POLYTECH MONTPELLIER

IG 4 - Algorithmique avancée

TD sur la programmation dynamique

Dans tout le sujet, on utilise les notations suivante pour un tableau t. La notation $t[i..j]$ (avec $0 \leq i \leq j < t.length$) dénote le sous tableau compris entre les indices i et j (inclus tous les deux), et la notation $t[i..]$ dénote $t[i..(t.length-1)]$. Par exemple avec $t = [5, 2, 4, 9, 8]$, $t[1..3] = [2, 4, 9]$, et $t[3..] = [9, 8]$. Ces notations sont étendues aux String.

Exercice 1. *Complexité d'une DP*

On considère la fonction suivante (ne cherchez pas à comprendre ce qu'elle calcule, remarquez juste qu'elle est correctement définie, au sens où elle termine toujours).

```
public static int f(int i, int j){
  //0 <= i <= n1
  //0 <= j <= n2 (n1 et n2 sont des var globales avec 0 <= n1 <= n2)
  //0 <= i <= j
  if(i==0 || i==j){
      return 1;
  }
  else{
      return f(i,j-1)+f(i-1,j-1);
  }
}
```

1. Supposons que l'on transforme f en DP en ajoutant un tableau de mémoisation des résultats. Quelle est maintenant la complexité de f en fonction de n_1 et n_2 ? (justifiez en utilisant un théorème du cours)

Exercice 2. *Transformation en DP*

On considère le problème suivant (noté $P2||C_{max}$) 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 représente 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 $\max(C_1, C_2)$ correspondant à la date de fin de la dernière tâche

Par exemple, pour $t = [10, 2, 2, 20, 5, 4]$, l'optimal est 22, correspondant à $M_1 = [2, 3]$ et $M_2 = [0, 1, 4, 5]$. Soit $n = t.length$, et $S = \sum_{i=0}^{n-1} t[i]$ la somme des durées de toutes les tâches.
Pour résoudre ce problème, on va brancher pour chaque i en essayant de mettre la tâche i sur M_1 ou sur M_2. Il faudra donc indiquer dans la récurrence la charge (appelée load_i) déjà accumulée sur chaque machine i. On obtient donc l'algorithme suivant:

```
public static int P2CMAXAUX(int []t, int i, int load1, int load2){
//prerequis
        //t contient des entiers positifs
        //0 <= i <= n
        //0 <= load1 <= S et 0 <= load2 <= S

//retourne la valeur optimale pour ordonnancer les taches de t[i..t.length]
        //en supposant que la machine i contient déjà des tâches entre 0 et loadi
        if(i==t.length){
                return max(load1,load2);
        }
        else{
                return min(P2CMAXAUX(t,i+1,load1+t[i],load2),P2CMAXAUX(t,i+1,load1,load2+t[i]));
        }
}
```

Comme dans le cours, on pourrait prouver que P2CMAXAUX résoud optimalement le problème auxiliaire suivant:
Problème $P_2||C_{max} - AUX$

- entrée : trois entiers i,$load_1$,$load_2$ avec $0 <= i <= n$, $0 <= load_i <= S$
- sortie : une partition des tâches $\{i, \ldots, n-1\}$ en deux ensembles M_1 et M_2 (M_i représente les indices des tâches sur la machine i)
- fonction objectif :
  - on note $C_i = load_i + \sum_{i \in M_i} t[i]$ la date de fin sur la machine i
  - le but est de minimiser $\max(C_1, C_2)$ correspondant à la date de fin de la dernière tâche

1. Transformez la fonction "P2CMAXAUX" en programmation dynamique en ajoutant la méthode "cliente" associée (et le tableau de mémoïsation) comme vu en TD/cours. Donnez la complexité de la programmation dynamique obtenue.

2. Transformez la méthode "P2CMAXAUX" (celle fournie de base, pas celle de la question précédente) en une méthode "P2CMAXAUX-V2" pour qu'elle calcule maintenant une solution et plus seulement sa valeur. P2CMAXAUX-V2 sera donc juste une méthode récursive "classique" (sans aucune instruction liée à la programmation dynamique). Vous pouvez utilisez un type de retour un tableau de deux ArrayList<Integer>. On rappelle que l'on peut manipuler une ArrayList<Integer> ainsi :

- ArrayList<Integer> liste = new ArrayList<Integer>(); //construit une liste vide
- int x = liste.get(i); //pour obtenir le ième element, avec 0 <= i <liste.size();
- liste.add(z); //pour ajouter un entier z à liste

Exercice 3. *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 decoupe(int[] p, int i)` : étant donné

- un tableau p indexé de 1 à n tel que $p[i] = p_i$
- 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 $p = [1, 5, 8, 9]$ et $n = 4$, `decoupe(p,4)` devra retourner 10.

1. Ecrire récursivement $decoupe(p, i)$ (sans pour l'instant le transformer en programmation dynamique).

[1] Exercice tiré de transparents d'Oliviez Bournez

Nous allons maintenant prouver que decoupe respecte bien ses spécifications (c'est à dire retourne bien une valeure optimale). Reformulons les choses en plus formel en définissant le problème de maximisation $\Pi_{DECOUPE}$

- entrée : un couple (p, i) tel que spécifié dans decoupe
- sortie : un ensemble $S = \{i_1, \ldots, i_k\}$ d'entiers non nuls tels que $\sum_{x \in S} x = i$ ($S = \emptyset$ autorisé quand $i = 0$)
- fonction objectif : maximser $f(S) = ...$

2. Que vaut $f(S)$?

Le but est donc de montrer que pour toute entrée (p, i) de $\Pi_{DECOUPE}$, $decoupe(p, i) = opt(p, i)$

3. Etant donné une entrée (p, i) de découpe, on considère une solution optimale S^*. Soit l^* un des entiers de S^*. Ecrivons $S^* = \{l^*\} \cup S'$ (S' peut éventuellement être vide si S^* ne contenait qu'un entier $l^* = i$). Montrez que $f(S') \leq opt(p, i - l^*)$. En déduire que $opt(p, i) \leq ..$

4. En déduire que $decoupe(p, i) \geq opt(p, i)$, et donc est égal.

5. Modifiez l'algorithme précédent pour en faire une DP que l'on appelle DPdecoupe.

6. Quelle est la complexité de DPdecoupe(p,n) ?

7. Modifiez DPdecoupe pour qu'il retourne également la solution (sous forme d'une liste par exemple).

Exercice 4. *Decomposition d'une chaîne en blocs*
On considère un alphabet réduit deux caractères $A = \{a, b\}$. On considère le problème suivant de décomposition d'une chaîne en un minimum de patterns.

- entrée : une chaine s sur l'alphabet A, et une liste de mots p (aussi sur l'alphabet A) appelés les patterns. On supposera que p contient (au moins) "a" et "b".
- sortie : un découpage de s en k blocs $b_i, i \in [k]$ tels que
  - $s = b_1.b_2 \ldots .b_k$, où . dénote la concaténation de chaînes
  - pour tout $i \in k$, $b_i \in p$
- fonction objectif : minimiser k

Par exemple, pour $s =$ "$baabaaaa$" et $p = [$"a","b","aab","$aaaa$","aba","aaa"$]$, les deux décompositions suivantes sont possibles (mais il y en a d'autres)

- $s =$ "b"."aab"."$aaaa$" (on a donc $b_1 =$ "b", $b_2 =$ "aab", $b_3 =$ "$aaaa$", et donc une solution de coût 3, qui est d'ailleurs optimale)
- $s =$ "b"."a"."aba"."aaa" (on a donc $b_1 =$ "b", $b_2 =$ "a", $b_3 =$ "aba", $b_4 =$ "aaa", et donc une solution de coût 4)

On remarque que l'on a imposé que p contienne au moins "a" et "b" afin d'être sûr qu'il y ait toujours une solution avec tous les b_i de taille 1.

1. On considère l'algorithme glouton G qui part de la gauche de s, et qui a chaque étape choisit un $b_i \in P$ de longueur la plus grande possible. Sur l'exemple ci-dessus $G(s, p)$ retournerait 3, puisqu'il construirait la première des deux solutions. Constuire une instance (s', p') sur laquelle G est le plus mauvais possible. Remarque : essayez d'abord d'obtenir une instance sur laquelle G n'est pas optimal, puis essayez de montrer que pour toute constance c il existe une instance (s', p') telle que $G(s', p') \geq c \times opt(s', p')$.

On va maintenant écrire une DP qui résoud optimalement ce problème:

```
public static int minPattern(int i){
//prerequis
        //(s,p) est une entree du problème,
        //et ces tableaux sont déclarés à l'exterieur de minPattern
        //0 <= i <= s.length()

//retourne le plus petit nombre de blocs pour décomposer la sous chaine de s commençant en i.
        //Plus formlement, retourne le plus petit nombre k de blocs b1..bk tels que
        //s[i..]=b1..bk, et les bi sont dans p
}
```

Dans l'exemple pour $s =$ "*baabaaaa*" et $p =$ ["*a*","*b*","*aab*","*aaaa*","*aba*","*aaa*"] ci-dessus, minPattern(3) doit retourner 2.

2. Ecrire l'algorithme minPattern.

Remarque : on ne demande pas d'ajouter le tableau de mémoisation, écrivez simplement un algorithme récursif pour pourrait être facilement transformé en une DP polynomiale. Remarque : vous pouvez utilisez les primitives suivantes :

- s.length(), qui dénote le nombre de caractères de s (par ex $s.length()$ retourne 7 dans l'exemple ci-dessus)
- s.subString(i,j) qui pour tout $0 \leq i \leq j < s.length()$ retourne $s[i..j]$ (par ex $s.subString(2,4)$ retourne "*baa*" dans l'exemple ci-dessus)
- p.contains(b), avec p de type ListeString et b un String, qui retourne vrai ssi p contient b

Exercice 5.*Interval Stabbing*
On considère le problème suivant de "Interval Stabbing" qui consiste, étant donné n intervalles (dessinés horizontalement), à trouver le minimum de lignes verticales qui "percent" tous ces intervalles. Plus formellement

- entrée : n intervalles avec $I_i = [d_i, f_i]$, avec d_i et f_i dans $\mathbb{N}$
- sortie : un ensemble d'entiers S tel que pour tout $i \in [1, n]$, il existe $k \in S \cap I_i$ (on dit que I_i est percé par k)
- objectif : minimiser $|S|$

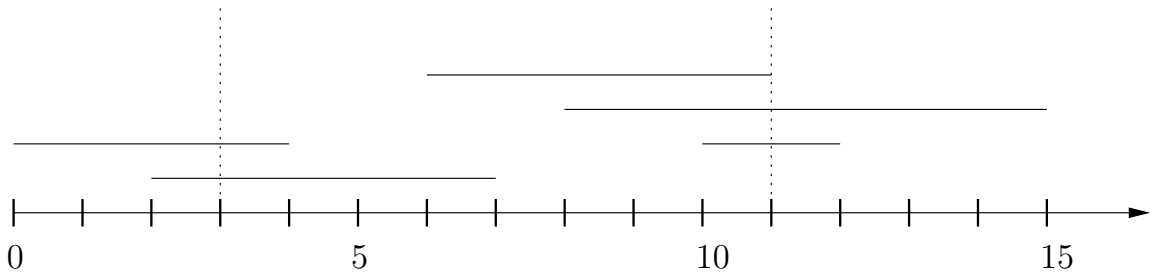

Figure 1: Un exemple avec $n = 5$, et $I_1 = [0, 4]$, $I_2 = [2, 7]$, etc. Ici $S = \{3, 11\}$ est une solution.

1. Ecrivez une fonction récursive AUX **dont vous préciserez les spécifications (liste des paramètres et prérequis, et explication de ce que la fonction calcule)**, ainsi qu'une fonction AUXCLIENT (qui appelle AUX avec les bons paramètres) permettant de calculer la valeur optimale. On ne vous demande pas de transformer AUX et AUXCLIENT en programmation dynamique.

2. Donnez la complexité qu'aurait votre fonction AUXCLIENT si on la transformait en programmation dynamique. Votre formule peut dépendre de n et de $\Delta = max_{i \in [1,n]} f_i$, mais idéalement essayez d'expliquer comment ne dépendre que de n (et avoir ainsi une complexité polynomiale en n).