilistes qui cherchent à identifier l'élément majoritaire d'une séquence débalancée:

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

**Résultat:**  $\text{maj}(s)$

$\text{majoritéA}(s)$ :

```

faire
  | choisir  $i \in [1..n]$  aléatoirement de façon uniforme
  | choisir  $j \in [1..n]$  aléatoirement de façon uniforme
  tant que  $\neg(i \neq j \wedge s[i] = s[j])$  // deux éléments égaux?
  retourner  $s[i]$ 

```

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

**Résultat:**  $\text{maj}(s)$

$\text{majoritéB}(s)$ :

```

choisir  $i \in [1..n]$  aléatoirement de façon uniforme
choisir  $j \in [1..n] \setminus \{i\}$  aléatoirement de façon uniforme //  $n-1$  choix car  $i$  exclu
si  $s[j] = s[i]$  alors // deux éléments égaux?
  | retourner  $s[j]$ 
retourner  $s[1]$ 

```

Pour chacun des algorithmes  $\text{majoritéA}$  et  $\text{majoritéB}$ :

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

3 pts

$\text{majoritéA}$ : *Las Vegas* puisqu'il est correct (on quitte si on a identifié le nombre qui apparaît plusieurs fois), mais que la boucle peut être arbitrairement longue (par ex. en pigeant  $i = j$  continuellement).  
 $\text{majoritéB}$ : *Monte Carlo* car son temps est toujours de  $\mathcal{O}(1)$ , et car il peut se tromper (par ex. si  $s[1]$  n'est pas majoritaire et qu'on pige  $i, j$  tels que  $s[i] \neq s[j]$ ).

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

7 pts

$\text{majoritéA}$ : La probabilité de quitter la boucle est:

$$\frac{\overbrace{\frac{(n+1)/2}{n}}^{\text{piger } i \text{ maj.}} \cdot \overbrace{\frac{((n+1)/2) - 1}{n}}^{\text{piger } j \neq i \text{ maj.}}}{n} = \frac{(n+1)/2 \cdot (n-1)/2}{n^2} = \frac{n^2 - 1}{4n^2}.$$

Le nombre d'itérations espéré est donc de  $4n^2/(n^2 - 1)$  (loi géométrique). Puisque les opérations sont toutes élémentaires, le temps espéré est de  $\mathcal{O}(1)$ .

$\text{majoritéB}$ : Si  $s[1] = \text{maj}(s)$ , alors il n'y a pas d'erreur. Sinon, la probabilité d'obtenir un succès est

$$\underbrace{\frac{(n+1)/2}{n}}_{\text{piger } i \text{ maj.}} \cdot \underbrace{\frac{1}{2}}_{\text{piger } j \text{ maj.}} = \frac{n+1}{4n}.$$

La probabilité d'erreur est donc de  $1 - (n+1)/(4n)$ .

## 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

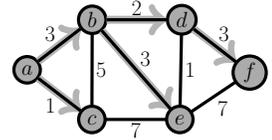
- ▶ *Sous-question*: # pièces pour rendre  $j$  avec pièces 1 à  $i$ ?
- ▶ *Identité*:  $T[i, j] = \min(T[i-1, j], T[i, j-s[i]] + 1)$
- ▶ *Exemple*: montant  $m = 10$  et pièces  $s = [1, 5, 7]$

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