Révision Final

1) Notation asymptotique:

- On classe les algorithmes selon leur complexité en temps dans le pire des cas, en moyenne ou dans le meilleur des cas
  - Dans le cours, on s'est intéressé à la complexité dans le pire des cas

- Analyse théorique
  - Pour calculer la complexité en temps d'un algorithme, on compte le nombre d'opérations élémentaires qu'on doit exécuter dans le pire des cas
  - On utilise la notation asymptotique pour exprimer la complexité d'un algorithme

2) Grand $\mathcal{O}$:

- Soit une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$. On définit l'ordre de $f(n)$ par $\mathcal{O}(f(n))$ comme l'ensemble des fontions bornées supérieurement par $f(n)$:

$$\mathcal{O}(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q. \\ \forall n > n_0, t(n) \leq cf(n)\}$$

- Règle du seuil: Soient deux fonctions strictement positives $f, t : \mathbb{N} \longrightarrow \mathbb{R}^{+}$ alors

$$t(n) \in \mathcal{O}(f(n)) \Longleftrightarrow \exists c' \in \mathbb{R}^{+}\ t.q.$$

$$\forall n \in \mathbb{N}, t(n) \leq c' f(n)$$

2) Grand $\mathcal{O}$ (suite):

- La relation $\mathcal{O}$ est réflexive et transitive i.e.
  - $t(n) \in \mathcal{O}(t(n))$
  - Si $f(n) \in \mathcal{O}(g(n))$ et $g(n) \in \mathcal{O}(h(n))$ alors $f(n) \in \mathcal{O}(h(n))$

- Règle du maximum: Soient deux fonctions $f, g : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ alors

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$$

- $\mathcal{O}(f(n)) = \mathcal{O}(g(n)) \Longleftrightarrow f(n) \in \mathcal{O}(g(n))$ et $g(n) \in \mathcal{O}(f(n))$

3) Grand Ω:

- Soit une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$. On définit $\Omega(f(n))$ comme l'ensemble des fontions bornées inférieurement par : $f(n)$

$$\Omega(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q.\ \forall n \geq n_0,\ t(n) \geq cf(n)\}$$

- Règle de dualité:

$$t(n) \in \Omega(f(n)) \Longleftrightarrow f(n) \in \mathcal{O}(t(n))$$

4) Grand Θ :

- On dit que $t(n)$ est dans l'ordre exact de $f(n)$, dénoté $t(n) \in \Theta(f(n))$, si

$$t(n) \in \mathcal{O}(f(n)) \cap \Omega(f(n))$$

De façon équivalente:

$$\Theta(f(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c, d \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q. \\ \forall n \geq n_0,\ cf(n) \leq t(n) \leq df(n)\}$$

5) Règle de la limite:

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$ alors $f(n) \in \Theta(g(n))$
  ou $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ alors $f(n) \in \mathcal{O}(g(n))$ et
  $g(n) \notin \mathcal{O}(f(n))$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ alors $g(n) \in \mathcal{O}(f(n))$ et
  $f(n) \notin \mathcal{O}(g(n))$

6) Notation asymptotique conditionnelle:

- Soient $f, g : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ et $P : \mathbb{N} \longrightarrow \{\text{vrai}, \text{faux}\}$

$$\mathcal{O}(f(n) \mid P(n)) = \{t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N}\ t.q.$$

$$\forall n \geq n_0, \text{ si } P(n) \rightarrow t(n) \leq cf(n)$$

- De la même façon, on peut définir $\Omega(f(n)|P(n))$ et $\Theta(f(n)|P(n))$

7) Règle de lissage

Définitions:

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est éventuellement non décroissante ou e.n.d. s'il $\exists n_0 \in \mathbb{N}$ tel que

$$\forall n \geq n_0, \quad f(n+1) \geq f(n)$$

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est b-lisse pour $b \in \mathbb{N},\ b \geq 2$ si

  1) elle est e.n.d.

  2) $f(bn) \in \mathcal{O}(f(n))$

- Une fonction $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ est lisse si elle est si elle est b-lisse pour tout $b \in \mathbb{N},\ b \geq 2$

7) Règle de lissage (suite)

- **Règle de lissage:**

  Soit $f : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ une fonction lisse, soit $b \in \mathbb{N},\; b \geq 2$ et soit $t : \mathbb{N} \longrightarrow \mathbb{R}^{\geq 0}$ une fonction e.n.d. alors

$$t(n) \in \Theta(f(n) \mid n \text{ puissance de } b)$$

$$\Longleftrightarrow$$

$$t(n) \in \Theta(f(n))$$

8) Résolution de récurrences linéaires homogènes à coefficients constants:

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

Voici les étapes de la résolution:

1) Trouver le polynôme caractéristique $P(x)$ de la récurrence R

2) Trouver les racines de $P(x)$

Si ces racines sont distinctes

3) La solution générale est de la forme $t_n = \sum_{i=1}^{k} c_i r_i^n$

4) Résoudre le système d'équations linéaires donné par les conditions initiales pour trouver la valeur des constantes $c_1, c_2, \ldots, c_k$

5) Écrire la solution t_n en fonction de ces constantes c_i

8) Résolution de récurrences linéaires homogènes à coefficients constants:

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

Voici les étapes de la résolution:

1) Trouver le polynôme caractéristique $P(x)$ de la récurrence R

2) Trouver les racines de $P(x)$

Si ces racines ne sont pas toutes distinctes

3) La solution générale est de la forme $t_n = \sum_{i=1}^{\ell} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$

où on a ℓ racines r_i de multiplicité m_i

4) Résoudre le système d'équations linéaires donné par les conditions initiales pour trouver la valeur des constantes $c_1, c_2, \ldots, c_k$

5) Écrire la solution t_n en fonction de ces constantes c_i

9) Résolution de récurrences linéaires non-homogènes à coefficients constants (cas particulier):

Soit la récurrence R

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n$$

où b est une constante.

Voici les étapes de la résolution:

1) On commence par transformer cette récurrence en une récurrence homogène

Pour ce cas particulier, on peut multiplier la récurrence R par b et ensuite remplacer n par $n-1$ dans l'équation obtenue

Si on soustrait de la récurrence initiale, cette nouvelle récurrence, nous obtenons une récurrence homogène R^*

2) Résoudre R^* (cas homogène) comme d'habitude mais en s'assurant que les conditions initiales satisfont aussi l'équation de départ

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n$$

10) Changement de variables:

- Il arrive qu'il soit plus facile de commencer par faire un changement de variables lorsque l'on veut résoudre une récurrence

- Par exemple, soit $T(n) = 4t(n/2) + n^2$. En faisant le changement de variables $t_i = T(2^i)$ on obtient la récurrence linéaire non-homogène $t_i - 4t_{i-1} = 4^i$

- On résout cette récurrence de la façon habituelle et ensuite on exprime la solution obtenue pour les t_i en fonction des $T(n)$ en utilisant le fait que $n = 2^i$ et donc que $i = \log_2 n$

11) Algorithmes voraces:

- Facile à développer
- On choisit un optimum local sans se soucier des effets dans le futur (pas de retour en arrière)
- On aimerait que cette stratégie locale nous amène à un optimum global
- On doit faire une preuve d'optimalité pour démontrer qu'un algorithme vorace trouve la solution optimale
- Exemples vus en classe:
  - Retour Monnaie
  - Arbre couvrant minimal (Kruskal - Prim)
  - Plus courts chemins (Dijkstra)
  - Sac à dos

11) Algorithmes voraces - Retour Monnaie:

Problème: On a un nombre illimité de pièces de différentes valeurs. On veut faire la monnaie d'un montant n de sorte qu'on retourne le moins de pièces possibles

Solution vorace: On commence par donner le maximum de pièces de la plus grande valeur (optimum local) et ensuite on complète le montant n avec les pièces de valeurs plus petites

Optimalité: L'optimalité dépend ici de la valeur des pièces en notre possession et du fait qu'on suppose qu'on a un nombre illimité de chacune des pièces.

11) Algorithmes voraces - Arbre Couvrant Minimal:

Problème: Étant donné $G = (N, A)$ un graphe non-orienté connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût, on veut trouver $A' \subseteq A$ tel que $T(N, A')$ est un arbre et tel que la somme

$$c(T) = \sum_{a \in A'} c(A)$$

est minimale.

Solutions voraces:

1) Commencer par un ensemble vide d'arêtes et sélectionner à chaque étape l'arête de plus petit coût qui n'a pas encore été choisie ou rejetée (peut importe où elle se situe dans le graphe) — Kruskal

2) Commencer dans un sommet du graphe et construire un arbre à partir de ce sommet en sélectionnant à chaque étape l'arête de coût minimal qui ajoute à l'arbre existant un nouveau noeud — Prim

11) Algorithmes voraces - Arbre Couvrant Minimal -Kruskal:

- L'algorithme maintient une forêt d'arbres
- À chaque itération, on choisit l'arête de coût minimal
- Cette arête est acceptée, si elle relit deux arbres distincts, sinon elle est rejetée (pourrait forme un cycle)
- L'algorithme se termine lorsqu'on a un seul arbre

11) Algorithmes voraces - Arbre Couvrant Minimal -Prim:

- On choisit un sommet s aléatoirement qu'on met dans un "nuage" et on construit l'arbre couvrant minimal en faisant grossir le "nuage" d'un sommet à la fois.

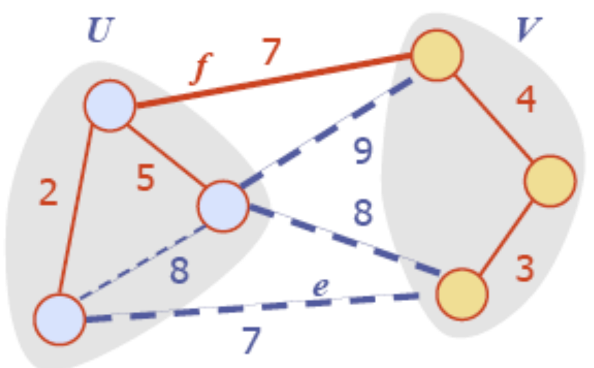
- On garde en mémoire à chaque sommet v, une étiquette d(v) qui ici est égale au poids minimal parmi les poids des arêtes reliant v à un sommet à l'intérieur du nuage.

- À chaque étape:
  - On ajoute au nuage le sommet u extérieur ayant la plus petite étiquette d(u)
  - On met à jour les étiquettes des sommets adjacents à u

11) Algorithmes voraces - Arbre Couvrant Minimal - Optimalité:

Propriété de partition:

- Considérons une partition des sommets de G en deux ensembles U et V
- Soit e une arête de poids minimal entre U et V
- Alors, il existe un arbre couvrant minimal de G contenant e

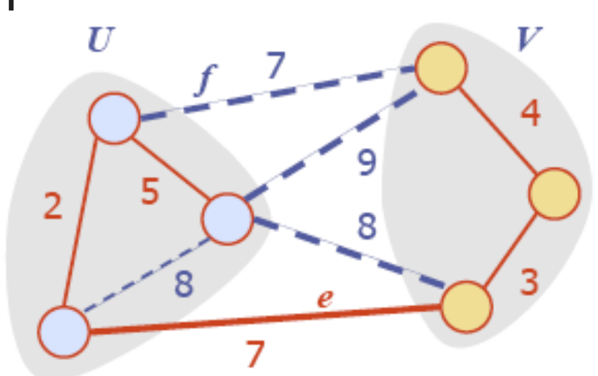


Preuve:

- Soit T un arbre couvrant minimal de G
- Si T ne contient pas e, soit C le cycle formé par l'addition de e à l'arbre T et soit f, une arête entre U et V
- Par la propriété de cycles, on a que

$$poids(f) \leq poids(e)$$

- Comme on avait pris e de poids minimal, on a que poids(f) = poids(e) et alors on obtient un autre ACM en remplaçant f par e



11) Algorithmes voraces - Arbre Couvrant Minimal - Optimalité:

L'optimalité des deux algorithmes découle de la propriété de partition des ACMs

- **Kruskal:**

  La partition du graphe considérée, étant donné une arête (u,v) de coût minimum d'un côté tous les sommets faisant partie de la composante connexe de u et de l'autre tous les autres sommets. Si u et v ne font pas partie de la même composante connexe, la propriété de partition garantie que l'arête (u,v) fait partie d'un ACM

- **Prim:**

  Ici, la partition considérée est nuage/ non-nuage

11) Algorithmes voraces - Plus courts chemins

Problème: Soit $G = (N, A)$ un graphe non-orienté (ou orienté) connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût. Étant donné un sommet source, on veut trouver les plus courts chemins de cette source à tous les autres sommets du graphe.

Solution vorace:

Algorithme de Dijkstra

Algorithme de Dijkstra

- La distance entre un sommet v et un sommet s est le poids total minimal d'un chemin entre v et s
- L'algorithme de Dijkstra est un algorithme glouton qui calcule les distances entre un sommet v et tous les autres sommets d'un graphe
- Ici, on va assumer:
  - Le graphe est connexe
  - Les poids des arêtes sont non-négatifs
- On va faire grossir un "nuage" de sommets, contenant au départ v et couvrant éventuellement tous les sommets
- On va donner une étiquette d(u) à chaque sommet, représentant la distance entre v et u dans le sous-graphe constitué des sommets dans le nuage et des arêtes adjacentes
- À chaque étape:
  - On ajoute au nuage le sommet u extérieur au nuage qui a la plus petite étiquette d(u)
  - On met à jour les étiquettes des sommets adjacents à u

11) Algorithmes voraces - Plus courts chemins

Problème: Soit $G = (N, A)$ un graphe non-orienté (ou orienté) connexe et $c : A \longrightarrow \mathbb{R}^+$ une fonction de coût. Étant donné un sommet source, on veut trouver les plus courts chemins de cette source à tous les autres sommets du graphe.

Solution vorace:

Algorithme de Dijkstra

Preuve d'optimalité: vient de la proposition suivante:

Proposition: Avec Dijkstra, chaque fois qu'un sommet u est inclus dans le nuage, D(u) est le coût d'un chemin minimal entre u et le sommet source et ce chemin est inclus entièrement dans le nuage

11) Algorithmes voraces - Sac à dos

Problème: On dispose de n objets de poids positifs $w_1, w_2, \ldots, w_n$ et de valeurs positives $v_1, v_2, \ldots, v_n$. Notre sac à doc à une capacité maximale en poids de W

Notre but est de remplir le sac de sorte de maximiser la valeur des objets inclus dans le sac tout en respectant la contrainte de poids.

Pour pouvoir résoudre se problème avec un algorithme vorace, on suppose qu'on peut apporter une fraction x_i de chaque objet $i, \quad 0 \leq x_i \leq 1$.

But: Maximiser $\sum_{i=1}^{n} x_i v_i$ tel que $\sum_{i=1}^{n} x_i w_i = W$ et $0 \leq x_i \leq 1$

11) Algorithmes voraces - Sac à dos

Solution vorace:

Sélectionner chaque objet à son tour dans un certain ordre, mettre la plus grande fraction possible de cet objet dans le sac (sans dépasser la capacité en poids du sac) et arrêter quand le sac est plein.

On sélectionne l'objet dans la valeur par unité de poids est maximale

Peruve d'optimalité: vient du théorème suivant:

Théorème: Si les objets sont choisis par ordre décroissant de $\frac{v_i}{w_i}$ alors cette stratégie retourne une solution optimale

12) Algorithmes de programmation dynamique

- C'est une approche du bas vers le haut

- L'idée est que pour résoudre un problème, on commence par résoudre les plus petits sous-problèmes et on conserve les valeurs de ces sous-problèmes dans une table de programmation dynamique. On utilise ensuite ces valeurs pour calculer la valeur de sous-problèmes de plus en plus grands, jusqu'à obtenir la solution de notre problème global.

- Principe d'optimalité: La solution optimale à un problème est composée de solutions optimales à des sous-problèmes

- Exemples vus en classe:
  - Retour Monnaie
  - Sac à dos
  - Plus courts chemins
  - Alignements de séquences

12) Programmation dynamique - Retour Monnaie

Problème: On a un montant M et des pièces de valeurs $v_1, v_2, \ldots, v_n$.
On veut trouver des entiers x_i tels que

$$\sum_{i=1}^{n} x_i v_i = M \quad \text{en minimisant} \quad \sum_{i=1}^{n} x_i$$

Solution: On va construire une table de programmation dynamique $C[1..n, 0..M]$.

où $C[i,j]$ = nombre minimum de pièces pour produire exactement le montant j en utilisant seulement des pièces de valeurs $v_1, v_2, \ldots, v_i$

La solution optimale sera en $C[n, M]$.

12) Programmation dynamique - Retour Monnaie

initialisation de la table: $C[i,0] = 0, \quad \forall i$

On a besoin de 0 pièce pour retourner le montant 0

Pour calculer $C[i,j]$, on a deux choix:

1) On ne prend pas de pièces de valeur v_i même si on en a maintenant le droit. Si on fait ce choix:

$$C[i,j] = C[i-1,j]$$

2) On prend au moins une pièce de valeur v_i. Dans ce cas, après avoir remis cette pièce au client, le total restant à payer est $j - v_i$. Donc, si on fait ce choix:

$$C[i,j] = 1 + C[i, j - v_i]$$

On veut minimiser $C[i,j]$ et donc on pose:

$$C[i,j] = min(C[i-1,j], 1 + C[i, j - v_i])$$

12) Programmation dynamique - Retour Monnaie

On a donc les équations suivantes:

$$C[i,0] = 0, \quad \forall i$$
$$C[i,j] = min(C[i-1,j], 1 + C[i, j - v_i])$$

Comme le problème n'est pas défini pour $C[0,j]$ et pour $C[i, j - v_i]$ quand $j < v_i$, on pose ces cas égales à ∞ pour ne pas les considérer dans le minimum. Aussi, si $i = 1$ et $j < v_1$, on pose $C[i,j] = \infty$ pour signifier qu'il est impossible de payer le montant j avec des pièces de valeur v_1.

Exemple: Supposons qu'on ait un montant de 8 et des pièces de valeurs $v_1 = 1, v_2 = 4$ et $v_3 = 6$:

	0	1	2	3	4	5	6	7	8
$v_1 = 1$	0	1	2	3	4	5	6	7	8
$v_2 = 4$	0	1	2	3	1	2	3	4	2
$v_3 = 6$	0	1	2	3	1	2	1	2	2

12) Programmation dynamique - Sac à dos

Problème: On dispose de n objets de poids positifs $w_1, w_2, \ldots, w_n$ et de valeurs positives $v_1, v_2, \ldots, v_n$. Notre sac à doc à une capacité maximale en poids de W

But: Maximiser $\sum_{i=1}^{n} x_i v_i$ tel que $\sum_{i=1}^{n} x_i w_i \leq W$ et $x_i \in \{0, 1\}$

On va construire une table de programmation dynamique $V[1..n, 0..W]$

où $V[i, j]$ = valeur maximale des objets que l'on peut transporter si le poids maximal permis est j et que les objets que l'on peut inclure sont ceux numérotés de 1 à i

La solution optimale sera en $V[n, W]$.

12) Programmation dynamique - Sac à dos

initialisation de la table: $V[i,0] = 0, \quad \forall i$

La valeur maximale qu'on peut apporter avec une capacité de 0 est 0

Pour calculer $V[i,j]$, on a deux choix:

1) On n'ajoute pas l'objet i dans le sac. Si on fait ce choix:

$$V[i,j] = V[i-1,j]$$

2) On ajoute l'objet i dans le sac. Si on fait ce choix:

$$V[i,j] = V[i-1,j-w_i] + v_i$$

On veut maximiser $V[i,j]$ et donc on pose:

$$V[i,j] = \max(V[i-1,j], V[i-1,j-w_i] + v_i)$$

12) Programmation dynamique - Sac à dos

On a donc les équations suivantes:

$$V[i,0] = 0, \quad \forall i$$
$$V[i,j] = \max(V[i-1,j], V[i-1,j-w_i] + v_i)$$

Le problème n'est pas défini pour $V[0,j]$ et $V[i,j]$ si $j < 0$. On pose $V[0,j] = 0$ pour signifier que si on a aucun objet alors la valeur est 0 peut importe la capacité et $V[i,j] = -\infty$ si $j < 0$ car on ne veut pas considérer de poids négatifs.

Exemple: Supposons qu'on ait 5 objets de poids 1,2,5,6 et 7 et de valeurs $1, 6, 18, 22, 28$ et que la capacité de notre sac est de 11.

v_i	w_i		0	1	2	3	4	5	6	7	8	9	10	11
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
6	2	2	0	1	6	7	7	7	7	7	7	7	7	7
18	5	3	0	1	6	7	7	18	19	24	25	25	25	25
22	6	4	0	1	6	7	7	18	22	24	28	29	29	40
28	7	5	0	1	6	7	7	18	22	28	29	34	35	40

12) Programmation dynamique - Sac à dos

Maintenant, si on a le même problème mais au lieu d'avoir n objets chacun 1 fois, on a n types d'objets, chaque objet en un nombre illimité:

Dans ce cas,

1) soit on ne prend pas d'objet de type i
2) soit on ajoute au sac un objet de type i pour la première fois
3) soit on ajoute au sac un objet de type i et il y a déjà au moins un objet de ce type dans le sac

Les équations de programmation dynamique deviennent:

$$V[i,0] = 0, \quad \forall i$$

$$V[i,j] = \max(V[i-1,j], V[i-1,j-w_i]+v_i, V[i,j-w_i]+v_i)$$

12) Programmation dynamique - Plus courts chemins

Problème: Soit G un graphe orienté, N l'ensemble de ces sommets et A l'ensemble de ces arêtes. Chaque arête à un poids positifs représentant la distance entre les 2 sommets. On veut calculer la longueur des plus courts chemins entre toutes les paires de sommets de G.

Solution: On va construire *n* matrices D_k pour $1 \leq k \leq n$.

où $D_k[i,j]$ = la longueur du plus court chemin entre *i* et *j* et dont les sommets intermédiaires sont dans l'ensemble $\{1, ..., k\}$

La longueur du plus court chemin entre chaque paire (i,j) de sommets est alors $D_n[i,j]$

12) Programmation dynamique - Plus courts chemins

initialisation:

- On pose $D_0[i,i] = 0, \forall i$ car la distance entre un sommet et lui-même est 0
- S'il y a une arête entre i et j, on pose

$$D_0[i,j] = \text{poids de cette arête}$$

- S'il n'y a pas d'arête entre i et j, on pose $D_0[i,j] = \infty$

Pour calculer $D_k[i,j]$, on a deux choix:

1) On ne passe pas par k

$$D_k[i,j] = D_{k-1}[i,j]$$

2) On passe par k

$$D_k[i,j] = D_{k-1}[i,k] + D_{k-1}[k,j]$$

On veut minimiser $D_k[i,j]$ et donc on pose:

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

12) Programmation dynamique - Alignements

Étant donné 2 séquences, $S = s_1 s_2 \ldots s_m$ et $T = t_1 t_2 \ldots t_n$, on définit

D(i,j): distance d'édition entre le préfixe de taille i de S, $s_1 \ldots s_i$, et le préfixe de taille j de T, $t_1 \ldots t_j$.

D définit une matrice de taille (m+1) X (n+1) qu'on appelle la matrice de programmation dynamique

L'idée est alors d'exprimer D(i,j) en fonction des valeurs de D pour des paires d'indices plus petits que (i,j)

Calculer D(i,j) à partir des 3 cases (i-1,j), (i, j-1) et (i-1,j-1):

1. L'alignement se termine par la suppression de s_i

Alignement optimal de $s_1 \dots s_{i-1}$ avec $t_1 \dots t_j$	s_i $-$
D(i-1,j) +	1

2. L'alignement se termine par l'insertion de t_j

Alignement optimal de $s_1 \dots s_i$ avec $t_1 \dots t_{j-1}$	$-$ t_j
D(i,j-1) +	1

3. L'alignement se termine par l'alignement de s_i avec t_j

Alignement optimal de $s_1 \dots s_{i-1}$ avec $t_1 \dots t_{j-1}$	s_i t_j
D(i-1,j-1) +	1 si $s_i \neq t_j$ 0 sinon

Remplissage de la table

Conditions initiales: $D(i,0) = i, \quad \forall i \quad 0 \leq i \leq m$

$$D(0,j) = j, \quad \forall j \quad 0 \leq j \leq n$$

Relation de récurrence pour i, j > 0:

$$D(i,j) = \min \begin{cases} D(i-1,j) & +1 \\ D(i,j-1) & +1 \\ D(i-1,j-1) + \delta(i,j) \end{cases},$$

Où $\delta(i,j) = 0$ si $x_i = y_j$ et 1 sinon.

Complexité: Pour remplir chaque case de la table, on examine 3 cases.
Il y a $O(nm)$ cases et donc complexité en temps de $O(nm)$

Trouver un alignement optimal

Au cours du remplissage de la table, garder des pointeurs:

- de (i-1,j) à (i,j) si D(i,j) = D(i-1,j) + 1
- de (i,j-1) à (i,j) si D(i,j) = D(i,j-1) + 1
- de (i-1,j-1) à (i,j) si D(i,j) = D(i-1,j-1) + $\delta(i, j)$

Un alignement optimal: Commencer à la case (m,n) et suivre des pointeurs jusqu'à la case (0,0).

Une case peut contenir plusieurs pointeurs: plusieurs alignements optimaux possibles

12) Programmation dynamique - Alignements

Conditions initiales: $D(i,0) = i, \quad \forall i \quad 0 \leq i \leq m$
$D(0,j) = j, \quad \forall j \quad 0 \leq j \leq n$

Relation de récurrence pour i, j > 0:

$$D(i,j) = \min \begin{cases} D(i-1,j) & +1 \\ D(i,j-1) & +1 \\ D(i-1,j-1) + \delta(i,j) \end{cases}$$

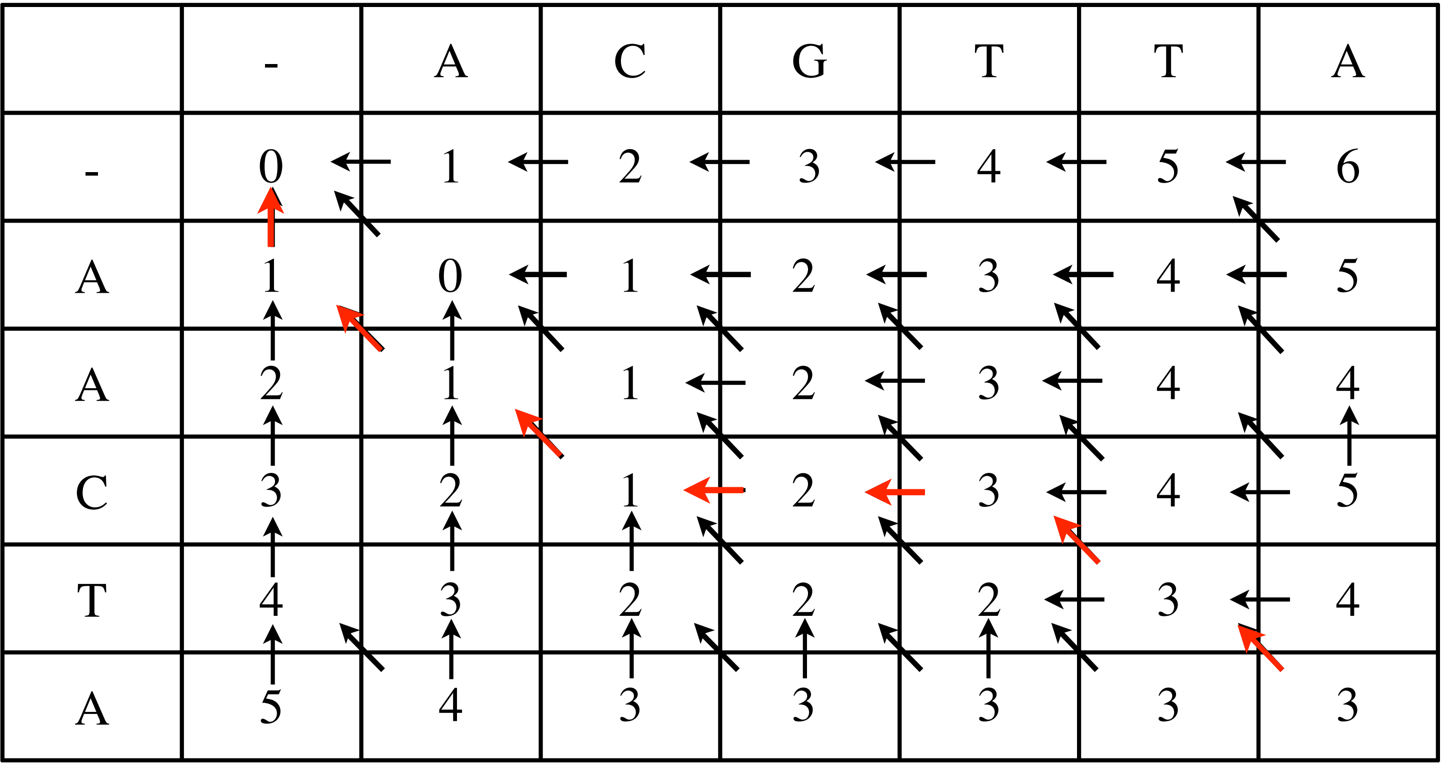
Où $\delta(i,j) = 0$ si $x_i = y_j$ et 1 sinon.

Exemple: Calculez la valeur d'un alignement global entre les mots ACGTTA et AACTA

	-	A	C	G	T	T	A
-	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
A	2	1	1	2	3	4	4
C	3	2	1	2	3	4	5
T	4	3	2	2	2	3	4
A	5	4	3	3	3	3	3

12) Programmation dynamique - Alignements

Exemple: Calculez la valeur d'un alignement global entre les mots ACGTTA et AACTA



	-	A	C	G	T	T	A
-	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
A	2	1	1	2	3	4	4
C	3	2	1	2	3	4	5
T	4	3	2	2	2	3	4
A	5	4	3	3	3	3	3

- A C G T T A

A A C - - T A

Recherche approché d'un motif

Problème: On a un "petit" motif P de taille m et une "longue" séquence T de taille n et on veut trouver toutes les occurrences approximatives de P dans T (moins de k erreurs).

		G	T	C	A	G	G	…
	0	0	0	0	0	0	0	…
C	1	1	1	0	1	1	1	
A	2	2	2	1	0	1	2	
T	3	3	2	2	1	1	2	

- initialiser la première ligne à 0, et la première colonne comme d'habitude
- même relations de récurrence que pour l'alignement global de séquences
- rechercher à la ligne m toutes les cases contenant des valeurs plus petites ou égales à k
- pour trouver un alignement, suivre les pointeurs jusqu'à la première ligne

Alignement local - Algorithme de Smith-Waterman

Relations de récurrence:

$$V(0,j) = 0$$

$$V(i,0) = 0$$

$$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j) & +p(s_i,-) \\ V(i,j-1) & +p(-,t_j) \\ V(i-1,j-1) & +p(s_i,t_j) \end{cases}$$

- Le 0 dans la récurrence permet d'ignorer un nombre quelconque de caractères en début de séquence

- Pour trouver un alignement local de score maximal:

  - On remplie la table
  - On recherche une case c contenant la valeur maximale de la table
  - De cette case c, on suit les pointeurs jusqu'à une case contenant la valeur 0

12) Programmation dynamique - Alignements

$$V(0,j) = 0$$
$$V(i,0) = 0$$

$$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j) + p(s_i,-) & \text{-2} \\ V(i,j-1) + p(-,t_j) & \text{-2} \\ V(i-1,j-1) + p(s_i,t_j) & \text{-1 si } \neq \\ & \text{+2 si } = \end{cases}$$

Exemple: Calculez la valeur d'un alignement local entre les mots ACGTTA et AACTA

	-	A	C	G	T	T	A
-	0	0	0	0	0	0	0
A	0	↖ 2	← 0	0	0	0	↖ 2
A	0	↖ 2	↖ 1	0	0	0	↖ 2
C	0	↑ 0	↖ 4	← 2	← 0	0	↑ 0
T	0	0	↑ 2	↖ 3	↖ 4	← 2 ↖	← 0
A	0	↖ 2	← ↑ 0	↑ 1	↖ ↑ 2	↖ 3	↖ 4

12) Programmation dynamique - Alignements

Exemple: Calculez la valeur d'un alignement local entre les mots ACGTTA et AACTA

	-	A	C	G	T	T	A
-	0	0	0	0	0	0	0
A	0	2	0	0	0	0	2
A	0	2	1	0	0	0	2
C	0	0	4	2	0	0	0
T	0	0	2	3	4	2	0
A	0	2	0	1	2	3	4

A C   A C G T   T A

A C   A C - T   T A

13) Algorithmes diviser-pour-régner:

- C'est une approche du haut vers le bas
- L'idée est qu'on prend un problème et on le casse en sous-problèmes. On résout les sous-problèmes (possiblement en les cassant de nouveau) et on combine les résultats pour obtenir la solution au problème original.
- Construction générale: Si l'instance à résoudre est suffisamment petite, on utilise un algorithme classique pour la résoudre, sinon, on la décompose en morceaux (si possible de même taille) et on rappelle l'algorithme sur ces morceaux pour ensuite les recombiner.
- Exemples vus en classe:
  - Recherche binaire
  - Tri par fusion
  - Tri rapide
  - Sélection
  - Médiane

13) Algorithmes diviser-pour-régner - efficacité:

3 conditions pour avoir un algorithme diviser-pour-régner efficace:

1) Bien décider quand utiliser l'algorithme simple sur de petites instances plutôt que les appels récursifs

2) La décomposition d'une instance en sous-instances et la recombinaison des sous-solutions doivent être efficaces

3) Les sous-instances doivent être, autant que possible, environ de la même taille

13) Algorithmes diviser-pour-régner - complexité:

Souvent la complexité en temps d'un algorithme diviser-pour-régner sur une instance de taille n peut s'écrire comme:

$$T(n) = bT(n/b) + g(n)$$

où $g(n)$ est le temps prit pour casser et reconstruire.

Si on peut montrer que $g(n) \in \Theta(n^k)$ pour un certain k alors le théorème suivant nous donne automatiquement la complexité de l'algorithme:

Soit $T : \mathbb{N} \longrightarrow \mathbb{R}^+$ une fonction éventuellement non décroissante telle que

$$T(n) = \ell T(\frac{n}{b}) + cn^k, \forall n > n_0,$$

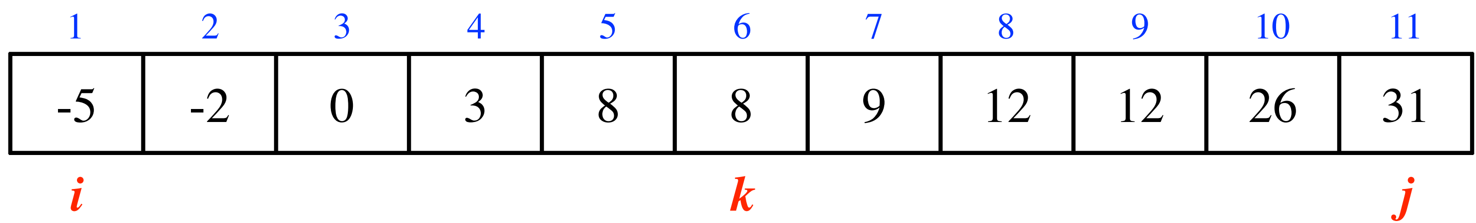
où $\frac{n}{n_0}$ est une puissance de b. Alors,

$$T(n) \quad \in \quad \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log_b n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

13) Algorithmes diviser-pour-régner - recherche binaire:

Problème: Soit L[1 .. n] une liste d'entiers triée en ordre croissant i.e. on a $L[i] \leq L[j], \quad \forall 1 \leq i \leq j \leq n$. Le problème consiste à trouver un entier x dans la liste, ou, si l'entier x n'est pas dans la liste, de trouver la position où il devrait être inséré.

Exemple: Trouver 7 dans la liste triée suivante:



- On initialise le pointeur *i* à 1 et *j* à taille(L) et on regarde si 7 est > L[*j*]. Si oui, la position d'insertion pour 7 est n+1, on fait la recherche binaire.

Recherche binaire: si i=j, retourner i, sinon comparer l'élément chercher à plancher((i+j)/2) et continuer la recherche dans le bon sous-tableau

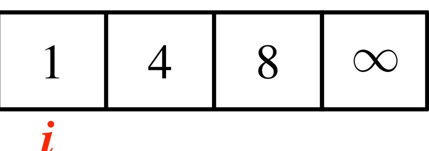
13) Algorithmes diviser-pour-régner - tri-fusion:

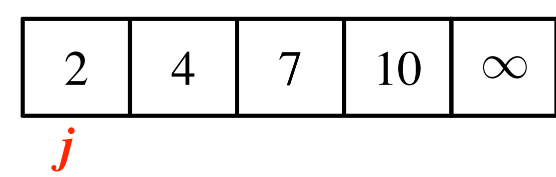
Problème: Trier une liste d'entiers L[1 .. n]

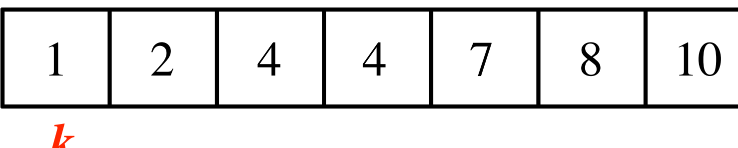
Solution Tri-fusion: Diviser la liste en deux. Trier ces parties par appels récursifs et fusionner les solutions de chaque partie en faisant attention de conserver l'ordre.

Complexité: $T(n) = 2T(n/2) + g(n)$ quand n est pair, où $g(n)$ est le temps prit pour la fusion qui est en $\Theta(n)$. Donc ici, on a $\ell = b = 2$ et $k = 1 \implies T(n) \in \Theta(n \log n)$

Fusion:







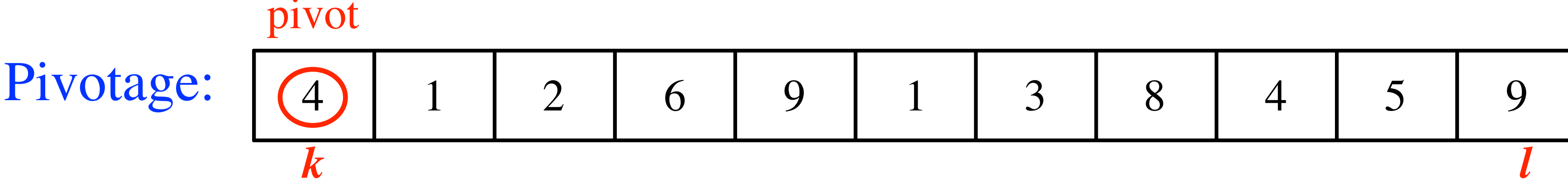
13) Algorithmes diviser-pour-régner - tri-rapide:

Problème: Trier une liste d'entiers L[1 .. n]

Solution Tri-rapide: Prendre le premier élément de la liste comme pivot, pivoter les éléments de L autour de ce pivot de sorte que les éléments plus petits se retrouvent dans une liste L1 à gauche du pivot et les autres éléments dans une liste L2 à droite du pivot. Trier récursivement L1 et L2 avec Tri-rapide

Complexité: En moyenne $\mathcal{O}(n \log n)$. Dans le pire cas $\Omega(n^2)$

Pivotage:



- Deux pointeurs k, initialisé à 1, l initialisé à taille(L).
- Bouger k vers la droite jusqu'à un élément > pivot
- Bouger l vers la gauche jusqu'à un élément $\leq$ pivot
- Échanger L[k] et L[l] et répéter tant que $k < l$
- Échanger pivot et L[l]

13) Algorithmes diviser-pour-régner - tri-rapide:

Complexité: En moyenne $\mathcal{O}(n \log n)$. Dans le pire cas $\Omega(n^2)$

Peut-on éviter le pire cas en $\Omega(n^2)$? Oui

- On calcule la médiane de la liste en temps linéaire en utilisant l'algorithme de sélection implanté avec la procédure pseudo-médiane

- On utilise un pivotage différent qui sépare la liste L en trois parties: les éléments plus petits que le pivot, les éléments égaux au pivot et les éléments plus grands que le pivot

- On rappelle le tri-rapide seulement sur les listes d'éléments plus petits et plus grands que le pivot

Mais non …

L'algorithme modifié de cette façon fonctionne toujours en $\mathcal{O}(n \log n)$ mais la constante cachée liée à cette modification est tellement grande quand pratique le nouveau tri - rapide devient toujours plus lent que tri fusion

13) Algorithmes diviser-pour-régner - sélection:

Soit *L[1..n]* une liste d'entiers, et soit *s* un entier entre *1* et *n.* Le s ième plus petit entier de *L* est défini comme étant l'élément qui serait en position s si la liste *L* était ordonnée de façon non décroissante.

Étant donnés *L* et *s*, le problème de trouver le *s* ième plus petit élément de *L* est appelé le problème de sélection.

On peut se servir d'une algorithme qui calcule la médiane d'une liste pour faire la sélection du s ième plus petit élément de cette liste, sélection(L[1..n],s).

- on utilise un algo, médiane(L), qui retourne p la médiane de la liste L
- on appelle l'algorithme pivotbis(L,p ; k,ℓ) qui va partitionner L en 3 sections:
  - L[1..k] qui contient les éléments < p — $s \leq k$ — sélection(L[1..k],s)
  - L[k+1..ℓ-1] qui contient les éléments = p — $k < s < \ell$ — s=p
  - L[ℓ..n] qui contient les éléments > p — $s \geq \ell$ — sélection(L[ℓ..n],s)

13) Algorithmes diviser-pour-régner - sélection:

Peut-on modifier l'algo de sélection pour ne pas avoir à utiliser la médiane?

Oui - on calcule une pseudo-médiane

Algorithme PseudoMed($L[1..n]$)

Si $n \leq 5$ alors
    retourner vraiMediane(L)
Sinon
    $z \leftarrow \lfloor \frac{n}{5} \rfloor$
    initialiser une liste $M[1..z]$
    Pour i de 1 à z faire
        $M[i] \leftarrow$ vraiMediane($L[5i - 4..5i]$)
    Fin Pour
Fin Si
Retouner Selection(M, $\lceil \frac{z}{2} \rceil$)

13) Algorithmes diviser-pour-régner - médiane:

On peut montrer que l'algorithme sélection(L[1..n],s) qui utilise la pseudo-médiane est en $\Theta(n)$

Donc pour calculer la médiane en $\Theta(n)$ on utilise cette algorithme de sélection avec sélection(L[1..n],$\lceil n/2 \rceil$)

14) Algorithmes de parcours de graphe - parcours en profondeur

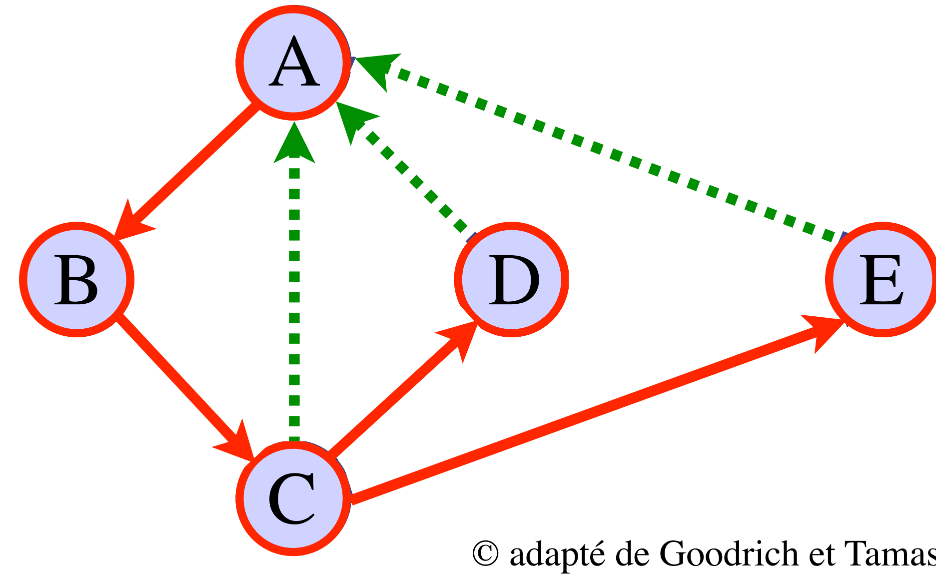
A Sommets non visités

Sommets visités

Arêtes non visitées

Arêtes sélectionnées

Arêtes de retour



© adapté de Goodrich et Tamassia 2004

Algorithme dfSearch(G)

$\forall v \in \mathbb{N}$ faire
étiquette$[v] \leftarrow$ non-visité
$\forall v \in \mathbb{N}$ faire
si étiquette$[v]$ = non-visité
dfs(v)

Algorithme dfs(v)
étiquette$[v] \leftarrow$ visité
pour chaque noeud w adjacent à v faire
si étiquette$[w]$ = non-visité
dfs(w)

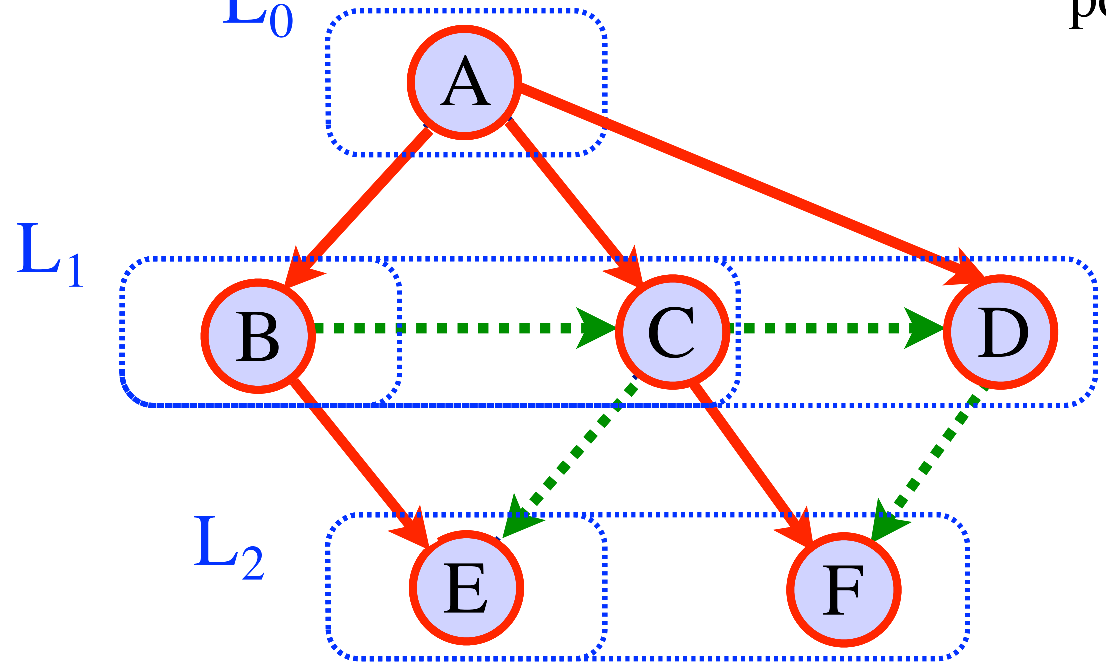
14) Algorithmes de parcours de graphe - parcours en largeur

Algorithme bfSearch(G)

$\forall v \in \mathbb{N}$ faire
étiquette$[v] \leftarrow$ non-visité
$\forall v \in \mathbb{N}$ faire
si étiquette$[v]$ = non-visité
bfs(v)

Algorithme bfs(v)

$Q \leftarrow$ file vide
étiquette$[v] \leftarrow$ visité
enfiler v dans Q
tant que Q est non vide faire
$u \leftarrow$ premier(Q)
défiler u de Q
pour chaque noeud w adjacent à u faire
si étiquette$[w]$ = non-visité alors
étiquette$[w] \leftarrow$ visité
enfiler w dans Q



© adapté de Goodrich et Tamassia 2004

DFS vs BFS

Applications	DFS	BFS
fôret couvrante, composantes connexes	ok	ok
chemins entre deux sommets, cycles	ok	ok
plus court chemin		ok

15) Algorithmes retour-arrière

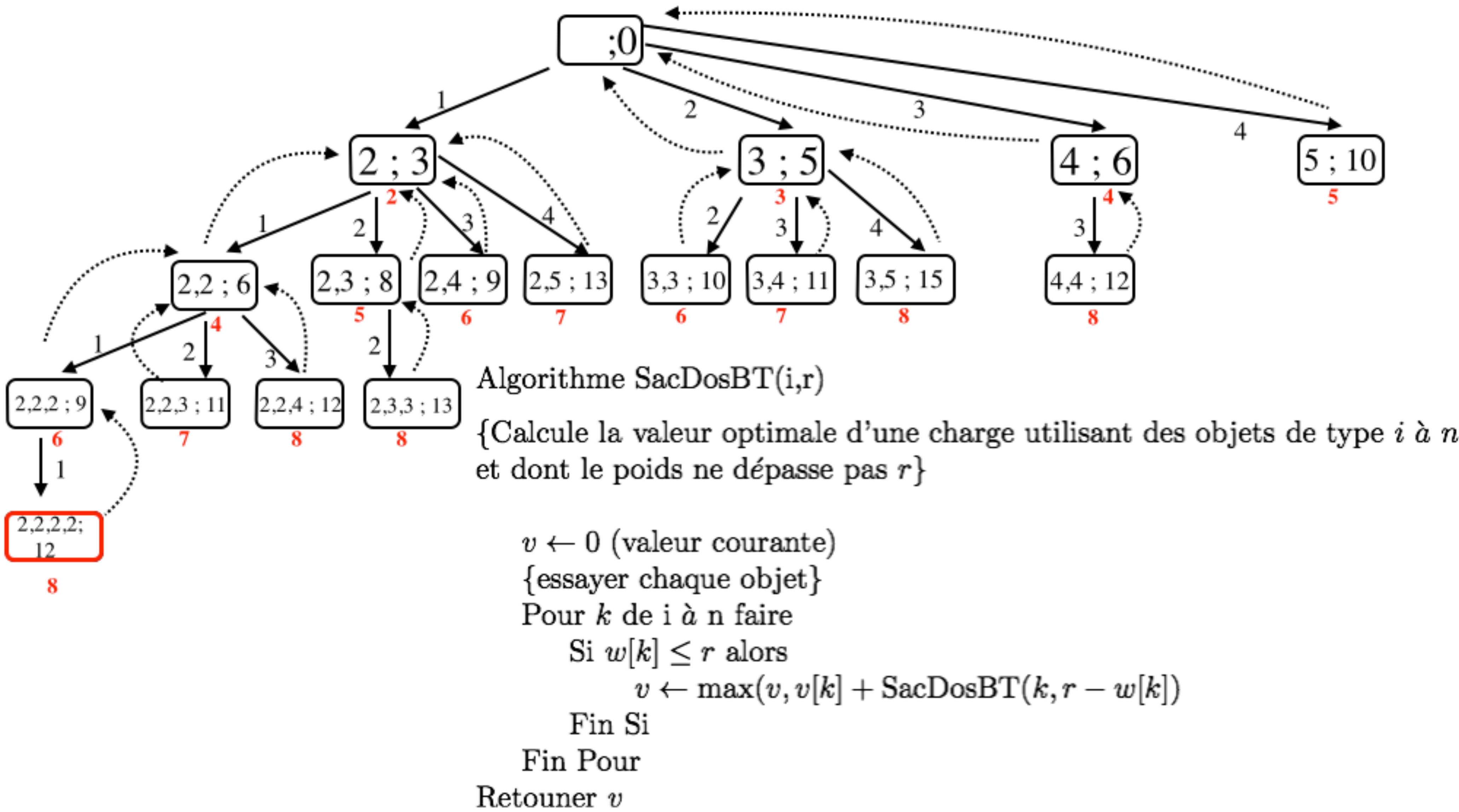
- L'idée est de parcourir un arbre implicite de toutes les sous-solutions d'un problème pour trouver la solution optimale

- Un algorithme retour en arrière commence à la racine de cet arbre implicite (solution nulle) et explore l'arbre en profondeur construisant les noeuds et les solutions partielles au fur et à mesure.

- Exemples vus en classe:
  - Sac à dos
  - 8 reines

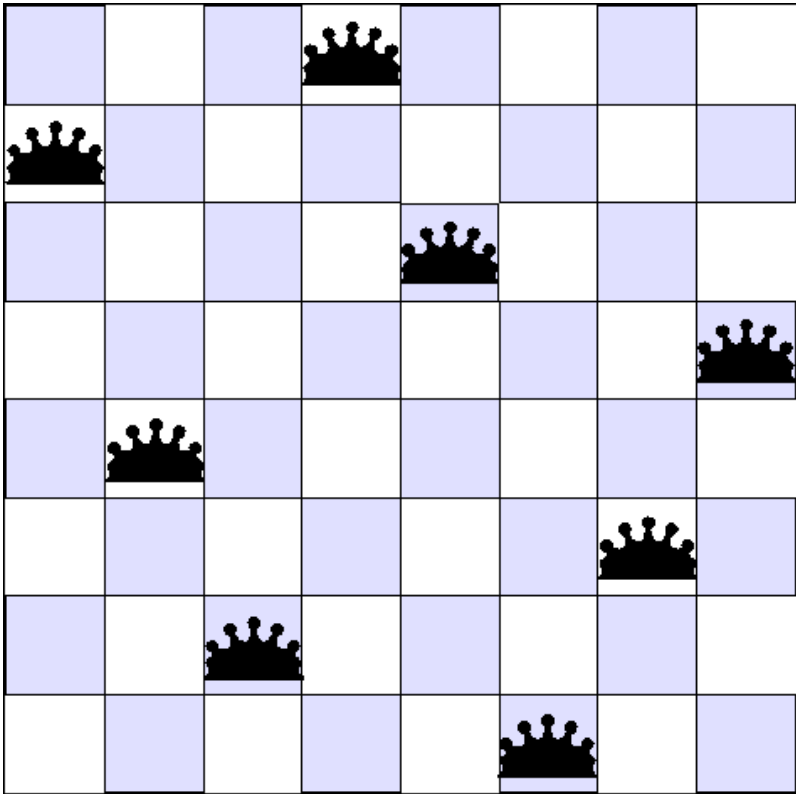
15) Algorithmes retour-arrière - sac à dos

Exemple: Supposons qu'on ait 4 types d'objets de poids respectifs 2, 3, 4 et 5 et de valeurs 3, 5, 6 et 10 et supposons que W=8.



15) Algorithmes retour-arrière - 8 reines

Problème: Placer 8 reines sur un échiquier de sorte qu'aucune reine n'en menace une autre. (Une reine menace toutes les pièces positionnées dans la même rangée, la même colonne ou la même diagonale).



Parallel Programming in C for the Transputer
© D. Thiébaut, 1995

15) Algorithmes retour-arrière - 8 reines

Notre problème des 8 reines peut être représenté par un arbre A=(N,E) où

- N est l'ensemble des vecteurs k-prometteurs, $0 \leq k \leq 8$

> Un vecteur V est k-prometteur, si pour toutes paires d'entiers i, j entre 1 et k on a V[i] - V[j] $\notin$ {i-j, 0, j-i}

- E est l'ensemble des arêtes tels que (u,v) $\in$ E ssi $\exists k,\ 0 \leq k \leq 8$, tel que

  - u est k-prometteur
  - v est k+1-prometteur
  - u[i] = v[i], $\forall i,\ 1 \leq i \leq k$

15) Algorithmes retour-arrière - 8 reines

Algorithme ReineBT(sol, k, col, diag45, diag135)

Si $k = 8$ alors
    Retourner sol
Sinon
    Pour j de 1 $à$ 8 faire
        Si $j \notin$ col et $j - k \notin$ diag 45 et $j + k \notin$ diag 135 alors
            $sol[k+1] \leftarrow j$
            reineBT(sol, $k+1$, col $\cup \{j\}$, diag45$\cup \{j-k\}$, diag135$\cup \{j+k\}$)
        Fin Si
    Fin Pour
Fin si

On appelle l'algorithme avec ReineBT([], 0, {},{},{})

16) Algorithmes branch-and-bound

- L'idée est de parcourir un arbre implicite de toutes les sous-solutions d'un problème pour trouver la solution optimale

- À chaque noeud, on calcule une borne pour les valeurs des solutions découlant de ce noeud

- Si cette borne montre que ces solutions seront nécessairement pire que la meilleure solution trouvée jusque là, nous n'avons pas besoin d'explorer cette partie du graphe

- Exemples vus en classe:
  - Assignation
  - Sac à dos

17) Algorithmes probabilistes

- La caractéristique principale d'un algorithme probabiliste est que le même algorithme peut se comporter différemment lorsqu'on le fait rouler plusieurs fois sur les mêmes données. Le temps d'exécution et même le résultat obtenu peut varier considérablement à chaque exécution

- Deux types d'algorithmes probabilistes ne garantissent pas la justesse de leur résultat:
  - Algorithmes numériques
  - Algorithmes Monte Carlo

- Un algorithme probabiliste qui retourne toujours le bon résultat est appelé algorihtme Las Vegas

17) Algorithmes probabilistes - types

1) Numérique: Retourne une solution approximative à un problème numérique

plus de temps = plus de précision

2) Monte Carlo: Retourne toujours une réponse mais peut se tromper.

plus de temps = plus grande probabilité que la réponse soit bonne

3) Las Vegas: Ne retourne jamais une réponse inexacte mais parfois ne trouve pas de réponse du tout

plus de temps = plus grande probabilité de succès sur chaque instance de départ

17) Algorithmes probabilistes - temps espéré vs temps moyen

TempsMoyen(n) = $\dfrac{\displaystyle\sum_{|w|=n} \text{temps sur instance } w}{\#\{w : |w| = n\}}$

Temps espéré: Défini sur chaque instance w

TempsEspéré(w) = C'est le temps moyen que prend l'algorithme probabiliste pour résoudre w un grand nombre de fois

TempsMoyenEspéré(n) = $\dfrac{\displaystyle\sum_{|w|=n} \text{TempsEspéré}(w)}{\#\{w : |w| = n\}}$

PireTempsEspéré(n) = $\max_{|w|=n} \{\ \text{TempsEspéré}(w)\ \}$

17) Algorithmes probabilistes - algorithmes numériques

Ce sont les premiers algorithmes à utiliser l'aléatoire

Retourne une solution approximative à un problème numérique

La solution retournée est toujours approximative mais sa précision augmente avec le temps disponible pour trouver une solution.

Habituellement, l'erreur est inversément proportionnelle à la racine carré de la quantité de travail effectuée.

Un des première utilisation des algorithmes probabilistes: Estimer π

17) Algorithmes probabilistes - algorithmes Las Vegas

Las Vegas de type 1: Utilise le hasard pour guider ses choix et arrive toujours à résoudre le problème

$$\text{mauvais choix} \Rightarrow \text{plus lent}$$

Exemple: tri

Las Vegas de type 2: Utilise le hasard pour tenter de résoudre un problème quitte à échouer

$$\text{mauvais choix} \Rightarrow \text{échec avoué}$$

Exemple: les 8 reines

17) Algorithmes probabilistes - algorithmes Las Vegas

Particulièrement utile quand un algorithme déterministe existe pour le problème qui est:

- bon en moyenne
- mauvais au pire cas

Un algorithme Las Vegas pour ce problème pourra alors:

- éliminer les instances pire cas
- uniformiser les instances
- maintenir un bon temps espéré

17) Algorithmes probabilistes - algorithmes Monte Carlo

Peut se tromper (pas d'avertissement si c'est le cas)

Trouve une **solution correcte** avec une bonne probabilité et ce **quelle que soit** l'instance de départ

On dit qu'un algorithme Monte Carlo est p-correct s'il retourne une solution correcte avec probabilité p ou plus, $0 < p < 1$

Une propriété intéressante des algorithmes Monte Carlo est qu'il est souvent possible de réduire la probabilité d'erreur en augmentant le temps de calcul (i.e. faire tourner l'algorithme un grand nombre de fois sur chaque instance de départ)

Amplifier l'avantage stochastique