Enseirb, Filière Electronique, Semestre 1
Algorithmes et Structures de Données

Corrigé de l'épreuve du 20 janvier 2009

Exercice 1 : Algorithme glouton. *Barème envisagé : 10 points*

On s'interesse au problème suivant : afin de réaliser un produit une entreprise doit acquerir tous les objets d'un ensemble de n éléments $A = \{e_1, e_2, \ldots, e_n\}$, ces objets ne sont pas vendus séparément mais par des paquets chaque paquet constituant un sous-ensemble S_i de E.

Chacun de ces paquets a un coût c_i et on doit déterminer le coût minimal permettant d'acquérir tous les objets. Un exemple est le suivant :

$$S_1 = \{1, 2\}, S_2 = \{3, 4\}, S_3 = \{1, 3, 4\}, S_4 = \{1, 4\}, S_5 = \{2, 4\}$$

avec les coûts :

$$c_1 = 5, c_2 = 3, c_3 = 5, c_4 = 4, c_5 = 2$$

Un premier algorithme glouton consiste à classer les paquets par coûts moyens croissants ; le coût moyen d'un sous-ensemble S_i est le quotient de son coût par son nombre n_i d'éléments. On pose ainsi $u_i = \frac{c_i}{n_i}$. Par cet algorithme on retient d'abord l'ensemble S_i tel que u_i est minimum, puis à chaque étape on retient un nouvel ensemble ayant la plus petite valeur de u_i parmi ceux qui n'ont pas été retenus et ceci jusqu'à obtenir que chaque e_j soit dans un S_i retenu.

Question 1. Sous-ensembles retenus par l'algorithme glouton dans l'exemple considéré plus haut. Le calcul des coûts moyens donne $u_1 = 2.5, u_2 = 1.5, u_3 = 1.666, u_4 = 2, u_5 = 1$ L'algorithme glouton classe les sous-ensembles dans l'ordre S_5, S_2, S_3, S_4, S_1 il retient d'abord S_5 puis S_2 et n'ayant pas encore pris l'élement 1, il doit retenir aussi S_3 pour un coût total de 10.

Cet algorithme glouton ne donne pas l'optimum, en effet la solution de retenir S_3 et S_5 a pour coût 7 ; elle est meilleure que celle donnée par glouton.

Question 2. On suppose que les ensembles S_i sont donnés par un tableau (noté `appartient`) à deux indices et à valeurs booléennes. Ainsi `appartient[j][i]` est égal à `true` si et seulement si l'élément e_j appartient à l'ensemble S_i. L'algorithme détaillé du calcul du tableau `nb` tel que `nb[i]` soit le nombre d'éléments de l'ensemble S_i est le suivant :

```
for (i = 1; i <= n; ++i)
   nb[i] =0;
for (i = 1; i <= m; ++i)
   for (j = 1; j <= n ; ++j)
       if (appartient[j][i])
            nb[i]++;
```

On modifie l'algorithme glouton de façon à ne pas tenir compte au cours des étapes des éléments déjà acquis. Ainsi après avoir sélectionné un S_j, on recalcule le coût moyen de chaque ensemble S_i en ne tenant compte que des éléments qui n'appartiennent pas à un S_k déjà retenu. On pose alors

$$u_i = \frac{c_i}{n'_i}$$

où n'_i est le nombre d'éléments de S_i qui ne sont pas dans un ensemble déjà retenu.

Question 3. On applique le nouvel algorithme à l'exemple donné plus haut : le premier choix est de prendre S_5 comme pour l'algorithme précédent les valeurs obtenues pour les nombres d'élements n'_i une fois ce choix fait sont :

$$n'_1 = n'_2 = n'_4 = 1, \quad n'_3 = 2$$

Ainsi les nouvelles valeurs des u_i sont données par :

$$u_1 = 5, u_2 = 3, u_3 = \frac{5}{2} = 2.5, u_4 = 4$$

C'est donc S_3, ayant la plus petite valeur qui est choisi et l'algorithme se termine en donnant la solution optimale.

Question 4. Pour modifier la valeur des `nb[i]` une fois retenu l'ensemble S_j on a l'algorithme :

```
reCalcule(nb, j){
   for (k = 1; k <= n; k++)
      for (i= 1 ; i <= m; ++k)
         if(appratient[k][i] && appartient[k][j])
              nb[i]--;
```

qui modifie la valeur des `nb[i]` une fois retenu l'ensemble S_j (d'indice `j`), en utilisant le tableau `appartient`.

Question 5. Si par exemple on prend les objets $1, 2, 3$ et les ensembles $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3\}$, $S_4 = \{1, 2, 3\}$ de coûts respectifs $1, 2, 3, 5$ alors l'algorithme choisit d'abord S_1 les recalculs donnent $u_2 = 2, u_3 = 3, u_4 = 2.5$ l'algoritme choisit S_2 puis S_4 le total fait un coût de 11 alors que le seul S_3 aurait suffit, cette solution a coût 4.

Exercice 2 *Barème envisagé : 10 points*

On se propose de calculer une plus longue sous-suite croissante d'une suite de nombres en utilisant des techniques inspirées de la programmation dynamique.

Pour ceci on commence par un algorithme simple qui sera utilisé par la suite.

Question 1. Soit `u` un tableau de `n` nombres entiers (`u[0], u[1], ..., u[n-1]`) tous distincts.

Pour la fonction `precedent(i)` (où `i` est un indice compris entre 0 et `n-1`), dont le résultat est `j`, indice de l'élément du tableau situé avant `u[i]` inférieur ou égal à `u[i]` et le plus proche de lui. Le résultat doit être -1 si tous les élements situés avant `u[i]` lui sont supérieurs. Ainsi on doit avoir (si `u[i] = j` est différent de -1) :

`j < i` `u[j] < u[i]` `u[k] > u[i]` pour `j < k < i`

On propose

```
precedent(i) {
    j = i-1;
    while (j >= 0  && u[j] > u[i])
          j--;
   //Noter que la valeur de j sera -1 si aucun element n'est plus petit que u[i]
    return j;
}
```

Question 2. Une sous-suite croissante d'une suite $u_0, u_1, \ldots u_{n-1}$ est donnée par $u_{i_1}, u_{i_2}, \ldots, u_{i_p}$ tels que

$$i_1 < i_2 \ldots < i_p \qquad u_{i_1} < u_{i_2} < \ldots < u_{i_p}$$

par exemple $2, 5, 9$ est une sous-suite croissante de la suite considérée à la Question 1 mais ce n'est pas la plus longue car $2, 5, 7, 8$ est plus longue.

On se propose de donner un algorithme qui calcule une sous-suite de longueur maximale. Tout d'abord on calcule cette longueur ; pour cela on note ℓ_i.

La longueur de la plus longue sous-suite croissante dont le dernier élément est u_i, est 1 si `precedent(i) = - 1` ; en effet une telle suite ne peut pas contenir d'autre élément que u_i.

On suppose calculé ℓ_j, la valeur de ℓ_i si `precedent(i) = j`, la première impression est que $\ell_i = \ell_j + 1$ mais ce n'est pas toujours le cas, prenons en effet l'exemple suivant : $u = 1, 3, 4, 2, 8$. On a $u_4 = 8$ et `precedent(4) = 3` car `u[3] = 2 < u[4]` mais $\ell_3 = 2$ car la sous-suite croissante qui se termine par `2` est `1,2` par contre $\ell_4 = 4$, la suite de longeur 4 `1,3,4,8` se termine en effet en 8.

On a donc si `precedent(i) = j` la valeur de ℓ_i

$$\ell_i = max_{k<=j, u_k<u_i}(1 + \ell_k)$$

Question 3. Algorithme qui calcule les ℓ_i en utilisant la fonction `precedent` donné à la Question 1.

```
ell[0] = 1;
for (i = 1; i <n ; ++i)
  j =  precedent[i];
  if (j == -1) ell[i] = 1;
  else {
     ell[i] = ell[j] + 1;
     for (k = j; k >= 0; k--)
       if (u[k] < u[i] && ell[k] >= ell[i])
         ell[i] = ell[k] +1
    }
```

Question 4. L'algorihme précédent afin qu'il affiche une suite de longueur maximale peut être complété en constituant un tableau `pred` des prédécesseurs de chaque u_i dans une plus longue sous-suite croissante qui se termine par u_i.

```
ell[0] = 1;
for (i = 1; i <n ; ++i)
  j =  precedent[i];
  if (j == -1) {
      ell[i] = 1; pred[i] = i;}
  else {
     ell[i] = ell[j] +1
     pred[i] = j;
     for (k = j; k >= 0; k--)
       if (u[k] < u[i] && ell[k] >= ell[i]){
         ell[i] = ell[k] +1;
         pred[i] = k;
    }
```

On calcule ensuite le i tel que ℓ_i est maximum et on remonte de predécesseur à prédécesseur.

```
max = 0;
for (i =1 ; i < n; ++i)
  if (ell[i] > ell[max])
    max = i;
j = max;
affiche(j);
while (j != pred[j]){
    affiche(pred[j]);
    j = pred[j];
}
```