Algorithmique avancée
Corrigé du devoir en temps libre

Frédéric Vivien

Travail demandé

Ce travail est à effectuer seul ou en binôme, à l'exclusion de tout regroupement de taille supérieure. Vous devez me rendre votre travail pour le lundi 14 janvier, soit directement, soit dans mon casier à la scolarité du département. La date du lundi 14 janvier ne pourra en aucun cas être dépassée : à partir du lendemain, vous pourrez retirer au secrétariat les annales du cours contenant les corrigés de tous les TDs et TPs et de ce devoir.

Vous tâcherez de proposer des solutions les plus simples et les plus esthétiques possibles. Tout vos algorithmes devront être expliqués et justifiés : un algorithme, même parfait, dont la justesse ne serait pas prouvée ne se verra attribuer que la moitié des points prévus par le barème.

À toutes fins utiles, je vous rappelle que les corrigés des TDs et TPs, ainsi que les notes de cours, sont disponibles à l'url : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/`.

Algorithme glouton : l'art de rendre la monnaie

On considère le problème consistant à rendre n centimes (de franc ou d'euro) en monnaie, en utilisant le moins de pièces possible.

1. Décrivez un algorithme glouton permettant de rendre la monnaie en utilisant des pièces de cinquante, vingt, dix, cinq et un centime.
   On note n_{50} le nombre de pièces de 50 centimes à rendre, n_{20} celui de pièces de 20 centimes, etc.

   RENDRE-LA-MONNAIE(n)
   $reste \leftarrow n$
   $n_{50} \leftarrow \lfloor \frac{reste}{50} \rfloor$
   $reste \leftarrow reste - 50 \times n_{50}$
   $n_{20} \leftarrow \lfloor \frac{reste}{20} \rfloor$
   $reste \leftarrow reste - 20 \times n_{20}$
   $n_{10} \leftarrow \lfloor \frac{reste}{10} \rfloor$
   $reste \leftarrow reste - 10 \times n_{10}$
   $n_5 \leftarrow \lfloor \frac{reste}{5} \rfloor$
   $reste \leftarrow reste - 5 \times n_5$
   $n_1 \leftarrow reste$
   renvoyer $(n_{50}, n_{20}, n_{10}, n_5, n_1)$

2. Démontrez que votre algorithme aboutit à une solution optimale.
   On considère une instance de notre problème, c'est-à-dire une somme s à rendre, la solution de notre algorithme glouton pour cette instance, notée $(n_{50}, n_{20}, n_{10}, n_5, n_1)$, et une solution optimale notée $(m_{50}, m_{20}, m_{10}, m_5, m_1)$. On veut montrer que les deux solutions sont identiques.

   ### Première solution

   On raisonne par l'absurde et on suppose qu'elles sont différentes.

On considère les pièces par valeur décroissante. Pour la pièce de plus grande valeur, v, pour laquelle les deux solutions diffèrent, notre solution gloutonne utilise, par construction, plus de pièces que la solution optimale (l'algorithme glouton utilise le maximum de pièces possible à chaque fois, et ici il reste la même somme à rendre pour les deux solutions). Nous avons cinq cas différents à considérer : $v = 1, 5, 10, 20$ et 50 :

(a) $v = 1$. *Ce cas est trivialement impossible. Les deux solutions utilisent le même nombre de pièces de 50 centimes, de 20, de 10 et de 5. Chaque solution utilise alors* $s-50\times n_{50}-20\times n_{20}-10\times n_{10}-5\times n_5$ *pièces de 1 centime et elles ne diffèrent pas. Il y a contradiction.*

(b) $v = 5$. *On a donc* $n_{50} = m_{50}$, $n_{20} = m_{20}$, $n_{10} = m_{10}$ *et* $n_5 > m_5$. *Comme* $n_5 > m_5$ *avec* $5 \times n_5 + n_1 = 5 \times m_5 + m_1$, $m_1 \geq 5$ *et on peut remplacer cinq pièces de 1 centime dans la solution optimale par une pièce de 5 ce qui nous donne une solution strictement meilleure, ce qui contredit l'optimalité.*

(c) $v = 10$. *On a donc* $n_{50} = m_{50}$, $n_{20} = m_{20}$, *et* $n_{10} > m_{10}$. *Comme* $n_{10} > m_{10}$ *avec* $10 \times n_{10} + 5 \times n_5 + n_1 = 10 \times m_{10} + 5 \times m_5 + m_1$, $5 \times m_5 + m_1 \geq 10$. *Alors*

i. *soit* $m_5 \geq 2$ *et on peut remplacer deux pièces de 5 centimes par une de 10 ;*

ii. *soit* $m_5 \leq 1$, *alors* $m_1 \geq 5$ *et on peut remplacer cinq pièces de 1 centime dans la solution optimale par une pièce de 5 ce qui nous donne une solution strictement meilleure, ce qui contredit l'optimalité.*

Dans tous les cas l'optimalité de la solution optimale est contredite.

(d) $v = 20$. *On a donc* $n_{50} = m_{50}$ *et* $n_{20} > m_{20}$. *Comme* $n_{20} > m_{20}$ *avec* $20 \times n_{20} + 10 \times n_{10} + 5 \times n_5 + n_1 = 20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1$, $10 \times m_{10} + 5 \times m_5 + m_1 \geq 20$. *On sait que* $m_{10} \geq 1$, *sinon on se trouverait dans le cas 2c que l'on a montré être impossible. Alors,*

i. *soit* $m_{10} \geq 2$ *et on peut remplacer deux pièces de 10 centimes par une de 20 ;*

ii. *soit* $m_{10} = 1$ *; dans ce cas* $5 \times m_5 + m_1 \geq 10$ *et on se retrouve dans le cas 2c que l'on a montré être impossible.*

Dans tous les cas l'optimalité de la solution optimale est contredite.

(e) $v = 50$. *On a donc* $n_{50} > m_{50}$. *Comme* $n_{50} > m_{50}$ *avec* $50 \times n_{50} + 20 \times n_{20} + 10 \times n_{10} + 5 \times n_5 + n_1 = 50 \times m_{50} + 20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1$, $20 \times m_{20} + 10 \times m_{10} + 5 \times m_5 + m_1 \geq 50$. *On sait que* $m_{20} \geq 1$, *sinon on se trouverait dans le cas 2c que l'on a montré être impossible. Alors,*

i. *soit* $m_{20} = 1$ *et alors* $10 \times m_{10} + 5 \times m_5 + m_1 \geq 30$ *ce qui ne peut être une solution optimale (voir l'étude du cas 2d) ;*

ii. *soit* $m_{20} = 2$ *et alors* $10 \times m_{10} + 5 \times m_5 + m_1 \geq 10$ *avec* $m_{10} \geq 1$ *(sinon* $5 \times m_5 + m_1 \geq 10$ *ce qui ne peut être optimal, comme le montre l'étude du cas 2b) et on peut remplacer les deux pièces de 20 et une de 10 par une pièce de 50 ;*

iii. *soit* $m_{20} \geq 3$ *et on peut remplacer trois pièces de vingt centimes par une pièce de cinquante et une de dix, ce qui contredit l'optimalité...*

Quelles que soient les configurations, on a abouti à une contradiction. Les deux solutions (la gloutonne et l'optimale) sont donc identiques.

Deuxième solution

Il est évident qu'une solution optimale contient au plus quatre pièces de 1 centime (cinq pièces de 1 centime pouvant être remplacées par une de 5 centimes), au plus une pièce de 5, au plus une pièce de 10, et au plus deux pièces de 20 (trois pièces de 20 pouvant être remplacées par une de 50 et une de 10). De plus, une solution optimale ne contient pas en même temps une pièce de 10 centimes et deux de 20. On a donc :

$$m_1 \leq 4, m_5 \leq 1, m_{10} \leq 1, m_{20} \leq 2 \text{ et } m_{10} + m_{20} \leq 2.$$

Par conséquent, dans une solution optimale, les pièces de 1, 5, 10 et 20 centimes représentent un montant d'au plus 49 centimes ($m_1 + 5 \times m_5 + 10 \times m_{10} + 20 \times m_{20} \leq 49$) *et il n'y a pas de lattitude quant au choix du nombre de pièces de 50 centimes :* $m_{50} = \lfloor \frac{n}{50} \rfloor = n_{50}$.
De ce qui précède on déduit que :

(a) *Le montant représenté par les pièces de 1, 5 et 10 centimes est inférieur à 19 centimes, et donc que* $m_{20} = \lfloor \frac{n-50\times n_{50}}{20} \rfloor = n_{20}$.

(b) *Le montant représenté par les pièces de 1 et 5 centimes est inférieur à 9 centimes, et donc que* $m_{10} = \lfloor \frac{n-50\times n_{50}-20\times n_{20}}{10} \rfloor = n_{10}$.

(c) *Le montant représenté par les pièces de 1 centime est inférieur à 4 centimes, et donc que* $m_5 = \lfloor \frac{n-50\times n_{50}-20\times n_{20}-10\times n_{10}}{5} \rfloor = n_5$.

(d) *Comme* $m_{50} = n_{50}$, $m_{20} = n_{20}$, $m_{10} = n_{10}$ *et* $m_5 = n_5$, *on a également* $m_1 = n_1$.

3. On suppose que les pièces disponibles ont pour valeur p^0, p^1, ..., p^k pour deux entiers $p > 1$ et $k \geq 1$ donnés. Montrez que l'algorithme glouton aboutit toujours à une solution optimale sur un tel jeu de pièces.

   On considère une instance de notre problème, c'est-à-dire une somme s *à rendre, la solution de notre algorithme glouton pour cette instance, notée* $(n_k, n_{k-1}, ..., n_1, n_0)$ *(où* n_j *est le nombre d'exemplaires utilisés de la pièce de valeur* p^j*), et une solution optimale notée* $(m_k, m_{k-1}, ..., m_1, m_0)$. *On veut montrer que les deux solutions sont identiques. On raisonne par l'absurde et on suppose qu'elles sont différentes.*

   Par construction de l'algorithme glouton, $n_k \geq m_k$. *L'algorithme glouton utilisant le maximum de pièces possible à chaque fois, il existe un entier* j *appartenant à l'intervalle* $[0, k]$ *tel que pour tout* i, $j+1 \leq i \leq k$, $n_i = m_i$ *et* $n_j > m_j$. $s = \sum_{i=0}^{k} n_i \times p^i = \sum_{i=0}^{k} m_i \times p^i$ *et* $\forall i \in [j+1, k], n_i = m_i$. *Donc,*

$$\sum_{i=0}^{j} n_i \times p^i = \sum_{i=0}^{j} m_i \times p^i \quad \Leftrightarrow \quad \sum_{i=0}^{j-1} (m_i - n_i) \times p^i = (n_j - m_j) \times p^j.$$

   En remarquant que tous les termes n_i *sont positifs, puis que* $n_j > m_j$ *implique* $n_j - m_j \geq 1$ *on obtient :*

$$(n_j - m_j) \times p^j \leq \sum_{i=1}^{j-1} m_i \times p^i \quad \Rightarrow \quad p^j \leq \sum_{i=0}^{j-1} m_i \times p^i.$$

   Montrons alors qu'il existe une pièce (d'une valeur autre que p^k*) utilisée au moins* p *fois dans la solution optimale, ce qui contredit l'optimalité : on peut remplacer* p *exemplaires de cette pièce par une pièce de valeur strictement supérieure. On raisonne par l'absurde et on suppose que toutes les pièces de valeurs strictement inférieures à* p^k *sont prises au plus* $p-1$ *fois :* $\forall i \in [0, k-1], m_i \leq p-1$. *On a alors :*

$$p^j \leq \sum_{i=0}^{j-1} m_i \times p^i \leq \sum_{i=0}^{j-1} (p-1) \times p^i = \sum_{i=0}^{j-1} p^{i+1} - \sum_{i=0}^{j-1} p^i = \sum_{i=1}^{j} p^i - \sum_{i=0}^{j-1} p^i = p^j - 1,$$

   ce qui est absurde.

4. Donnez un ensemble de valeurs de pièces pour lesquelles l'algorithme glouton ne donne pas une solution optimale.

   Si les pièces à notre disposition ont pour valeurs respectives : 7, 5 et 1 centimes, l'algorithme glouton utilisera une pièce de 7 centimes et quatre de 1 pour composer la somme de 11 centimes, quand la solution optimale utilise deux pièces de 5 et une de 1.

Diviser pour régner : intervalle de plus grande somme

Nous avons un tableau A de n entiers relatifs. Nous recherchons un sous-tableau de A dont la somme des éléments soit maximale. Autrement dit, nous recherchons un couple d'entiers i et j, $1 \leq i \leq j \leq n$ tel que $\sum_{k=i}^{j} A[k]$ soit maximale. La figure 1 montre un exemple de tableau pour lequel les valeurs cherchées sont $i = 4$ et $j = 5$.

1. Proposez un algorithme naïf.

2	5	-8	6	5	-9	3	4

FIG. 1 – Tableau dont l'intervalle de plus grande somme est défini par $i = 4$ et $j = 5$.

```
PGS-NAÏF(A)
  d ← 1
  f ← 1
  max ← A[1]
  pour i ← 1 à n faire
  s ← 0
    pour j ← i à n faire
      s ← s + A[j]
      si s > max alors
        d ← i
        f ← j
        max ← s
```

2. Quelle est sa complexité ?

   La première boucle comporte n itérations, pour i allant de 1 à n. La deuxième boucle comporte $n-i+1$ itérations. Toutes les autres opérations sont de coût constant. Le coût global est donc :

$$\sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

   D'où une complexité en $\Theta(n^2)$.

3. Proposez un algorithme « diviser pour régner » découpant le tableau en deux moitiés.

   Trois cas sont possibles : l'intervalle de plus grande somme est inclus dans la première moitié du tableau ; l'intervalle de plus grande somme est inclus dans la deuxième moitié du tableau ; l'intervalle de plus grande somme empiète sur les deux moitiés du tableau. Les deux premiers cas sont réglés par un appel récursif sur chacune des deux moitiés. Pour régler le troisième cas, on calcule l'intervalle de plus grande somme dont la limite droite est le dernier élément de la première moitié et l'intervalle de plus grande somme dont la limite gauche est le premier élément de la deuxième moitié. Ensemble, ces deux intervalles nous fournissent l'intervalle de plus grande somme empiétant sur les deux moitiés. On peut envisager deux méthodes pour réaliser ce calcul.

Première méthode

Ici tout est recalculé à chaque fois, sans tenir compte d'éventuels résultats précédents.

```
PGS(A, d, f)
  si d = f renvoyer (d, d, A[d])
  milieu ← ⌊(f−d)/2⌋
  (début_1, fin_1, somme_1) ← PGS(A, d, milieu)
  (début_2, fin_2, somme_2) ← PGS(A, milieu + 1, f)
  DemiSommeGauche ← A[milieu]
  LimiteGauche ← milieu
  s ← A[milieu]
  pour i ← milieu − 1 à d faire
    s ← s + A[i]
    si s > DemiSommeGauche alorsDemiSommeGauche ← s
                                LimiteGauche ← i
```

$DemiSommeDroite \leftarrow A[milieu + 1]$
$LimiteDroite \leftarrow milieu + 1$
$s \leftarrow A[milieu + 1]$
pour $i \leftarrow milieu + 2$ **à** f **faire**
  $s \leftarrow s + A[i]$
  si $s > DemiSommeDroite$ **alors** $DemiSommeDroite \leftarrow s$
    $LimiteDroite \leftarrow i$
$(début_3, fin_3, somme_3) \leftarrow (LimiteGauche, LimiteDroite, DemiSommeGauche + DemiSommeDroite)$
si $somme_3 \geq somme_2$ et $somme_3 \geq somme_1$
  alors renvoyer ($début_3$, fin_3, $somme_3$)
  sinon si $somme_2 \geq somme_3$ et $somme_2 \geq somme_1$
    alors renvoyer ($début_2$, fin_2, $somme_2$)
    sinon renvoyer ($début_1$, fin_1, $somme_1$)

Deuxième méthode

On évite ici de parcourir entièrement chaque moitié de tableau pour calcule l'intervalle de plus grande somme dont la limite droite est le dernier élément de la première moitié et l'intervalle de plus grande somme dont la limite gauche est le premier élément de la deuxième moitié. Pour ce faire on calcule à chaque fois trois valeurs : l'intervalle de plus grande somme dont la limite droite est le dernier élément du tableau, l'intervalle de plus grande somme dont la limite gauche est le premier élément du tableau, la somme de tous les éléments du tableau.

PGS(A, d, f)
  si $d = f$ **renvoyer** (d, $A[d]$, d, $A[d]$, $A[d]$, d, d, $A[d]$)
  $milieu \leftarrow \lfloor \frac{f-d}{2} \rfloor$
  $(début_1, somme\_début_1, fin_1, somme\_fin_1, somme_1, d_1, f_1, s_1) \leftarrow$ PGS(A, d, $milieu$)
  $(début_2, somme\_début_2, fin_2, somme\_fin_2, somme_2, d_2, f_2, s_2) \leftarrow$ PGS(A, $milieu + 1$, f)
  si $somme\_début_1 \geq somme_1 + somme\_début_2$
    alors
      $somme\_début \leftarrow somme\_début_1$
      $début \leftarrow début_1$
    sinon
      $somme\_début \leftarrow somme_1 + somme\_début_2$
      $début \leftarrow début_2$
  si $somme\_fin_2 \geq somme_2 + somme\_fin_1$
    alors
      $somme\_fin \leftarrow somme\_fin_2$
      $fin \leftarrow fin_2$
    sinon
      $somme\_fin \leftarrow somme_2 + somme\_fin_1$
      $fin \leftarrow fin_1$
  $somme \leftarrow somme_1 + somme_2$
  $d_3 \leftarrow fin_1$
  $f_3 \leftarrow début_2$
  $s_3 \leftarrow somme\_fin_1 + somme\_début_2$
  si $s_3 \geq s_2$ et $s_3 \geq s_1$
    alors renvoyer ($début$, $somme\_début$, fin, $somme\_fin$, $somme$, d_3, f_3, s_3)
    sinon si $s_2 \geq s_3$ et $s_2 \geq s_1$
      alors renvoyer ($début$, $somme\_début$, fin, $somme\_fin$, $somme$, d_2, f_2, s_2)
      sinon renvoyer ($début$, $somme\_début$, fin, $somme\_fin$, $somme$, d_1, f_1, s_1)

L'appel initial est alors de la forme :

$$(début, somme\_début, fin, somme\_fin, somme, i, j, somme\_maximale) = \mathrm{PGS}(A, 1, n).$$

4. Quelle est sa complexité ?

Première méthode

Un appel de PGS *sur un tableau de taille n génère deux appels récursifs sur des tableaux de taille $n/2$. La division a un coût constant (calcul du milieu). Par contre, la combinaison des résultats nécessite la recherche de l'intervalle de plus grande somme se terminant en l'extrémité droite (resp. gauche) du premier (resp. deuxième) demi-tableau. Cette double recherche a un coût en $\Theta(n)$ (parcours de tout le tableau). D'où l'équation définissant la complexité :*

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Nous avons donc ici : $a = 2$, $b = 2$ et $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$. Nous sommes donc dans le cas 2 du théorème et donc :

$$T(n) = \Theta(n \log n).$$

Deuxième méthode

Un appel de PGS *sur un tableau de taille n génère deux appels récursifs sur des tableaux de taille $n/2$. La division a un coût constant (calcul du milieu) tout comme la combinaison des résultats. D'où l'équation définissant la complexité :*

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

Nous avons donc ici : $a = 2$, $b = 2$ et $f(n) = \Theta(1) = O(n^{1-1}) = O(n^{(\log_2 2)-1})$. Nous sommes donc dans le cas 1 du théorème et donc :

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n).$$

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 2 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 mots de i à j, où $i \leq j$, et étant donné

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

FIG. 2 – Une ligne de 24 caractères contenant 3 mots et 6 caractères d'espacement supplémentaires.

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.

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

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.

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] ← 0
pour i ← n à 1 faire
  si n − i + Σ_{k=i}^{n} l_k ≤ M
    alors c[i] ← 0
    sinon
      c[i] ← +∞
      pour j ← i à n faire
        si (j − i + Σ_{k=i}^{j} l_k ≤ M) et ((M − j + i − Σ_{k=i}^{j} l_k)^3 + c[j + 1] < c[i])
          alors c[i] ← (M − j + i − Σ_{k=i}^{j} l_k)^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 ← 1 à n faire c[i] ← +∞
c[n + 1] ← 0
renvoyer COMPOSITION-RECENSEMENT(c, l, 1, n, M)
```

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

```
si c[i] < +∞ alors renvoyer c[i]
si n − i + Σ_{k=i}^{n} l_k ≤ M alors renvoyer 0
pour j ← i à n faire
  si (j − i + Σ_{k=i}^{j} l_k ≤ M)
    et ((M − j + i − Σ_{k=i}^{j} l_k)^3 + COMPOSITION-RECENSEMENT(c, l, j + 1, n, M) < c[i])
      alors c[i] ← (M − j + i − Σ_{k=i}^{j} l_k)^3 + 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 3 et 4 qui présente deux compositions différentes du même ensemble de mots sur des lignes de 24 caractères. La figure 3 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 4 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							

FIG. 3 – 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							

FIG. 4 – Solution optimale.