Algorithmique avancée
Corrigé du TD Programmation Dynamique

J.M. Dischler

Chaînes de caractères

On considère le problème suivant : soit deux chaînes de caractères s et t. On se propose de comparer ces deux chaînes sachant que la première chaîne peut contenir des caractères spéciaux : $\sharp$, $\star$ et ?. Le $\sharp$ remplace un ou plusieurs caractères, le $\star$ zéro ou plusieurs caractères et le ? un caractère. La fonction de comparaison renvoie un booléen, indiquant si la première chaîne peut correspondre à la seconde (on parle de *pattern matching*).

1. Proposer un algorithme récursif. Montrer que la complexité au pire est exponentielle.

```
/* Version recursive, complexite au pire si que des * dans s1, d'ou en O(l1*2^l2)*/
bool jocker(String s1, int p1, String s2, int p2)
{
if (p1==-1 && p2==-1) return true;
if (p1==-1) return false;
if (s1.charAt(p1)=='*')
{
if (p2==-1) return jocker(s1,p1-1, s2, p2);
return jocker(s1,p1-1, s2, p2) || jocker(s1, p1, s2, p2-1);
}
if (s1.charAt(p1)=='#')
{
if (p2==-1) return false;
return jocker(s1,p1-1, s2, p2-1) || jocker(s1, p1, s2, p2-1);
}
if (p2==-1) return false;
if (s1.charAt(p1)=='?') return jocker(s1, p1-1, s2, p2-1);
if (s1.charAt(p1)==s2.charAt(p2)) return jocker(s1, p1-1, s2, p2-1);
return false;
}
```

2. Proposer un algorithme basé sur le principe de la programmation dynamique. Quelle est sa complexité ?

```
/* Version dynamique, complexite en Theta(l1*l2)*/
bool dyna_jocker(String s1, String s2)
{
bool JOCKER[100][100];
int p1,p2;

for (p1=0; p1<=s1.length()+1; p1++) { JOCKER[p1][0]=false; }
for (p2=0; p2<=s2.length()+1; p2++) { JOCKER[0][p2]=false; }
JOCKER[0][0]=true;

for (p1=0; p1<=s1.length(); p1++)
for (p2=-1; p2<=s2.length(); p2++)
{
```

```
if (s1.charAt(p1)=='*')
{
if (p2==-1) JOCKER[p1+1][p2+1]=JOCKER[p1][p2+1];
else JOCKER[p1+1][p2+1]=JOCKER[p1][p2+1]||JOCKER[p1+1][p2];
}
else if (s1.charAt(p1)=='#')
{
if (p2==-1) JOCKER[p1+1][p2+1]=false;
else JOCKER[p1+1][p2+1]=JOCKER[p1][p2] || JOCKER[p1+1][p2];
}
else
{
if (p2==-1) JOCKER[p1+1][p2+1]=false;
else if (s1.charAt(p1)=='?') JOCKER[p1+1][p2+1]=JOCKER[p1][p2];
else if (s1.charAt(p1)==s2.charAt(p2) JOCKER[p1+1][p2+1]=JOCKER[p1][p2];
else JOCKER[p1+1][p2+1]=false;
}
}
```

Programmation dynamique : répartition des espaces en typographie

On considère le problème de la composition équilibrée d'un paragraphe dans un traitement de texte. Le texte d'entrée est une séquence de n mots de longueurs l_1, l_2, ..., l_n mesurées en nombre de caractères. On souhaite composer ce paragraphe de manière équilibrée sur un certain nombre de lignes contenant chacune exactement M caractères. Chaque ligne comportera un certain nombre de mots, les espaces nécessaires à séparer les mots les uns des autres (une espace [1] entre deux mots consécutifs) et des *caractères d'espacement supplémentaires* complétant la ligne pour qu'elle contienne exactement M caractères. La figure 1 présente un exemple de ligne contenant trois mots, deux espaces nécessaires à séparer ces trois mots, et six caractères d'espacement supplémentaires. Si une ligne donnée contient les

U	n		e	x	e	m	p	l	e		t	r	i	v	i	a	l							

FIGURE 1 – Une ligne de 24 caractères contenant 3 mots et 6 caractères d'espacement supplémentaires.

mots de i à j, où $i \leq j$, et étant donné que nous avons besoin d'une unique espace pour séparer deux mots consécutifs, le nombre de caractères d'espacement supplémentaires c nécessaires pour compléter la ligne est égal au nombre de caractères de la ligne (M), moins le nombre de caractères nécessaires pour écrire les mots ($\sum_{k=i}^{j} l_k$), moins le nombre de caractères nécessaires pour séparer les mots ($j - i$), autrement dit : $c = M - \sum_{k=i}^{j} l_k - (j - i)$. Le nombre c de caractères d'espacement supplémentaires doit bien évidemment être positif ou nul.

Notre critère « d'équilibre » est le suivant : on souhaite minimiser la somme, *sur toutes les lignes hormis la dernière*, des cubes des nombres de caractères d'espacement supplémentaires.

Travail minimum requis

Donnez un algorithme de programmation dynamique permettant de composer de manière équilibrée un paragraphe de n mots de longueurs l_1, ..., l_n données, et analysez la complexité de votre algorithme.

Travail idéal

Vous pourrez, *mais ce n'est pas obligatoire*, établir une relation de récurrence définissant la composition optimale, proposer un algorithme récursif implémentant cette récurrence et montrer que sa complexité est médiocre, proposer un algorithme de programmation dynamique et analyser sa complexité, proposer un algorithme par recensement, et finalement proposer un contre-exemple à l'algorithme glouton naïf.

1. En typographie, « espace » est un mot féminin, comme vous le confirmera le premier dictionnaire venu.

Indication pour l'établissement d'une récurrence

Comme l'on cherche ici un algorithme suivant le paradigme de la programmation dynamique, il est raisonnable de définir la composition optimale par une formule de récurrence. Pour ce faire, vous pourrez remarquer que dans un paragraphe de m lignes composé optimalement, les $(m-1)$ *dernières* lignes sont composées optimalement.

Récurrence. *Considérons une solution optimale s'étendant sur m lignes. La composition des $m-1$ dernières lignes est optimale : sinon nous pourrions combiner la composition de la première ligne et d'une composition optimale des $m-1$ dernières pour obtenir une composition strictement meilleure de l'ensemble, ce qui contredirait l'hypothèse d'optimalité.*

Notons $c(i)$ le coût de la composition des mots à partir du i^e (inclus). La première ligne de cette composition commence donc au mot i et finit à un certain mot j (inclus). Nous avons donc :

$$c(i) = \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c(j+1).$$

Pour obtenir la valeur optimale de $c(i)$, nous minimisons la formule précédente sur toutes les valeurs possibles de j, sachant que j vaut au minimum i et que le nombre de caractères supplémentaires doit être positif ou nul :

$$c(i) = \begin{cases} 0 \text{ si } n - i + \sum_{k=i}^{n} l_k \leq M, \\ \min\left\{ \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c(j+1) \,\middle|\, i \leq j \leq n, j - i + \sum_{k=i}^{j} l_k \leq M \right\} \text{ sinon.} \end{cases}$$

Algorithme récursif.

COMPOSITION-RÉCURSIVE(l, i, n, M)
  si $n - i + \sum_{k=i}^{n} l_k \leq M$ **alors renvoyer** 0
  $c \leftarrow +\infty$
  pour $j \leftarrow i$ **à** n **faire**
    si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$
      et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RÉCURSIVE}(l,\, j+1,\, n,\, M) < c\right)$
        alors $c \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RÉCURSIVE}(l,\, j+1,\, n,\, M)$
  renvoyer c

*On pourrait optimiser légèrement cet algorithme en remplaçant la boucle **pour** par une boucle **tant que**.*

Complexité de l'algorithme récursif. *La complexité de cet algorithme est définie par la récurrence :*

$$T(i,n) = 1 + \sum_{j=i+1}^{k} (1 + T(j,n)),$$

où k est le plus grand entier supérieur à i tel que $j - i + \sum_{k=i}^{j} l_k \leq M$. La présence de la valeur de k dans la récursion nous empêche de faire une estimation fine. Nous allons poser l'hypothèse, réaliste, que les lignes sont suffisamment longues et les mots suffisamment courts pour que l'on puisse toujours écrire au moins deux mots par ligne. Sous cette hypothèse, k est toujours supérieur ou égal à $i+2$ (quand $i+2$ est inférieur à n) et :

$$T(i,n) \geq 3 + T(i+1,n) + T(i+2,n)$$

d'où, par substitution de $T(i+1,n)$ par la même minoration,

$$T(i,n) \geq 6 + 2T(i+2,n) + T(i+3,n) \geq 2T(i+2,n).$$

Par conséquent,

$$T(i,n) = \Omega(2^{\frac{n-i}{2}}) \text{ et } T(1,n) = \Omega(2^{\frac{n}{2}}).$$

L'algorithme récursif est donc de complexité au moins exponentielle et il nous faut trouver une solution moins mauvaise.

Solution par programmation dynamique.

COMPOSITION-PROGDYN(l, n, M)

$c[n+1] \leftarrow 0$
pour $i \leftarrow n$ **à** 1 **faire**
  si $n - i + \sum_{k=i}^{n} l_k \leq M$
    alors $c[i] \leftarrow 0$
    sinon
      $c[i] \leftarrow +\infty$
      pour $j \leftarrow i$ **à** n **faire**
        si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$ et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c[j+1] < c[i]\right)$
          alors $c[i] \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + c[j+1]$

Complexité de la solution par programmation dynamique. *La complexité de cet algorithme est immédiate :*

$$T(n) = \sum_{i=n}^{1} \sum_{j=i}^{n} 1 = \sum_{i=n}^{1} (n - i + 1) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

en posant $k = n - i + 1$. *D'où :* $T(n) = \Theta(n^2)$.

Solution par recensement.

COMPOSITION-PARRECENSEMENT(l, n, M)

pour $i \leftarrow 1$ **à** n **faire** $c[i] \leftarrow +\infty$
$c[n+1] \leftarrow 0$
renvoyer COMPOSITION-RECENSEMENT(c, l, 1, n, M)

COMPOSITION-RECENSEMENT(c, l, i, n, M)

si $c[i] < +\infty$ **alors renvoyer** $c[i]$
si $n - i + \sum_{k=i}^{n} l_k \leq M$ **alors renvoyer** 0
pour $j \leftarrow i$ **à** n **faire**
  si $\left(j - i + \sum_{k=i}^{j} l_k \leq M\right)$
    et $\left(\left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RECENSEMENT}(c, l, j+1, n, M) < c[i]\right)$
      alors $c[i] \leftarrow \left(M - j + i - \sum_{k=i}^{j} l_k\right)^3 + \text{COMPOSITION-RECENSEMENT}(c, l, j+1, n, M)$
renvoyer $c[i]$

Contre-exemple à l'algorithme glouton. *La solution naïve est gloutonne : on place le maximum de mots sur la première ligne, puis on appelle récursivement l'algorithme sur les mots restants. Cet algorithme ne fournit pas la solution optimale, comme le montre les figures 2 et 3 qui présente deux compositions différentes du même ensemble de mots sur des lignes de 24 caractères. La figure 2 présente la solution de l'algorithme glouton : la première ligne ne contient aucune espace supplémentaire mais la deuxième en contient sept, d'où un coût de* $0^3 + 7^3 = 343$. *La figure 3 présente une composition optimale : la première ligne contient trois espaces supplémentaires et la deuxième quatre, d'où un coût de* $3^3 + 4^3 = 27 + 64 = 91$, *ce qui montre la non optimalité de l'algorithme glouton.*

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n		n	e	
p	e	u	t		p	a	s		ê	t	r	e		b	o	n								
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t								

FIGURE 2 – Solution de l'algorithme glouton.

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n				
n	e		p	e	u	t		p	a	s		ê	t	r	e		b	o	n					
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t								

FIGURE 3 – Solution optimale.