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 8½" × 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 **8 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, $s = [s[1], s[2], \ldots, s[n]]$ 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

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

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 ← chercher-solution(k + 1, valeur − s[k])      // sat. avec x[k] = −1?
            q ← chercher-solution(k + 1, valeur)             // sat. avec x[k] =  0?
            r ← chercher-solution(k + 1, valeur + s[k])      // sat. avec x[k] =  1?
            retourner p ∨ q ∨ 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: $=$, $\neq$, 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 $p = \textit{faux}$).

Sol. 1.

$$t(n) = \begin{cases} 3 & \text{si } n = 0, \\ 3 \cdot t(n-1) + 14 & \text{si } n > 0. \end{cases}$$

Sol. 2.

$$t(n) = \begin{cases} 23 & \text{si } n = 1, \\ 3 \cdot t(n-1) + 14 & \text{si } n > 1. \end{cases}$$

(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

On a la récurrence linéaire non homogène $t(n) - 3 \cdot t(n-1) = 14 \cdot 1^n$ et ainsi le polynôme $(x-3)(x-1)$. La forme close est donc de la forme $t(n) = c_1 \cdot 3^n + c_2$. On peut identifier les constantes en résolvant:

$$\begin{aligned} t(0) &= \ \ 3 = \ \ c_1 + c_2 \\ t(1) &= 23 = 3c_1 + c_2 \end{aligned}$$

(c) Peut-on rendre l'algorithme plus rapide dans le meilleur cas avec une forme d'élagage? Justifiez. 2,5 pts

Sol. 1. Oui, par ex. on peut éviter d'évaluer q et r lorsque p est vrai, car on a trouvé une solution.

Sol. 2. On cesse d'explorer lorsque $\mathit{valeur} = 1$, car on peut choisir $x[k] = 0$ pour tous les k restants.

Sol. 3. Si $\mathit{valeur} > 1$ et les nombres négatifs restants ne permettent pas de descendre sous $-\mathit{valeur}+1$, alors on peut arrêter l'exploration car on ne pourra pas obtenir 1.

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 ← maj-div(s[1 : n ÷ 2])
        /* Complétez ce segment                                     */
        /*     de pseudocode (sur autant de lignes que nécessaire)  */
```

```
maj-div(s):
    si n = 1 alors
        retourner s[1]
    sinon
        a ← maj-div(s[1 : n ÷ 2])
        b ← maj-div(s[n ÷ 2 + 1 : n])
        c ← 0; c' ← 0                    // vérifier si a ou b est majoritaire
        pour x ∈ s
            si x = a alors  c ← c + 1
            si x = b alors  c' ← c' + 1
        si          c > n/2 alors  retourner a
        sinon si c' > n/2 alors  retourner b
        sinon                  retourner aucune
```

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

Remarque: si $a = b$, alors a est forcément majoritaire; si $a \neq b$, alors on ne peut pas conclure qu'il n'y a pas de majorité, par ex. pour $s = [10, 20, 20]$ nous avons $a = 10$ et $b = 20$, alors que b est majoritaire.

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

Cette solution ou des variantes équivalentes:

```
maj-triée(s):
    i ← première-occ(s, s[⌈n/2⌉])
    si s[i + ⌊n/2⌋] = s[i] alors  retourner s[i]
    sinon  retourner aucune
```

Remarque: on ne peut pas choisir $s[\lfloor n/2 \rfloor]$ dans l'appel à `première-occ`. Par exemple, si $s = [10, 10, 20, 20, 20]$, alors $s[\lfloor n/2 \rfloor] = s[2] = 10$, qui n'est pas la valeur majoritaire.

(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

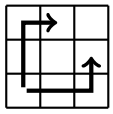
```
première-occ(s, x):
    fouille(i, j):
        /* Complétez le pseudocode (sur autant de lignes que nécessaire)          */
    retourner fouille(1, n);
```

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):
        si i = j alors
            si s[i] = x alors retourner i
            sinon          retourner aucune
        sinon
            m ← (i + j) ÷ 2
            si x ≤ s[m] alors  retourner fouille(i, m)
            sinon              retourner fouille(m + 1, j)
    retourner fouille(1, n);
```

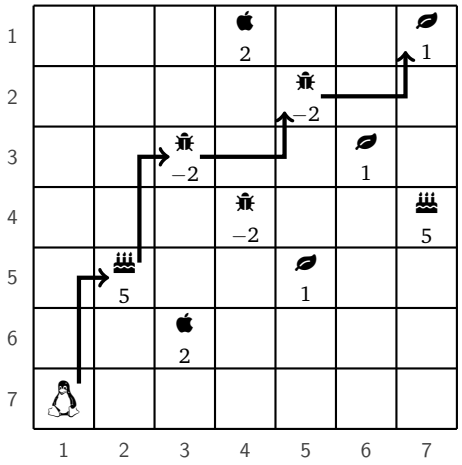
Le temps d'exécution s'exprime par $t(n) = t(n \div 2) + f(n)$ où $f \in \mathcal{O}(1)$. En invoquant le théorème maître avec $b = 1$, $c = 2$ et $d = 0$, on obtient $t \in \mathcal{O}(\log n)$.

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:

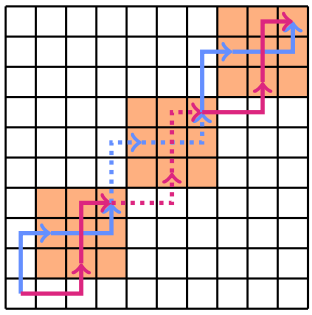
- *La case de départ vaut toujours* 0 *point;*
- *Dans l'exemple ci-dessus, les cases vides valent* 0 *point; ce n'est pas écrit pour éviter de polluer l'illustration;*
- *Il est possible que Tux soit incapable d'atteindre la case de fin, auquel cas le pointage maximal est* $-\infty$.

(a) Quel est le pointage maximal pour la grille ci-dessus? (*Aucune justification n'est requise.*) 1 pt

5 points.

(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

Non, il y a au moins $2^{\lfloor (n-1)/3 \rfloor} \in \Omega(2^{n/3})$ parcours:



Remarque: dire qu'il y a deux déplacements ne suffit pas; il faut montrer que des combinaisons peuvent être chaînées.

(c) Donnez un algorithme, sous forme de pseudocode, qui identifie le pointage maximal en temps $\mathcal{O}(n^2)$. 6 pts

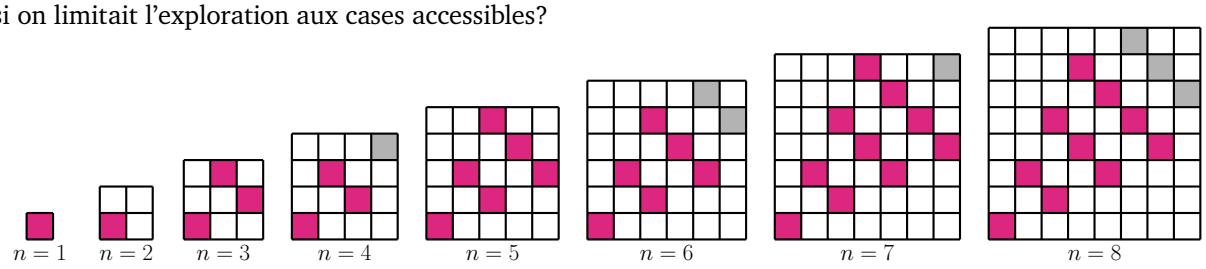
Sol. 1. Une solution itérative parmi d'autres:

```
pointage-max(T):
    initialiser tableau P[1..n, 1..n] avec −∞  // P[i,j] = pointage max. du départ vers (i,j)
    P[n, 1] ← 0
    pour i ∈ [n − 1..1]
        pour j ∈ [1..n]
            m ← −∞
            pour (x, y) ∈ [(−2, 1), (−1, 2)]          // meilleur prédecesseur de (i,j)?
                si (1 ≤ i − x ≤ n) ∧ (1 ≤ j − y ≤ n) alors
                    m ← max(m, P[i − x, j − y])
            P[i, j] ← m + T[i, j]      // P[i,j] = pointage meilleur préd. + points en (i,j)
    retourner P[1, n]
```

Sol. 2. Une solution récursive parmi d'autres:

```
pointage-max(T):
    initialiser tableau associatif vide P    // important pour que le temps soit polynomial
    aux(i, j):
        si (i, j) ∉ P alors
            si (i = 1) ∧ (j = n) alors  P[i, j] ← 0
            sinon
                a ← −∞; b ← −∞
                si (i − 2 ≥ 1) ∧ (j + 1 ≤ n) alors  a ← T[i − 2, j + 1] + aux(i − 2, j + 1)
                si (i − 1 ≥ 1) ∧ (j + 2 ≤ n) alors  b ← T[i − 1, j + 2] + aux(i − 1, j + 2)
                P[i, j] ← max(a, b)
        retourner P[i, j]
    retourner aux(n, 1)
```

(d) Plusieurs cases d'une grille $n \times n$ ne sont pas accessibles. Par exemple, les cases accessibles pour $n \in \{1, \ldots, 8\}$ 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



Non. Soit $f(n)$ le nombre de cases accessibles d'une grille $n \times n$. Nous avons:

$$\begin{aligned} f(n) &\geq 1 + 2 + \ldots + \lceil n/2 \rceil \\ &= (\lceil n/2 \rceil \cdot (\lceil n/2 \rceil + 1))/2 \\ &\geq ((n/2) \cdot (n/2))/2 \\ &= n^2/8 \\ &\in \Theta(n^2). \end{aligned}$$

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

Soit T la grille. On pondère chaque arête $((i, j), (i', j'))$ de $\mathcal{G}$ par $-T[i', j']$. On exécute l'algorithme de Bellman-Ford qui calcule un tableau d. On retourne $-d[(1, n)]$.

(ii) Quel est le temps d'exécution dans le pire cas, par rapport à n? 1,5 pts

$\Theta(n^4)$.

Raison: On a $|V| = n^2$. Chaque case a au plus deux successeurs. Toute case sur les $n-2$ premières lignes et colonnes, a deux successeurs. Donc, $2(n-2)^2 \leq |E| \leq 2n^2$ et ainsi $|E| \in \Theta(n^2)$. Le temps est donc de $\Theta(|V| \cdot |E|) = \Theta(n^4)$.

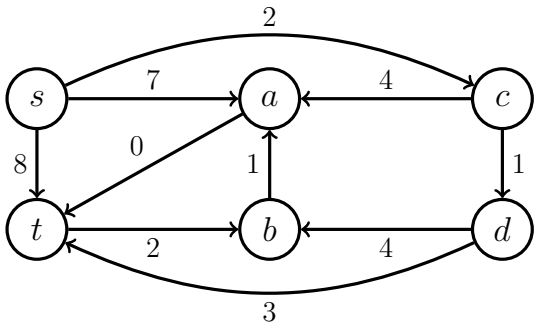
(iii) Dans quel ordre devrait-on considérer les sommets pour réduire le temps d'exécution? 1,5 pts

En ordre topologique, par ex. de gauche à droite, du bas vers le haut.

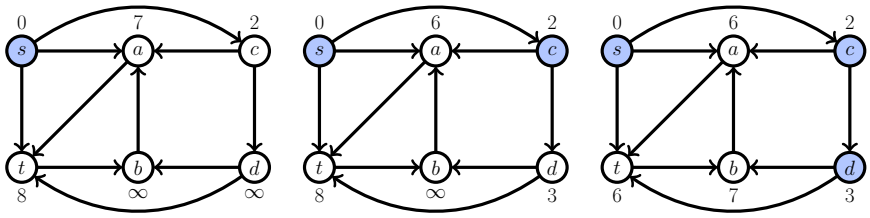
Raison: L'algorithme de Bellman-Ford propage les distances le long des arêtes. Ainsi, en suivant cet ordre, il trouvera les plus courts chemins après un seul tour de la boucle principale, donc en temps $\Theta(n^2)$.

Remarque: le pire cas s'atteint lorsqu'on considère les sommets de la fin vers le départ. Choisir les arêtes dans l'ordre (croissant ou décroissant) de leur poids peut également mener au pire cas, selon la répartition des poids.

(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 numéroter l'ordre des calculs/marquages.



À la prochaine itération, on pourrait marquer a ou t car ils sont tous deux minimaux. On peut donc s'arrêter ici en marquant t, ce qui garantit qu'il ne sera plus amélioré.

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 ∈ [1..n] aléatoirement de façon uniforme
    choisir j ∈ [1..n] aléatoirement de façon uniforme
    retourner max(s[i], s[j]) mod 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 ∈ [1..n] aléatoirement de façon uniforme
        choisir j ∈ [1..n] aléatoirement de façon uniforme
    tant que s[i] = s[j]
    retourner max(s[i], s[j]) mod 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

`max-paritéA`: *Monte Carlo* car son temps est toujours de $\mathcal{O}(1)$, et car il peut se tromper (par ex. si $\min(s)$ et $\max(s)$ n'ont pas la même parité et qu'on pige i, j tels que $s[i] = s[j] = \min(s)$).

`max-paritéB`: *Las Vegas* puisqu'il est correct (on quitte si on a identifié a et b), mais que la boucle peut être arbitrairement longue (par ex. en pigeant $i = j$ continuellement).

(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.

`max-paritéA`: Si $\min(s)$ et $\max(s)$ ont la même parité, alors il n'y a pas d'erreur. Sinon, il y a une erreur si on pige i, j tels que $s[i] = s[j] = \min(s)$. Cela se produit avec probabilité $(1/2) \cdot (1/2) = 1/4$. La probabilité d'erreur est donc de $\max(0, 1/4) = 1/4$.

Pour amplifier, on exécute trois fois `max-paritéA` et on retourne la sortie majoritaire, par ex. si on obtient [*vrai*, *faux*, *vrai*], alors on retourne *vrai*. Il y a erreur s'il y a eu au moins deux verdicts erronés. La nouvelle probabilité d'erreur est donc de

$$\underbrace{(1/4)^2}_{\text{deux erreurs}} \cdot \underbrace{(3/4)}_{\text{un succès}} \cdot \overbrace{3}^{\text{nb. de permutations}} + \underbrace{(1/4)^3}_{\text{trois erreurs}} = 10/64.$$

Remarque: on ne peut pas amplifier avec $\vee$ ou $\wedge$, car il peut autant y avoir des erreurs avec vrai que faux.

`max-paritéB`: La probabilité de quitter la boucle est de $1/2$ car il y a une chance sur deux que $s[j] \neq s[i]$. Le nombre d'itérations espéré est donc de $1/(1/2) = 2$ (loi géométrique). Puisque les opérations sont toutes élémentaires, le temps espéré est de $\mathcal{O}(1)$.

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 $\mathcal{O}(n \log n)$

Récurrences 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 $= c \cdot b^n$, 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 $t(n), t(n-1), t(n-2), \ldots$ 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

- *Hanoï*: *src*$[1:n-1] \to$ *tmp*, *src*$[n] \to$ *dst*, *tmp*$[1:n-1] \to$ *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*: débuter 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

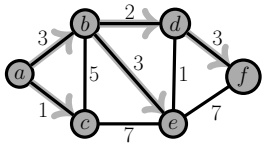
- *Sous-question*: # pièces pour rendre j avec pièces 1 à i?
- *Identité*: $T[i,j] = \min(T[i-1,j],\ T[i,j-s[i]]+1)$
- *Exemple*: montant $m=10$ et pièces $s=[1,5,7]$

	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 $\geq 1, 2, \ldots, |V|-1$ 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 $\Pr(Y_x \neq$ 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*: $\sum$(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