Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Exemple 1 : découpage de tiges d'acier

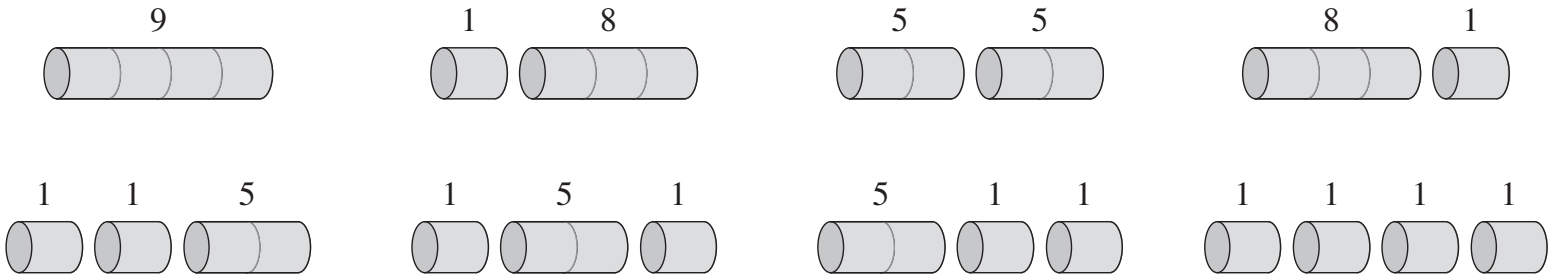
- Soit une tige d'acier qu'on découpe pour la vendre morceau par morceau
- La découpe ne peut se faire que par nombre entier de centimètres
- Le prix de vente d'une tige dépend (non linéairement) de sa longueur
- On veut déterminer le revenu maximum qu'on peut attendre de la vente d'une tige de n centimètres
- Problème algorithmique :
  - Entrée : une longueur $n > 0$ et une table de prix p_i, pour $i = 1, 2, \ldots, n$
  - Sortie : le revenu maximum qu'on peut obtenir pour des tiges de longueur n

Illustration

- Soit la table de prix :

Longueur i	1	2	3	4	5	6	7	8	9	10
Prix p_i	1	5	8	9	10	17	17	20	24	30

- Découpes possibles d'une tige de longueur $n = 4$



- Meilleur revenu : découpage en 2 tiges de 2 centimètres, revenu de 10

Approche par force brute

- Enumérer toutes les découpes, calculer leur revenu, déterminer le revenu maximum
- Complexité : exponentielle en n :
  - Il y a 2^{n-1} manières de découper une tige de longueur n (on peut couper ou non après chacun des $n-1$ premiers centimètres)
  - Plusieurs découpes sont équivalentes (1+1+2 et 1+2+1 par exemple) mais même en prenant cela en compte, le nombre de découpes reste exponentiel
- Infaisable pour n un peu grand

Idée

- Soit r_i le revenu maximum pour une tige de longueur i
- Peut-on formuler r_n de manière récursive ?
- Déterminons r_i pour notre exemple :

i	r_i	solution optimale
1	1	1 (pas de découpe)
2	5	2 (pas de découpe)
3	8	3 (pas de découpe)
4	10	2+2
5	13	2+3
6	17	6 (pas de découpe)
7	18	1+6 ou 2+2+3
8	22	2+6 ...

Formulation récursive de r_n : version naïve

- r_n peut être calculé comme le maximum de :
  - p_n : le prix sans découpe
  - $r_1 + r_{n-1}$: le revenu max pour une tige de 1 et une tige de $n-1$
  - $r_2 + r_{n-2}$: le revenu max pour une tige de 2 et une tige de $n-2$
  - ...
  - $r_{n-1} + r_1$.
- C'est-à-dire

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Formulation récursive de r_n : version simplifiée

- Toute solution optimale a un découpe la plus à gauche
- On peut calculer r_n en considérant toutes les tailles pour la première découpe et en combinant avec le découpage optimal pour la partie à droite
- Pour chaque cas, on n'a donc qu'à résoudre un seul sous-problème (au lieu de deux), celui du découpage de la partie droite
- En supposant $r_0 = 0$, on obtient ainsi :

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Implémentation récursive directe

- La formule récursive peut être implémentée directement

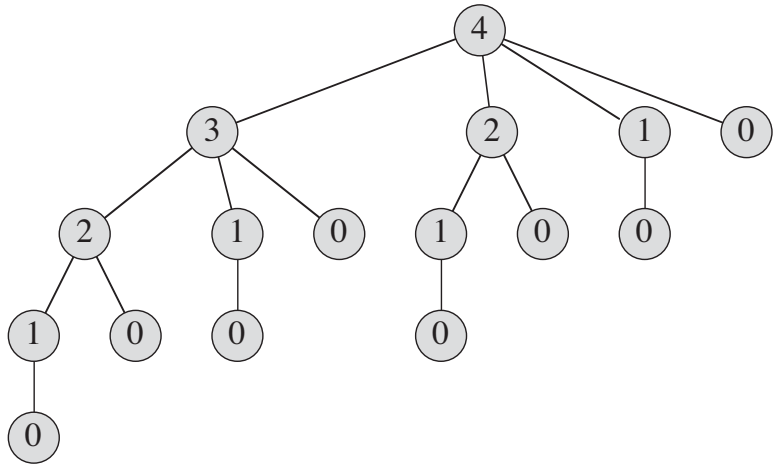
```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = -∞
4  for i = 1 to n
5        q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

(p est un tableau de taille n contenant les prix des tiges de tailles 1 à n)

- Complexité ?

Implémentation récursive directe : analyse

- L'algorithme est extrêmement inefficace à cause des appels récursifs redondants
- Exemple : arbre des appels récursifs pour le calcul de r_4



- En général, le nombre de nœuds $T(n)$ de l'arbre est 2^n.
  Preuve par induction :
  - Cas de base : $T(0) = 1$
  - Cas inductif :

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + 2^n - 1 = 2^n$$

- Complexité de l'algorithme est exponentielle en n

Solution par programmation dynamique

- Solution : plutôt que de résoudre les mêmes sous-problèmes plusieurs fois, s'arranger pour ne les résoudre chacun qu'une seule fois
- Comment ? En sauvegardant les solutions dans une table et en se référant à la table à chaque demande de résolution d'un sous-problème déjà rencontré
- On échange du temps de calcul contre de la mémoire
- Permet de transformer une solution en temps exponentiel en une solution en temps polynomial
- Deux implémentations possibles :
  - descendante (top-down) avec mémoization
  - ascendante (bottom-up)

Approche descendante avec mémoization

```
MEMOIZED-CUT-ROD(p, n)
1  Let r[0..n] be a new array
2  for i = 1 to n
3       r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

(Attention : suppose que le tableau est passé par pointeur)

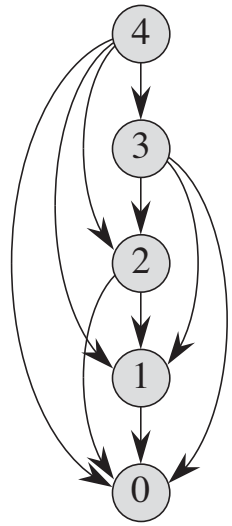
Approche ascendante

Principe : résoudre les sous-problèmes par taille en commençant d'abord par les plus petits

```
BOTTOM-UP-CUT-ROD(p, n)
1  Let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4        q = -∞
5        for i = 1 to j
6              q = max(q, p[i] + r[j - i])
7        r[j] = q
8  return r[n]
```

Programmation dynamique : analyse

- Solution ascendante est clairement $\Theta(n^2)$ (deux boucles imbriquées)
- Solution descendante est également $\Theta(n^2)$
  - Chaque sous-problème est résolu une et une seule fois
  - La résolution d'un sous-problème passe par une boucle à n itérations
- Graphes des sous-problèmes :



(une flèche de x à y indique que la résolution de x dépend de la résolution de y)

Reconstruction de la solution

- Fonction BOTTOM-UP-CUT-ROD calcule le revenu maximum mais ne donne pas directement la découpe correspondant à ce revenu
- On peut étendre l'approche ascendante pour enregistrer également la solution dans une autre table

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  Let r[0..n] and s[1..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4        q = -∞
 5        for i = 1 to j
 6              if q < p[i] + r[j - i]
 7                    q = p[i] + r[j - i]
 8                    s[j] = i
 9        r[j] = q
10  return r and s
```

- $s[j]$ contient la coupure la plus à gauche d'une solution optimale au problème de taille j

Reconstruction de la solution

- Pour afficher la solution, on doit "remonter" dans s

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3        PRINT s[n]
4        n = n − s[n]
```

- Exemple :

i	0	1	2	3	4	5	6	7	8
$p[i]$	0	1	5	8	9	10	17	17	20
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

$$\text{PRINT-CUT-ROD-SOLUTION}(p, 8) \Rightarrow \text{"2 6"}$$

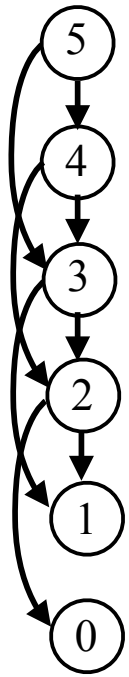
Programmation dynamique : généralités

- La programmation dynamique s'applique aux problèmes d'optimisation qui peuvent se décomposer en sous-problèmes de même nature, et qui possèdent les deux propriétés suivantes :
  - Sous-structure optimale : on peut calculer la solution d'un problème de taille n à partir de la solution de sous-problèmes de taille inférieure
  - Chevauchement des sous-problèmes : Certains sous-problèmes distincts partagent une partie de leurs sous-problèmes
- Implémentation directe récursive donne une solution de complexité exponentielle
- Sauvegarde des solutions aux sous-problèmes donne une complexité linéaire dans le nombre d'arcs et de sommets du graphe des sous-problèmes

Exemple 2 : Fibonacci

- La fonction FIBONACCI-ITER vue au début du cours est un exemple de programmation dynamique (ascendante)

```
FIBONACCI-ITER(n)
 1  if n ≤ 1
 2        return n
 3  else
 4        pprev = 0
 5        prev = 1
 6        for i = 2 to n
 7              f = prev + pprev
 8              pprev = prev
 9              prev = f
10        return f
```



- On peut se contenter de ne stocker que les deux dernières valeurs
- Complexité $\Theta(n)$ (graphe contient $n+1$ nœuds et $2n-2$ arcs)

(Exercice : écrivez la version descendante avec mémoization)

Interlude : Fibonacci en $\Theta(\log n)$

- Peut-on faire mieux que $\Theta(n)$ pour Fibonacci ? Oui !
- Propriété :

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

- Preuve par induction :
  - Cas de base ($n = 1$) : ok puisque $F_0 = 0$, $F_1 = 1$, et $F_2 = 1$
  - Cas inductif ($n \geq 2$) :

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \end{aligned}$$

□

Interlude : Fibonacci en $\Theta(\log n)$

- Approche par force brute pour le calcul de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$: $\Theta(n)$
- Idée : utiliser le diviser-pour-régner pour le calcul de a^n

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{si } n \text{ est pair} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{si } n \text{ est impair} \end{cases}$$

- Complexité : $\Theta(\log n)$ (comme la recherche binaire)

(Exercice : implémenter l'algorithme)

Exemple 3 : sous-séquence maximale

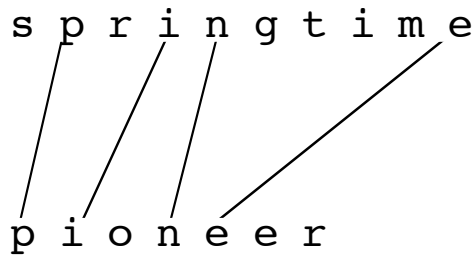
```
MAX-SUBARRAY-LINEAR(A)
 1  Let m[1..n] be a new array
 2  max-so-far = A[1]
 3  m[1] = A[1]
 4  for i = 2 to A.length
 5        if m[i − 1] > 0
 6              m[i] = m[i − 1] + A[i]
 7        else m[i] = A[i]
 8        if m[i] > max-so-far
 9              max-so-far = m[i]
10  return max-so-far
```

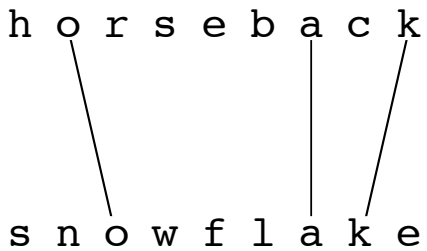
- Complexité : $\Theta(n)$ (diviser pour régner : $\Theta(n \log n)$)
- $m[i]$ est la somme de la sous-séquence maximale qui se termine en i
- L'algorithme calcule $m[i]$ à partir de $m[i-1]$
- Forme de programmation dynamique ascendante (très simple)

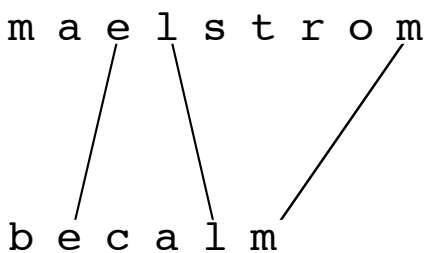
(Exercice : ajouter le calcul des bornes d'un sous-tableau solution, remplacer le tableau m par une seule variable)

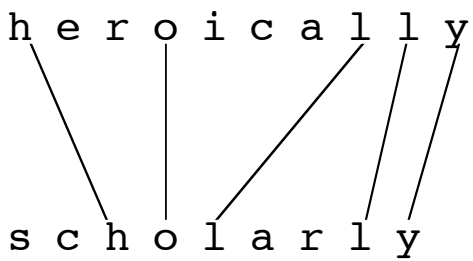
Exemple 4 : plus longue sous-séquence commune

- Définition : Une sous-séquence (non contiguë) d'une séquence $\langle x_1, \ldots, x_m \rangle$ est une séquence $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$, où $1 \leq i_1 < i_2 < \ldots < i_k \leq m$.
- Problème : Etant donné 2 séquences, $X = \langle x_1, \ldots, x_m \rangle$ et $Y = \langle y_1, \ldots, y_n \rangle$, trouver une plus grande sous-séquence commune aux deux séquences
- Exemples :









Solution par force brute

- On énumère toutes les sous-séquences de la séquence la plus courte
- Pour chacune d'elles, on vérifie si c'est une sous-séquence de la séquence la plus longue

- Complexité : $\Theta(n \cdot 2^m)$ (en supposant que $n < m$)
  - 2^m sous-séquences possibles dans une séquence de longueur m
  - Vérification de l'occurence d'une sous-séquence dans une séquence de longueur n en $\Theta(n)$

(Exercice : implémenter la vérification)

Solution par programmation dynamique

Propriété de sous-structure :

- Soit $X_i = \langle x_1, \ldots, x_i \rangle$ un préfixe de X et $Y_i = \langle y_1, \ldots, y_i \rangle$ un préfixe de Y
- Soit $Z = \langle z_1, \ldots, z_k \rangle$ une plus longue sous-séquence commune de X et Y
- Les propriétés suivantes sont vérifiées :
  - Si $x_m = y_n$, alors $z_k = x_m = y_n$ et Z_{k-1} est une plus longue sous-séquence commune de X_{m-1} et Y_{n-1}.
  - Si $x_m \neq y_n$, alors $z_k \neq x_m \Rightarrow Z$ est une plus longue sous-séquence commune à X_{m-1} et Y
  - Si $x_m \neq y_n$, alors $z_k \neq y_n \Rightarrow Z$ est une plus longue sous-séquence commune à X et Y_{n-1}

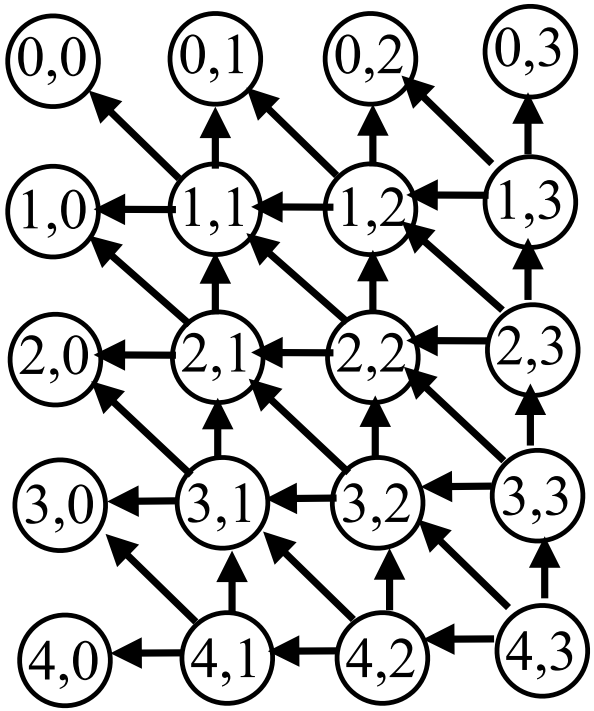
$\Rightarrow$ Une plus longue sous-séquence commune de deux séquences a pour préfixe une plus longue sous-séquence des préfixes des deux séquences.

Solution par programmation dynamique

- Soit $c[i,j]$ la longueur d'une plus longue sous-séquence de X_i et Y_j.
- Formulation récursive :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c[i-1,j-1]+1 & \text{si } i,j > 0 \text{ et } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{si } i,j > 0 \text{ et } x_i \neq y_j, \end{cases}$$

- Graphe des sous-problèmes :



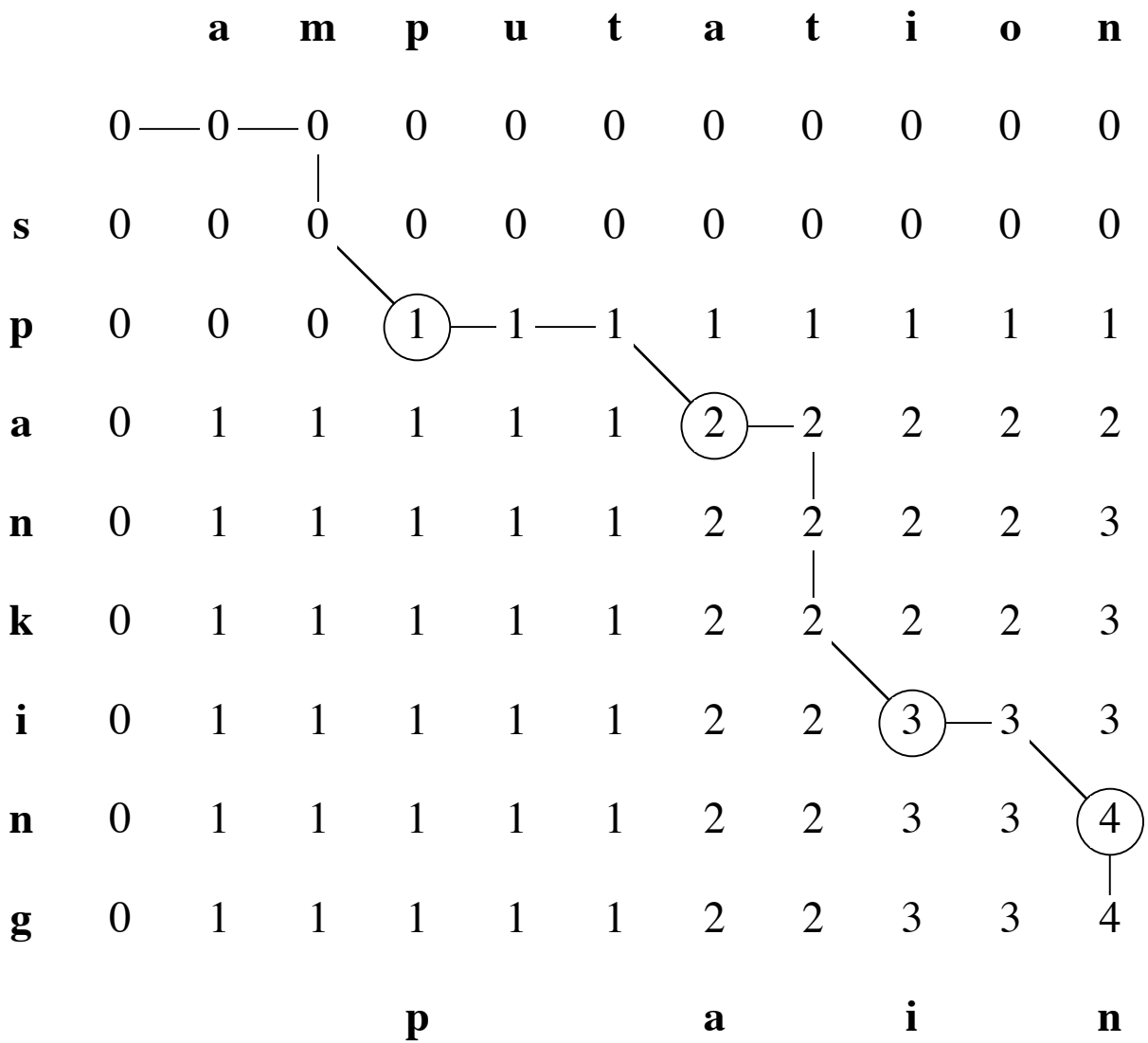
Implémentation (ascendante)

```
LCS-LENGTH(X, Y, m, n)
 1  Let c[0..m, 0..n] be a new table
 2  for i = 1 to m
 3        c[i,0]=0
 4  for j = 0 to n
 5        c[0,j]=0
 6  for i = 1 to m
 7        for j = 1 to n
 8              if x_i == y_j
 9                    c[i,j] = c[i-1,j-1] + 1
10              elseif c[i-1,j] ≥ c[i,j-1]
11                    c[i,j] = c[i-1,j]
12              else c[i,j] = c[i,j-1]
13  return c
```

Complexité : $\Theta(m \cdot n)$

Illustration

amputation versus *spanking*



Trouver la plus longue sous-séquence

```
LCS-Length(X, Y, m, n)
 1  Let c[0..m, 0..n] be a new table
 2  Let b[1..m, 1..n] be a new table
 3  for i = 1 to m
 4       c[i,0]=0
 5  for j = 0 to n
 6       c[0,j]=0
 7  for i = 1 to m
 8       for j = 1 to n
 9            if x_i == y_j
10                 c[i,j] = c[i-1,j-1] + 1
11                 b[i,j] = " ↖ "
12            elseif c[i-1,j] ≥ c[i,j-1]
13                 c[i,j] = c[i-1,j]
14                 b[i,j] = " ↑ "
15            else c[i,j] = c[i,j-1]
16                 b[i,j] = " ← "
17  return c and b
```

```
Print-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i,j] == " ↖ "
4       Print-LCS(b, X, i-1, j-1)
5       print x_i
6  elseif b[i,j] == " ↑ "
7       Print-LCS(b, X, i-1, j)
8  else Print-LCS(b, X, i, j-1)
```

Exemple 5 : le problème du sac à dos (knapsack)

Problème :

- Un voleur se rend dans un musée pour commettre un méfait avec un sac à dos pouvant contenir W kg.
- Le musée comprend n œuvres d'art, chacune de poids p_i et de prix v_i $(i = 1, \ldots, n)$
- Le problème pour le voleur est de déterminer une sélection d'objets de valeur totale maximale et n'excédant pas le poids total admissible dans le sac à dos.

Formellement :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeurs $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in \{0, 1\}$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Exemple

Capacité du sac à dos :

$$W = 11$$

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Exemple :

- $\{5, 2, 1\}$ a un poids de 10 et une valeur de 35
- $\{3, 4\}$ a un poids de 11 et une valeur de 40

Approche par force brute

- Recherche exhaustive : on énumère tous les sous-ensembles de S, et on calcule leur poids et leur valeur
- Complexité en temps : $O(n2^n)$
- Améliorations :
  - Ne tester que les sous-ensembles de W/p_{min} objets où p_{min} est la taille minimale
  - Tester les objets par ordre croissant et s'arrêter dès que l'un d'entre eux n'entre plus
- Diminue la constante mais la complexité reste la même

Approche par programmation dynamique

- Définition : soit $M(k, w)$, $0 \leq k \leq n$ et $0 \leq w \leq W$, le bénéfice maximum qu'on peut obtenir avec les objets $1, \ldots, k$ de S et un sac à dos de charge maximale w
  (On suppose que les poids p_i et W sont entiers)
- Deux cas :
  - On ne sélectionne pas l'objet k : $M(k, w)$ est le bénéfice maximum en sélectionnant parmi les $k-1$ premiers objets avec comme limite w ($M(k-1, w)$)
  - On sélectionne l'objet k : $M(k, w)$ est la valeur de l'objet k plus le bénéfice maximum en sélectionnant parmi les $k-1$ premiers objets avec la limite $w - p_k$

$$M(k,w) = \begin{cases} 0 & \text{si } k = 0 \\ M(k-1,w) & \text{si } p_k > w \\ \max\{M(k-1,w), v_k + M(k-1, w-p_k)\} & \text{sinon} \end{cases}$$

Implementation

```
KnapSack(p, v, n, W)
 1  Let M[0..n, 0..W] be a new table
 2  for w = 0 to W
 3        M[0,w]=0
 4  for k = 1 to n
 5        M[k, 0] = 0
 6  for k = 1 to n
 7        for w = 1 to W
 8              if p[k] > w
 9                    M[k, w] = M[k − 1, w]
10              elseif M[k − 1, w] > v[k] + M[k − 1, w − p[k]]
11                    M[k, w] = M[k − 1, w]
12              else M[k, w] = v[k] + M[i − 1, w − p[k]]
13  return M[n, W]
```

Exemple

M	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1,2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1,2,3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1,2,3,4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1,2,3,4,5\}$	0	1	6	7	7	18	22	28	29	34	35	40

Solution optimale : $\{4,3\}$
Bénéfice : $22 + 18 = 40$

$W = 11$

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Récupération des x_i

En remontant dans le tableau M :

```
KnapSack(p, v, n, W)
 1  // Compute M
 2  ...
 3  // Retrieve solution
 4  Let x[1..n] be a new table
 5  w = W
 6  for k = n downto 1
 7        if M[k, w] == M[k − 1, w]
 8              x[k] = 0
 9        else
10              x[k] = 1
11              w = w − p[k]
12  return x
```

Complexité

- Complexité en temps et en espace : $\Theta(nW)$
  - Remplissage de la matrice M : $\Theta(nW)$
  - Recherche de la solution : $\Theta(n)$

  (Exercice : proposez une version $\Theta(n + W)$ en espace)

- Note : L'algorithme n'est en fait pas polynomial en fonction de la taille de l'entrée
  - Si W nécessite n_w bits pour son codage, la complexité est $\Theta(n2^{n_w})$
  - Comme pour le voyageur de commerce, on n'a pas encore trouvé d'algorithme polynomial pour le problème du sac à dos (et il y a peu de chance qu'on y arrive)

Programmation dynamique : résumé

Grandes étapes :

- Caractériser la structure du problème
- Définir de manière récursive la valeur de la solution optimale
- Calculer les valeurs de la solution optimale (c'est-à-dire remplir un tableau)
- Reconstruire la (une) solution optimale à partir de l'information calculée ("bottom-up")

Applications :

- Command unix diff (comparaison de fichiers)
- Algorithme de Viterbi (reconnaissance vocale)
- Alignement de séquences d'ADN (Smith-Waterman)
- Plus court chemin dans un graphe (Bellman-Ford)
- Compilateurs (analyse syntaxique et optimisation du code)
- ...

Programmation dynamique versus diviser-pour-régner

- L'approche Diviser-pour-régner décompose aussi le problème en sous-problèmes
- Mais ces sous-problèmes sont significativement plus petits que le problème de départ ($n \rightarrow n/2$)
  - Alors que la programmation dynamique réduit généralement un problème de taille n en sous-problèmes de taille $n - 1$
- Et ces sous-problèmes sont indépendants
  - Alors qu'en programmation dynamique, ils se recouvrent
- Pour ces deux raisons, la récursivité ne fonctionne pas pour la programmation dynamique

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Algorithme glouton (greedy)

- Utilisé pour résoudre des problèmes d'optimisation (comme la programmation dynamique)
- Idée principale :
  - Quand on a un choix local à faire, faire le choix (glouton) qui semble le meilleur tout de suite (et ne jamais le remettre en question)
- Pour que l'approche fonctionne, le problème doit satisfaire deux propriétés :
  - Propriété des choix gloutons optimaux : On peut toujours arriver à une solution optimale en faisant un choix localement optimal
  - Propriété de sous-structure optimale : Une solution optimale du problème est composée de solutions optimales à des sous-problèmes
- Même si ces propriétés ne sont pas satisfaites, l'approche gloutonne peut parfois fournir une approximation intéressante au problème
- Parfois, il est possible de caractériser la distance de la solution gloutonne à la solution optimale

Exemple 1 : rendre la monnaie

- Objectif : Etant donné des pièces de 1, 2, 5, 10, et 20 cents, trouver une méthode pour rembourser une somme de x cents en utilisant le moins de pièces possible.
- Exemple : 34 cents :
  - 1ère possibilité : $\{1, 1, 2, 5, 5, 20\} \rightarrow 6$ pièces
  - 2ième possibilité : $\{2, 2, 10, 20\} \rightarrow 4$ pièces

- Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser
- Exemple : 49 cents $\rightarrow \{20, 20, 5, 2, 2\}$ (5 pièces)

Implémentation

- Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser

```
CoinchangingGreedy(x, c, n)
1  // c[1..n] contains the n coin values in decreasing order
2  Let s[1..n] be a new table
3  // s[i] is the number of ith coin in solution
4  CoinCount = 0
5  for i = 1 to n
6        s[i] = ⌊x/c[i]⌋
7        x = x − s[i] ∗ c[i]
8        CoinCount = CoinCount + s[i]
9  return (s, CoinCount)
```

- Complexité : $O(n)$ (sans compter le tri des pièces)
- Cet algorithme permet-il de trouver une solution optimale ?

Analyse de CoinChangingGreedy

Théorème : l'algorithme CoinChangingGreedy est optimal pour $c = [20, 10, 5, 2, 1]$

Preuve :

- Soit $S^*(x)$ l'ensemble optimal de pièces pour un montant x et soit c^* le plus grand $c[i] \leq x$. On doit montrer que :
  1. $S^*(x)$ contient c^* *(propriété des choix gloutons optimaux)*
  2. $S^*(x) = \{c^*\} \cup S^*(x - c^*)$ *(propriété de sous-structure optimale)*

- Propriété (2) découle directement de (1)
  - $S^*(x)$ contient c^* par (1)
  - Donc $S^*(x) \setminus \{c^*\}$ représente le change pour un montant de $x - c^*$
  - Ce change doit être optimal sinon $S' = \{c^*\} \cup S^*(x - c^*)$ serait une meilleure solution que $S^*(x)$ pour un montant de x
  - On a donc $S^*(x) = \{c^*\} \cup S^*(x - c^*)$

- Propriété (1) : $S^*(x)$ contient c^*
  - Avec $c = [20, 10, 5, 2, 1]$, une solution optimale ne contient jamais :
    - plus d'une pièce de 1, 5, ou de 10 (car $2 \times 1 = 2$, $2 \times 5 = 10$, $2 \times 10 = 20$)
    - plus de deux pièces de 2 (car $3 \times 2 = 5 + 1$)
  - Analysons les différents cas pour x :

$x = 1$: $c^* = 1$, meilleure solution $S^*(x) = \{1\}$ contient c^*

$2 \leq x < 5$: $c^* = 2$, avec un seul 1, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 5$: $c^* = 5$, meilleure solution $S^*(x) = \{5\}$ contient c^*

$5 < x < 10$: $c^* = 5$, avec un seul 1, et deux 2, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 10$: $c^* = 10$, meilleure solution $S^*(x) = \{10\}$ contient c^*

$10 < x < 20$: $c^* = 10$, avec un seul 1, deux 2, et 1 seul 5, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 20$: $c^* = 20$, meilleure solution $S^*(x) = \{20\}$ contient c^*

$x > 20$: $c^* = 20$, avec un seul 1, deux 2, 1 seul 5 et 1 seul 10, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

  - $S^*(x)$ contient donc toujours bien c^*

□

Analyse de CoinChangingGreedy

- L'approche greedy n'est correcte que pour certains choix particuliers de valeurs de pièces
  - ok pour la plupart des monnaies courantes, euros, dollars...
- Contre-exemple : $C = [1, 10, 21, 34, 70, 100]$ (valeurs de timbres aux USA) et $x = 140$
  - Algorithme glouton : $100, 34, 1, 1, 1, 1, 1, 1$
  - Solution optimale : $70, 70$

- Solution pour résoudre le cas général : programmation dynamique
- Très proche du problème de découpage de tige et du sac à dos

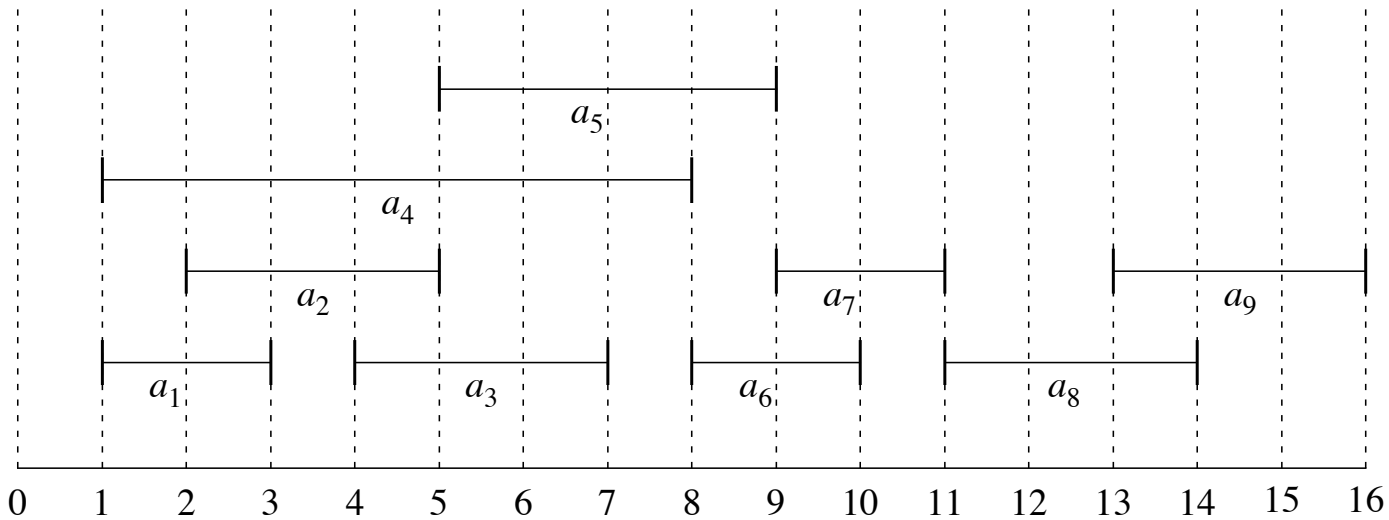
(Exercice : écrivez une fonction CoinChangeDP*)*

Exemple 2 : sélection d'activités

- Un salle est utilisée pour différentes activités
  - Soit $S = \{a_1, a_2, \ldots, a_n\}$ un ensemble de n activités
  - a_i démarre au temps s_i et se termine au temps f_i
  - Deux activités a_i et a_j sont compatibles si soit $f_i \leq s_j$, soit $f_j \leq s_i$

  Problème : trouver le plus grand sous-ensemble de tâches compatibles
- Exemple :

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

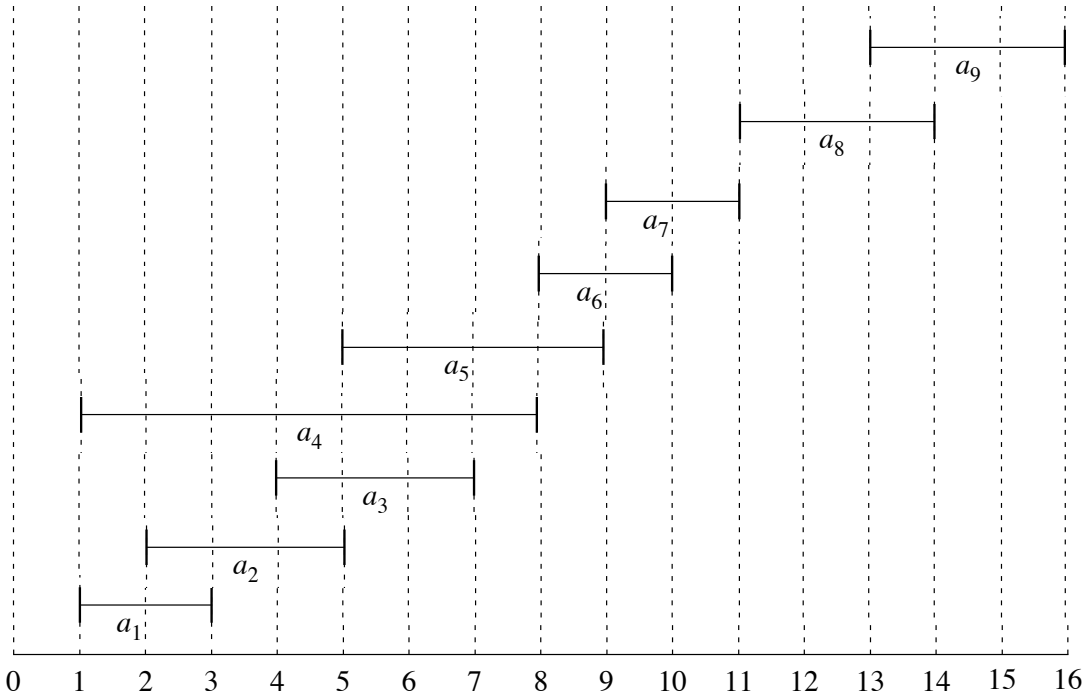


Sélection d'activités : approche gloutonne

- Schéma d'une solution gloutonne :
  - définir un ordre "naturel" sur les activités
  - sélectionner les activités dans cet ordre pour autant qu'elles soient compatibles avec celles déjà choisies
- Exemples : trier les activités selon s_i (début), selon f_i (fin), selon $f_i - s_i$ (durée), nombre de conflits avec d'autres activités...

Sélection d'activités : approche gloutonne

- Schéma d'une solution gloutonne :
  - définir un ordre "naturel" sur les activités
  - sélectionner les activités dans cet ordre pour autant qu'elles soient compatibles avec celles déjà choisies
- Exemples : trier les activités selon s_i (début), selon f_i (fin), selon $f_i - s_i$ (durée), nombre de conflits avec d'autres activités...
- Montrer par des contre-exemples que seul le tri selon f_i fonctionne



Sélection d'activités : approche gloutonne

- Considérer les activités par ordre croissant de f_i et sélectionner chaque activité compatible avec celles déjà prises
- Implémentations : en supposant s et f ordonnés selon f

```
REC-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]
3        m = m + 1
4  if m ≤ n
5        return {a_m} ∪ ...
6              ...REC-ACTIVITY-SELECTOR(s, f, m, n)
7  else return ∅
```

```
ITER-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5        if s[m] ≥ f[k]
6              A = A ∪ {a_m}
7              k = m
8  return A
```

Appel initial :

REC-ACTIVITY-SELECTOR$(s, f, 0, s.length)$

- Complexité : $\Theta(n)$ $(+\Theta(n \log n)$ pour le tri selon $f_i)$

Sélection d'activités : analyse

- La solution gloutonne est-elle correcte ?

- 1. Propriété des choix gloutons optimaux : Soit $a_x \in S$ tel que $f_x \leq f_i$ pour tout $a_i \in S$. Il existe une solution optimale OPT^* qui contient a_x.
- Preuve :
  - Soit une solution optimale OPT telle que $a_x \notin OPT$
  - Soit a_m l'activité qui se termine en premier dans OPT
  - Construisons $OPT^* = (OPT \setminus \{a_m\}) \cup \{a_x\}$
  - OPT^* est valide :
    - Toute activité $a_i \in OPT \setminus \{a_m\}$ débute en un temps $s_i \geq f_m$
    - Par définition de a_x, $f_m \geq f_x$ et donc pour tout activité a_i, $s_i \geq f_x$
    - Toute activité a_i est donc compatible avec a_x
  - OPT^* est donc optimale puisque $|OPT^*| = |OPT|$

Sélection d'activités : analyse

- La solution gloutonne est-elle correcte ?

- 2. Propriété de sous-structure optimale : Soit $a_x \in S$ le choix glouton et $S' = \{a_i | s_i \geq f_x\}$ les activités de S compatibles avec a_x. Soit $OPT^* = \{a_x\} \cup OPT'$. Si OPT' est une solution optimale pour S' alors OPT^* est une solution optimale pour S.
- Preuve :
  - Soit OPT une solution optimale pour S
  - Si OPT^* n'est pas une solution optimale pour S, alors $|OPT^*| < |OPT|$ et donc aussi $|OPT'| < |OPT| - 1$
  - Soit a_m l'activité qui se termine en premier dans OPT et $\bar{S} = \{a_i | s_i \geq f_m\}$
  - Par construction, $OPT \setminus \{a_m\}$ est une solution pour $\bar{S}$
  - Par construction, $\bar{S} \subseteq S'$ et $OPT \setminus \{a_m\}$ est une solution valide pour S' (pas nécessairement optimale)
  - Ce qui veut dire qu'il existe une solution pour S' de taille $|OPT| - 1$, ce qui contredit $|OPT'| < |OPT| - 1$ et OPT' optimal pour S' (par hypothèse).

Problèmes similaires

D'autres problèmes similaires pour lesquels il existe un algorithme glouton :

- Allocation de resources :
  - Etant donnée un ensemble d'activités S avec leurs temps de début et de fin, trouver le nombre minimum de salles permettant de les réaliser toutes
- Planification de tâches :
  - Soit un ensemble de tâches avec leur durée et l'instant auquel elles doivent chacune être terminées (leur deadline)
  - Sachant qu'on ne peut exécuter qu'une seule tâche simultanément, trouver l'ordonnancement de ces tâches qui minimise le dépassement maximal des deadlines associées aux tâches (latence).

Exemple 3 : problème du sac à dos

Rappel : problème (0/1) du sac à dos :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeur $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in \{0, 1\}$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Solution par programmation dynamique : $\Theta(nW)$

Peut-on le résoudre par une approche gloutonne ?

Programmation dynamique versus approche gloutonne

- Rappel du transparent 430 :
  - soit $M(k, w)$, $0 \leq k \leq n$ et $0 \leq w \leq W$, le bénéfice maximum qu'on peut obtenir avec les objets $1,\ldots,k$ de S et un sac à dos de charge maximale w. On a :

$$M(k,w) = \begin{cases} 0 & \text{si } i = 0 \\ M(k-1,w) & \text{si } p_i > w \\ \max\{M(k-1,w), v_k + M(k-1,w-p_k)\} & \text{sinon} \end{cases}$$

- Approche gloutonne : consisterait à remplacer le max par le choix qui nous semble le meilleur localement
- Quels choix possibles ?
  - Le moins lourd, le plus lourd ?
  - Le moins coûteux, le plus coûteux ?
  - Le meilleur rapport valeur/poids ?

Approche gloutonne

- Idée d'algorithme :
  - Ajouter à chaque itération l'objet de rapport $\frac{v_i}{p_i}$ maximal qui rentre dans le sac
  - Implémentation très proche du problème de change : $\Theta(n \log n)$
- Est-ce que ça fonctionne ? Non !

i	v_i	p_i	v_i/p_i
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,7
5	28	7	4

W=11 :

- Solution greedy : $\{5, 2, 1\} \Rightarrow$ valeur=35
- Solution DP : $\{4, 3\} \Rightarrow$ valeur=40

Problème fractionnel du sac à dos (fractional knapsack)

Par rapport au problème 0/1, il est maintenant permis d'inclure des fractions d'objets (≤ 1) :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeur $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in [0, 1]$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Exemple :

i	v_i	p_i	v_i/p_i
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,7
5	28	7	4

W=11 :

- Solution optimale 0/1 : $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$, $x_5 = 0 \Rightarrow$ valeur=40
- Solution optimale fractionnelle : $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 2/3$, $x_5 = 1$ $\Rightarrow$ valeur=42,66

Algorithme glouton

- Pour la version fractionnelle, l'algorithme glouton est optimal
- Implémentation :

```
FracKnapSack(p, v, n, W)
1  // Assume the objects are sorted according to v[i]/p[i]
2  Let x[1..n] a new table
3  w = 0
4  for i = 1 to n
5        d = min(p[i], W − w)
6        w = w + d
7        x[i] = d/p[i]
8  return x
```

- Complexité : $\Theta(n)$ ($+ \Theta(n \log n)$ pour le tri)

Correction

Théorème : Le problème fractionnel du sac à dos possède la propriété des choix gloutons optimaux

Preuve :

- Soit deux objets i et j tels que
$$\frac{v_i}{p_i} > \frac{v_j}{p_j}$$
- Etant donné un choix $(x_1, x_2, \ldots, x_n)$, on le transforme en $(x'_1, x'_2, \ldots, x'_n)$ tel que :
  - $\forall k \in [1, n] \setminus \{i, j\} : x'_k = x_k$,
  - $x'_i = x_i + \frac{\Delta}{p_i}$, et
  - $x'_j = x_j - \frac{\Delta}{p_j}$,

  où $\Delta = \min(p_i(1 - x_i), p_j x_j)$.
- Cette transformation ne modifie pas le poids total, mais améliore le bénéfice.
- On en déduit qu'il est toujours avantageux de prendre la fraction maximale de l'objet i possédant le plus grand rapport $\frac{v_i}{p_i}$. □

Algorithme glouton : résumé

- Très efficaces quand ils fonctionnent. Simples et faciles à implémenter
- Ne fonctionnent pas toujours. Leur correction peut être assez difficile à prouver

Applications :

- Arbre de couverture minimal (voir partie 7)
- Plus court chemin dans un graphe (algorithme de Dijkstra)
- Allocation de resources
- Codage de Huffman
- ...

Approche gloutonne versus programmation dynamique

- Tous deux nécessitent la propriété de sous-structure optimale
- Les algorithmes gloutons nécessitent que la propriété de choix gloutons optimaux soit satisfaite
  - On n'a pas besoin de solutionner plus d'un sous-problème
  - Le choix glouton est fait avant de résoudre le sous-problème
  - Il n'y a pas besoin de stocker les résultats intermédiares
- La programmation dynamique marche sans la propriété des choix gloutons optimaux
  - On doit solutionner plusieurs sous-problèmes et choisir dynamiquement l'un deux pour obtenir la solution globale
  - La solution doit être assemblée "bottom-up"
  - Les sous-solutions aux sous-problèmes sont réutilisées et doivent donc être stockées

Exemple 4 : codage de Huffman

- Soit une séquence S très longue définie sur base de 6 caractères : a, b, c, d, e et f
  - Par exemple, $n = |S| = 10^9$
- Quelle est la manière la plus efficace de stocker cette séquence ?
- Première approche : encoder chaque symbole par un mot binaire de longueur fixe :

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	100	101

  - 6 symboles nécessitent 3 bits par symbole
  - $3 \times 10^9/8 = 3.75 \times 10^8$bytes (un peu moins de 400Mb)
- Peut-on faire mieux ?

Idée

- Codage avec des mots de longueur fixe :

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	100	101

- Observation : l'encodage de e et f est redondant :
  - Le second bit ne nous aide pas à distinguer e de f
  - En d'autres termes, si le premier bit est 1, le second ne nous donne pas d'information et peut être supprimé
- Suggère de considérer un codage avec des mots binaires de longueurs variables

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	10	11

- Encodage et décodage sont bien définis et non ambigüs
- Permet de gagner $n_e + n_f$ bits, où n_e et n_f sont les nombres de e et de f dans la séquence

Définition du problème

- Soit un ensemble de symboles C et $f(c)$ la fréquence du symbole $c \in C$.
- Trouver un code $E : C \to \{0,1\}^*$ tel que
  - E est un code sans préfixe
    - Aucun mot de code $E(c_1)$ n'est le préfixe d'un autre mot de code $E(c_2)$
  - La longueur moyenne des mots de code est minimale

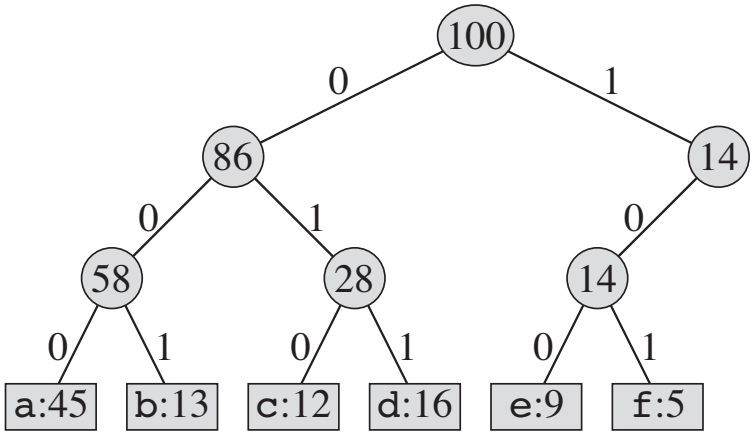
$$B(S) = \sum_{c \in C} f(c)|E(c)|$$

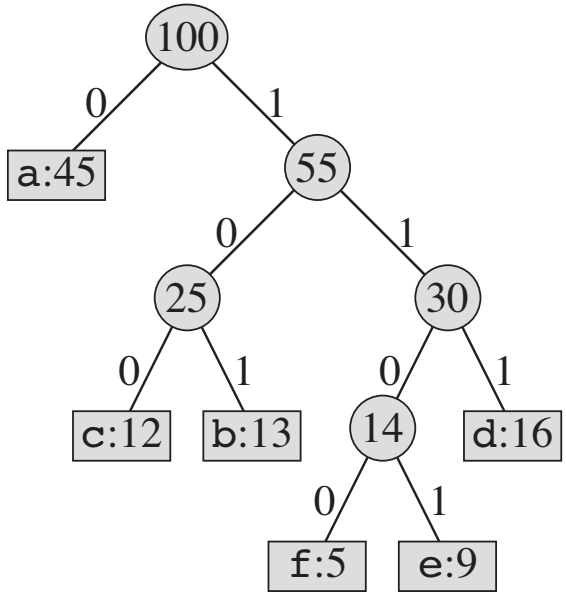
($nB(S)$ est la longueur de l'encodage de S)

- Exemple :

c	a	b	c	d	e	f	
f(c)	45%	13%	12%	16%	9%	5%	$B(S)$
Code 1	000	001	010	011	100	101	3.00
Code 2	000	001	010	011	10	11	2.86
Code 3	0	101	100	111	1101	1100	2.24

Code sans préfixe





- Un code sans préfixe peut toujours se représenter sous la forme d'une arbre binaire
  - Chaque feuille est associée à un symbole
  - Le chemin de la racine à une feuille est le code du symbole
  - La fréquence d'un nœud est la fréquence du préfixe
- Un code optimal est toujours représenté par un arbre binaire entier *(Pourquoi ?)*

Algorithme glouton

On peut montrer que le codage optimal peut être obtenu par un algorithme glouton

- On construit l'arbre de bas en haut en partant des feuilles
- A chaque étape, on fait le choix "glouton" de fusionner les deux nœuds les moins fréquents (symboles ou préfixes)

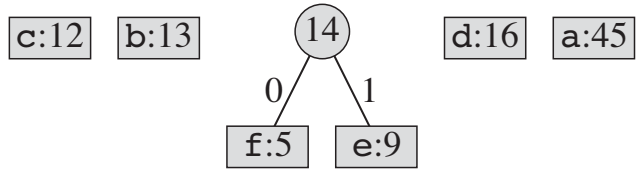
Idée de la preuve (pour information) :

- Choix gloutons optimaux :
  - Il existe un code sans préfixe optimal où les deux symboles les moins fréquents sont frères et à la profondeur maximale
  - Par l'absurde : si un tel code n'existait pas, on pourrait l'obtenir en échangeant la position des deux symboles les moins fréquents avec les feuilles les plus profondes sans augmenter $B(S)$
- Sous-structure optimale :
  - Si l'arbre qui a pour feuille le nouveau nœud issu de la fusion gloutonne est optimal, l'arbre complet est optimal
  - Plus difficile à montrer

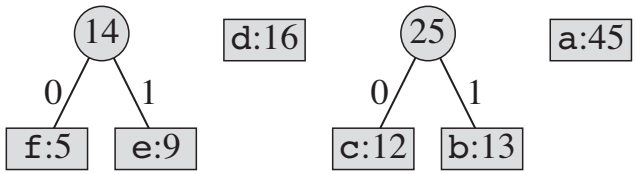
Algorithme glouton : exemple

(a)

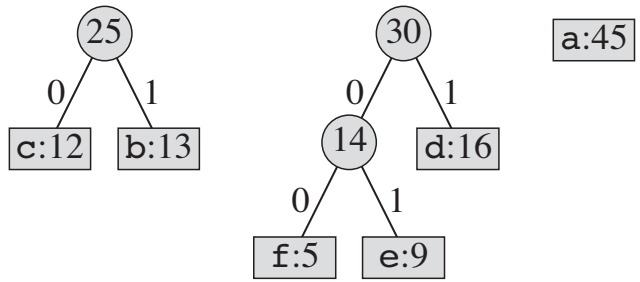
(b)



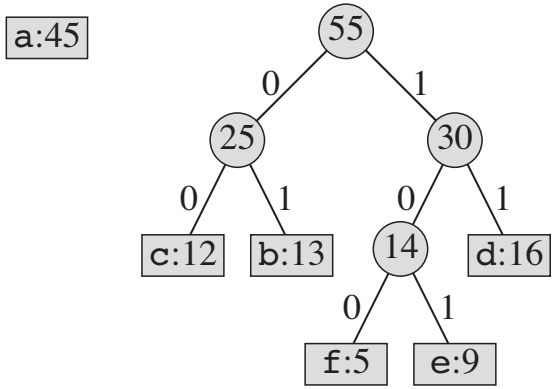
(c)



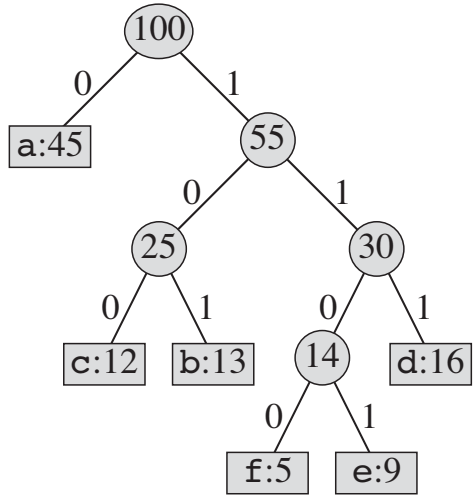
(d)



(e)



(f)



Algorithme glouton : implémentation

```
HUFFMAN(C)
1   n = |C|
2   Q =”create a min-priority queue from C”
3   for i = 1 to n − 1
4         Allocate a new node z
5         z.left = EXTRACT-MIN(Q)
6         z.right = EXTRACT-MIN(Q)
7         z.freq = z.left.freq + z.right.freq
8         INSERT(Q, z)
9   return EXTRACT-MIN(Q)
```

- Implémentation avec une file à priorité
- Complexité : $O(n \log n)$ si Q est implémentée avec un tas (min)
  - Ligne 2 : $O(n)$ si on utilise BUILD-MIN-HEAP
  - Ligne 8 : $O(\log n)$ (répétée $n - 1$ fois)

Fin

Pour aller plus loin :