IUT de Montpellier **TP : programmation dynamique (DP)**
R4.A.04 Qualité Algorithmique

Il n'y a pas de code à récupérer pour ce TP (pensez à écrire des tests!).

Exercice 1. Nombre de chemins dans une grille

On considère une grille rectangulaire de largeur $l \geq 1$ et de hauteur $h \geq 1$. On appelle *chemin monotone* dans une grille un chemin

- qui part de la case en haut à gauche
- qui termine en bas à droite
- et qui, à chaque étape, avance soit d'une case vers le bas, soit d'une case vers la droite

Question 1.1.
Combien y-a-t-il de chemins monotones dans une grille 2×3 ?

Question 1.2.
Ecrire un algorithme `nbCheminsAux(int l,int h)` qui calcule le nombre de chemins monotones dans une grille $l \times h$.

Question 1.3.
A partir de quelle valeur de x observez vous que `nbCheminsAux(x,x)` prend plus d'une minute ?

Question 1.4.
Transformez `nbCheminsAux` en DP :

- Ecrire `nbCheminsDPAux(int[][]t, int l,int h)` qui utilise la même stratégie que `nbCheminsAux`, mais en se servant du tableau de mémoïsation t.
- Ecrire `nbCheminsClient(int l, int h)` qui prépare le tableau t et fait appel à `nbCheminsDPAux`.

Question 1.5.
Quelle est la complexité de `nbCheminsClient(l,h)` ?

Question 1.6.
Testez `nbCheminsClient(x,x)` (avec le x de la question précédente), pour comparer!

Question 1.7.
(Bonus) Sur cet exercice, on peut être malin et trouver une formule qui nous donne directement le nombre de chemins monotones dans une grille $l \times h$.

Exercice 2. Découpe de planche[1]

On considère une scierie qui connaît le prix de vente p_i pour une planche longueur i. Lorsqu'elle reçoit une planche de longueur n, elle peut soit en tirer le prix p_n, soit la découper en k morceaux de longueur $i_1, \ldots, i_k$ (avec $\sum_{\ell=1}^{k} i_l = n$) et en tirer $\sum_{\ell=1}^{k} p_{i_l}$.

On considère les spécifications suivantes pour l'algorithme `int decoupeAux(int[] p, int i)` : étant donné

- un tableau p tel que $p[i] = p_i$ pour $1 \leq i \leq n$ (et $p[0] = 0$)
- un i avec $0 \leq i \leq n$,

calcule le meilleur prix que l'on puisse tirer d'une planche de taille i. Par exemple, pour $p = [0, 1, 6, 8, 9]$ et $i = 4$, `decoupeAux(p,4)` devra retourner 12.

Question 2.1.
Ecrire `decoupeAux(p,i)` (sans pour l'instant le transformer en programmation dynamique). Indication : vous ne savez pas où placer votre premier trait de coupe ? Branchez ..

Question 2.2.
Transformez l'algorithme précédent en DP (deux algorithmes à écrire).

---

[1] Exercice tiré de transparents d'Oliviez Bournez

Question 2.3.
Quelle est la complexité de la DP obtenue ?

Nous allons maintenant faire des modifications pour obtenir une solution, et pas seulement sa valeur.

Question 2.4.
Ecrire une classe `SolutionDecoupe` avec au minimum :

- un attribut `ArrayList<Integer> l` représentant la séquence des traits de découpe, remarquez que on s'imposera :
  - pour tout $0 < i < l.length$, $1 \leq l.get(i) < p.length$
  - si la planche est vide, alors on affectera l à l'arrayList vide (et pas à null, ni une arrayList contenant 0)
- un constructeur `SolutionDecoupe(SolutionDecoupe s)` qui fera la recopie en profondeur de l'arrayList de s (vous verrez pourquoi)
- une méthode `add(int j)` qui ajoute j à la solution
- une méthode `getPrice(int[]p)` qui retourne le prix de la solution vis à vis des prix dans p

Question 2.5.
En s'inspirant de votre ancienne DP, écrire une DP qui calcule une solution. Indications :

- Le tableau de mémoïsation sera donc du type `SolutionDecoupe[]`.
- Attention, quand dans votre branchement vous faites un appel récursif `tmp = .. appelrec(..)`, pensez ensuite à faire une copie en profondeur de `tmp`. En effet, `tmp` est la solution stockée dans la case correspondant aux paramètres de l'appel récursif, et on ne veut surtout pas la modifier!

Exercice 3. Ordonnancement sur 2 machines

On considère le problème suivant d'ordonnancemnt de tâches indépendantes sur 2 machines. Les tâches sont ordonnancées depuis le temps 0, et sans interruptions entre les tâches.

- entrée : un tableau t de n entiers positifs représentant les durées des tâches (la ième tâche dure $t[i]$)
- sortie : une partition des tâches en deux ensembles M_1 et M_2 (M_i contient les indices des tâches sur la machine i)
- fonction objectif :
  - on note $C_i = \sum_{i \in M_i} t[i]$ la date de fin sur la machine i
  - le but est de minimiser $C = \min(C_1, C_2)$ correspondant à la date de fin de la dernière tâche

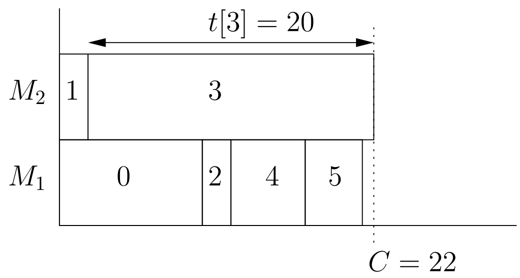

Figure 1: Sur cet exemple avec $t = [10, 2, 2, 20, 5, 4]$, l'optimal est 22, correspondant à $M_1 = [0, 2, 4, 5]$ et $M_2 = [1, 3]$.

On va maintenant écrire une DP qui résoud optimalement ce problème. Une stratégie de branchement possible est la suivante : on parcours les tâches en partant de 0, et pour chaque tâche i, on branche pour savoir si on place i sur la machine 1, ou sur la machine 2. Pour transformer cette stratégie "naïve" de branchement en une DP efficace, il faut réflechir aux informations "minimales" qu'il suffit de donner à la récurrence.

Question 3.1.
Pourquoi il ne faut surtout *pas* écrire une DP ainsi ?

```
static int ordoAuxBAD(int []t, int i, ArrayList<Integer> M1, ArrayList<Integer> M2){
//prerequis
        //t contient des entiers positifs
        //0 <= i <= t.length
        //M1 U M2 = {0,..,i-1}
        //(M1 et M2 stockent les décisions prises pour les tâches {0,..,i-1}

//retourne la valeur optimale pour ordonnancer les taches de t[i..t.length]
        //en supposant que la machine i contient déjà des tâches Mi
}
```

Question 3.2.
Ecriez la DP correctement (on se contentera de calculer la valeur de l'optimal). Vous devriez avoir une complexité en $\mathcal{O}(nC^2)$, où $C = \sum_{i=1}^{n} t[i]$.