ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 6 1 November 2010

1. Running time

1. Nous voyons que

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} b + \sum_{j=1}^{n}\sum_{k=1}^{j}\sum_{\ell=j+1}^{j+n} b \\ &= n\cdot b + \sum_{j=1}^{n}\sum_{k=1}^{j}\sum_{\ell'=1}^{n} b \\ &= n\cdot b + \sum_{j=1}^{n}\sum_{k=1}^{j} n\cdot b \\ &= n\cdot b + n\cdot b\cdot\sum_{j=1}^{n} j \\ &= n\cdot b + n\cdot b\cdot\frac{n\cdot(n+1)}{2} \\ &= n\cdot\ln(n) + \frac{n^2\cdot(n+1)\cdot\ln(n)}{2}. \end{aligned}$$

2. Nous avons donc

$$f(n) = \theta\big(n^3\cdot\ln(n)\big),$$

Soit $s = 3$ et $t = 1$.

Remarque : Pour les questions concernant les notations θ, O, o et Ω, il est important de pouvoir raisonner *intuitivement*, c'est à dire de pouvoir donner l'ordre de grandeur d'une fonction simplement en la regardant.

3. Le temps de parcours des lignes 4 et 8 est constant. Ainsi pour calculer le temps de parcours de l'algorithme il nous faut faire le même calcul que ci-dessus, en remplaçant $\ln(n)$ par 1, ce qui donne un temps de parcours de

$$\theta\big(n^3\big).$$

2. Diviser pour régner : Le problème du skyline

a) Nous parcourons l'axe des x de gauche à droite, en regardant les points un par un. Pour chaque point de L et de R nous devons décider s'il faut ajouter un point C, et si oui quelle va être sa hauteur.

Un algorithme est donnée ci-dessous. On remarque que le nombre d'opérations est $O(|L|+|R|)$ où $|L|$ et $|R|$ représentent les nombres de points dans les skylines de L et R.

Dans l'algorithme ci-dessous, les skylines sont données sous forme de tableau, avec les entrées étant des couples (x, h). Par convention, la dernière entrée est toujours $(\infty, 0)$ et marque la fin de la skyline.

Call : SkylineMerge
Input: Skylines L et R
Output: Skyline ret, superposition des deux skylines.

```
i_ℓ, i_r ← 0
h_ℓ ← 0, h_r ← 0, h ← 0
while min(L[i_ℓ].x, R[i_r].x) < ∞ do
  if L[i_ℓ].x < R[i_r].x then
    h_ℓ ← L[i_ℓ].h
    x ← L[i_ℓ].x
    i_ℓ ← i_ℓ + 1
  else if L[i_ℓ].x = R[i_r].x then
    h_ℓ ← L[i_ℓ].h
    h_r ← R[i_r].h
    x ← L[i_ℓ].x
    i_ℓ ← i_ℓ + 1
    i_r ← i_r + 1
  else
    h_r ← R[i_r].h
    x ← R[i_r].x
    i_r ← i_r + 1
  if max(h_ℓ, h_r) ≠ h then
    h ← max(h_ℓ, h_r)
    ret.add((x, h))
ret.add((∞, 0))
return ret
```

Dans le pseudocode ci-dessus, la méthode add ajoute un élément à la fin d'un array.

b) Remarquons d'abord que l'algorithme SkylineMerge ne résout pas le problème Skyline. En effet, Skyline attend comme input un ensemble de bâtiments (i.e. de triplets (x_1, x_2, h)) et retourne comme output le skyline correspondant. Par contre, SkylineMerge a comme input deux skylines et retourne le skyline obtenu en les combinant.

MergeSort : Notre solution est très similaire à *MergeSort*, nous rappelons donc les détails de cet algorithme. Le problème à résoudre est d'ordonner une suite de nombres naturels (l'input est donc une suite de nombres naturels, et l'output une suite qui contient les mêmes éléments, mais ordonnés). Nous avons d'abord construit un algorithme Merge qui, étant donné deux suites S_1 et S_2 de nombres naturels *ordonnées*, retourne une suite ordonnée contenant les éléments de S_1 et S_2.

L'algorithme MergeSort divise la suite donnée en 2 sous-suites (plus petites), qui sont ensuite ordonnées (avec un appel récursif à MergeSort). Puis, ces 2 sous-suites ordonnées sont combinées avec l'algorithme Merge pour obtenir une version ordonnée de la suite de l'input.

Puisque nous avons fait un appel récursif à MergeSort, chacune des sous-suites sera elle même divisée en deux sous-suites, etc. Il faut donc définir un cas de base où la récurrence se terminera. Il s'agit du cas où la suite ne contient qu'un seul élément et est donc forcément déjà ordonnée.

On l'appelle un algorithme diviser-pour-régner puisqu'on divise le problème en deux instances plus petites du même problème. l'algorithme Merge est utilisé pour recoller les solutions de ces instances pour obtenir la solution du problème original.

Pour résoudre le problème Skyline nous procédons de la même façon. Notre algorithme (que nous appelons SkylineSolve) commence par diviser la liste de bâtiments en deux sous-listes (dont la taille vaut à peu près la moitié de celle de la suite originale). Nous trouvons ensuite le Skyline de chacune de ces suites avec un appel récursif à SkylineSolve.

Nous réduisons donc notre problème à deux sous-problèmes qui sont toujours des instances de Skyline, mais avec un input plus petit (il y a moins de bâtiments).

Les solutions de ces deux sous-problèmes (les deux skylines) sont ensuite combinées avec l'algorithme SkylineMerge donné dans la partie *a*) (ce pas correspond donc à Merge).

Comme avec MergeSort, puisqu'il y a cet appel récursif, il faut définir un cas de base. C'est le cas où il n'y a qu'un seul bâtiment, et le skyline est égal a ce bâtiment.

On obtient au final, en pseudocode :

Call : SkylineSolve
Input: Bat : Une liste de n bâtiments (de triplets (x_1, x_2, h))
Output: A : Le skyline correspondant (une suite de points (x, h))
  if $n = 1$ **then**
    return $((Bat[1].x_1, Bat[1].h), (Bat[1].x_2, 0))$
  else
    $m \leftarrow \lceil n/2 \rceil$
    $Bat_1 \leftarrow$ la sous-suite de Bat composée des $m-1$ premiers éléments de Bat
    $Bat_2 \leftarrow$ la sous-suite de Bat composée des autres éléments (bâtiments) de Bat
    $Skyline_1 \leftarrow$ SkylineSolve(Bat_1)
    $Skyline_2 \leftarrow$ SkylineSolve(Bat_2)
    return SkylineMerge$(Skyline_1, Skyline_2)$

3. Sac à dos 0/1

a) La valeur maximale est 45. Il y a deux solutions correctes : les ensembles qui atteignent ce maximum sont :

$$\{A, C, E\} \qquad \text{et} \qquad \{B, D, E\}$$

Pour les détails voir la figure 1.

Les flèches diagonales corréspondent à la branche passant par le pas 9 de l'algorithme du polycopié et les flèches verticales aux branches passant par 11 ou 14. Les flèches en gris sont les décisions rejetées parce que l'autre chemin est mieux.

Les flèches en gras donnent des chemins optimaux (flèche diagonale à la $i$ème étape : on prend l'objet i ; flèche verticale : on ne le prend pas).

b) La solution optimale n'est pas unique, c.f. le corrigé du point précédent. On peut adapter l'algorithme pour trouver toutes les solutions optimales. La méthode la plus simple est d'utiliser une récursion pour le faire (Initialement, nous appélons GetOptimalChoice2(W, n, C)).

```
Call : GetOptimalChoice2(w, i, C)
Input: L'input à Knapsack et le C calculé. x[ ] est statique.
Output: Un choix optimal.
  if i ≥ 1 then
    if c_{i-1,w} = c_{i,w} then
      x[i] ← 0
      GetOptimalChoice2(w, i − 1, C)
    if c_{i-1,w-w_i} + v_i = c_{i,w} then
      x[i] ← 1
      GetOptimalChoice2(w − w_i, i − 1, C)
  else
    Print "Solution :" x[ ]
```

c) Chaque solution (optimale ou non, admissible ou non) est un sous-ensemble de

$$\{1, 2, \ldots, n\}.$$

Donc, l'affirmation est vraie parce que $|\mathrm{Pot}(\{1, 2, \ldots, n\})| = 2^n$.

d) On peut choisir $v_i = 1$ et $w_i = 1$ pour $i = 1, \ldots, n$ et $W = n/2$. Les solutions optimales sont alors des sous-ensembles de $\{1, \ldots, n\}$ de taille $n/2$, et il y en a $\binom{n}{n/2}$. Il est facile de vrifier que

$$\binom{n}{n/2} > 2^{n/2},$$

d'où le résultat.

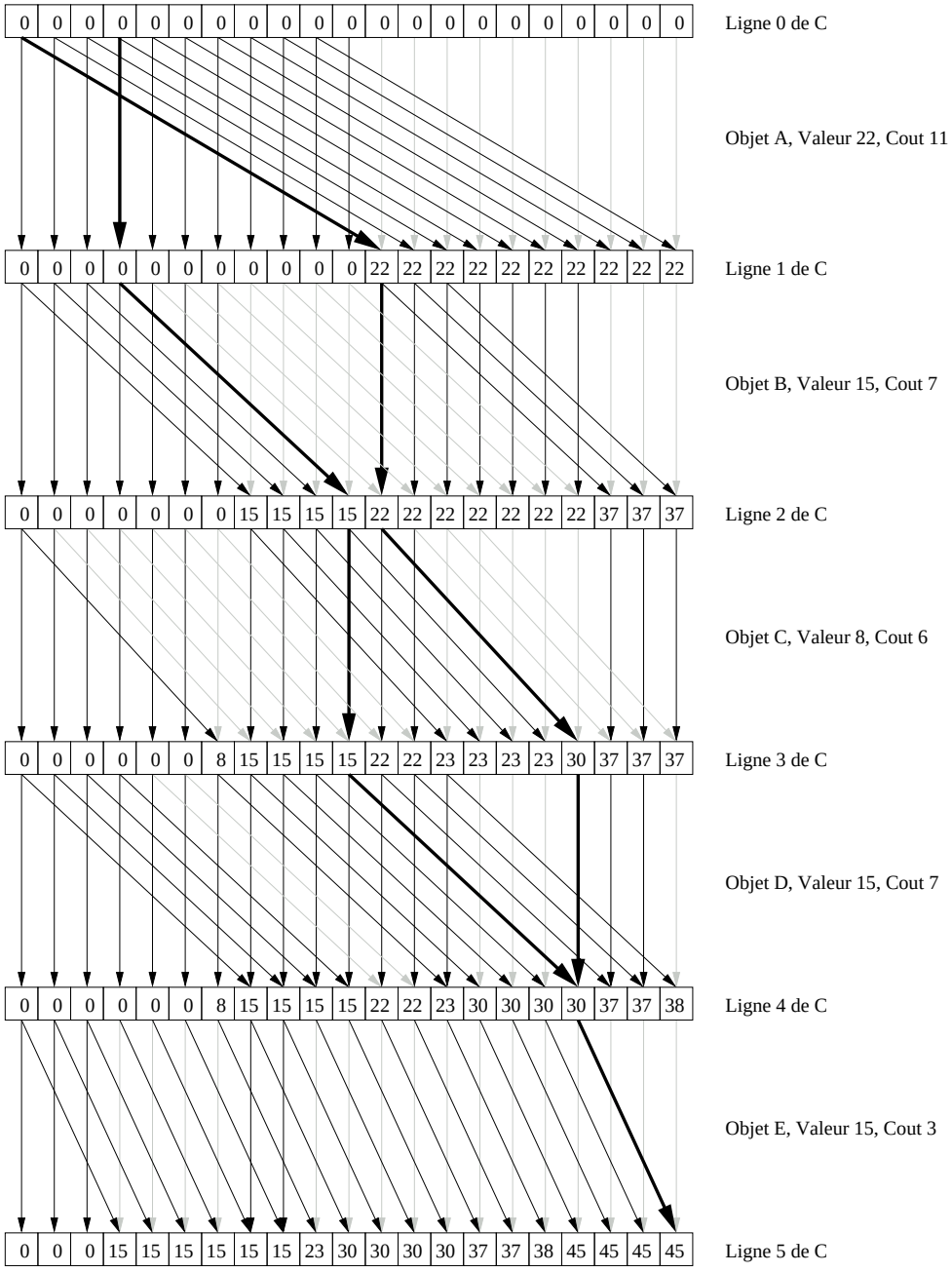

FIG. 1 – Knapsack 0/1