Chapitre 4

Programmation Dynamique

I. Introduction:

La programmation dynamique, comme la méthode « diviser pour régner », résout les problèmes en combinant les solutions de sous-problèmes. elle s'applique quand les sous-problèmes ne sont pas indépendants mais ont des sous-sous-problèmes en commun.
Dans ce cas, un algorithme « diviser pour régner » fait plus de travail que nécessaire, en résolvant plusieurs fois les sous-sous-problèmes communs.

Un algorithme de programmation dynamique résout chaque sous-sous-problème une unique fois et mémorise sa solution dans un tableau, s'épargnant ainsi le re-calcul de la solution chaque fois que le sous-sous-problème est rencontré.

La programmation dynamique est surtout efficace pour les problèmes d'optimisation : ces problèmes peuvent admettre plusieurs solutions, parmi lesquelles on veut choisir une solution optimale (maximale ou minimale pour une certaine fonction de coût).

Exemple:

Trouver le plus court chemin pour aller d'un point à un autre dans un réseau de transport

La conception d'un algorithme de programmation dynamique peut être planifiée en 4 étapes:

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale partant des cas simples (cas d'arrêt des récursions) et
en remontant progressivement jusqu'à l'énoncé du problème initial.
4. Construire une solution optimale à partir des informations calculées à l'étape précédente.

II. Multiplication d'une suite de Matrices

On suppose que l'on a une suite de n matrices, $A_1 \ldots A_n$, et que l'on souhaite calculer le produit :

$$A_1 \times A_2 \times \ldots\ldots \times A_n$$

On peut évaluer cette expression en utilisant comme sous-programme l'algorithme classique de multiplications de 2 matrices. Une fois qu'on l'a parenthèsée de manière à lever toute ambigüité liée à l'ordre des multiplications de matrices.

- un produit de matrices complètement parenthésé est

soit une matrice unique soit le produit de deux produits de matrice complètement parenthésés).

Exemple: $A_1 \times A_2 \times A_3 \times A_4$

La multiplication de matrices étant associative, le résultat de la multiplication est indépendant du parenthèsage. Il y a ainsi cinq manières différentes de calculer le produit de quatre matrices :

$$\begin{aligned} A_1 A_2 A_3 A_4 &= (A_1(A_2(A_3 A_4))) \\ &= (A_1((A_2 A_3)A_4)) \\ &= ((A_1(A_2 A_3))A_4) \\ &= (((A_1 A_2)A_3)A_4)) \end{aligned}$$

Le parenthèsage du produit peut avoir un impact crucial sur le coût de l'évaluation du produit. Le produit d'une matrice A de taille $p \times q$ par une matrice B de taille $q \times r$ produit une matrice C de taille $p \times r$ en pqr multiplications scalaires.

Considérons trois matrices $A_{1(10\times100)}$; $A_{2(100\times5)}$ et $A_{3(5\times50)}$

Si on effectue la multiplication de ces trois matrices suivant le parenthèsage $((A_1A_2)A_3)$, on effectue

$10 \times 100 \times 5 = 5000$ multiplications dans un premier temps,

puis $10 \times 5 \times 50 = 2500$ dans un

deuxième temps, soit 7500 au total. Si, au contraire, on effectue la multiplication suivant le parenthèsage $(A_1(A_2A_3))$ on effectue:

$100 \times 5 \times 50 = 25000$ multiplications dans un premier temps, puis

$10 \times 100 \times 50 = 50000$ dans un deuxième temps, soit 75000 au total et 10 fois plus qu'avec le premier parenthèsage !

Problématique

Etant donnée une suite $A_1,\ldots\ldots,A_n$ de n matrices, avec $p_{i-1}\times p_i$, la dimension de la matrice A_i .

Il faut trouver le parenthèsage du produit $A_1A_2\ldots\ldots A_n$ qui minimise le nombre de multiplications scalaires à effectuer.

Remarque:

Le nombre de parenthèsages différents est une fonction exponentielle de n donc l'idée d'énumérer tous les parenthèsages est à exclure pour de grandes valeurs de n.

II. 1 Structure d'un Parenthèsage Optimal:

La première étape du paradigme de la programmation dynamique consiste à caractériser la structure d'une solution optimale.

Un parenthèsage optimal de $A_1A_2\ldots\ldots A_n$ ($A_{1\ldots n}$) découpe le produit entre les matrices A_k et A_{k+1} / $1 \leq k \leq n-1$, on calcule d'abord $A_{1\ldots k}$ puis $A_{k+1\ldots n}$ ensuite on multiplie ces produit pour obtenir le produit final $A_{1\ldots n}$.

$\Rightarrow$ le coût de ce parenthèsage $\equiv$ Σ des coûts de calcul de $A_{1\ldots k}$ et $A_{k+1\ldots n}$ et du produit de ces deux matrices

Par conséquent le parenthésage de la sous-suite $A_{1....}A_k$ doit être lui-même un parenthésage optimal de même pour $A_{k+1....}A_n$ par conséquent une solution optimale d'un problème de parenthésage contient elle-même des solutions optimales de sous-problèmes de parenthésage.

Donc La présence de sous-structure optimale à l'intérieur d'une solution optimale est l'une des caractéristiques des problèmes pour lesquels la programmation dynamique est applicable.

II. 2 Résolution récursive:

La deuxième étape du paradigme consiste à définir récursivement la valeur d'une solution optimale en fonction de solutions optimales de sous-problèmes.

Dans notre cas, un sous-problème consiste à déterminer le coût minimum d'un parenthésage de $A_i A_{i+1} \ldots A_j$, pour $1 \leq i \leq j \leq n$.

Soit m[i, j] le nombre minimum de multiplications scalaires nécessaires au calcul de $A_{i \ldots j}$ (le nombre minimum pour $A_{1 \ldots n}$ = m[1, n]).

On peut calculer m[i, j] récursivement de la manière suivante:

- Si $i = j$ alors **$m[i, i] = 0$**
- Sinon ($i < j$) **$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$**

Cette équation nécessite la connaissance de la valeur de k, ce que nous ignorons. Il nous faut donc passer en revue tous les cas possibles et il y en a j-i :

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j. \end{cases}$$

Les valeurs m[i, j] nous donne les coûts des solutions optimales des sous-problèmes.

⇒Pour reconstruire une solution à postériori nous allons stocker dans s[i, j] la valeur de k qui minimise **m[i, j] = m[i, k] + m[k + 1, j] + $p_{i-1}p_k p_j$** .

*Algorithme récursif: (une 1ère solution à notre PBème)

```
CHAÎNEDEMATRICES-RECURSIF(p, i, j)
si i = j alors retourner 0
m[i, j] ←+∞
pour k←i a j − 1 faire
      q ←CHAÎNEDEMATRICES-RECURSIF(p, i, k)+
          CHAÎNEDEMATRICES-RECURSIF(p, k + 1, j) + p_{i−1} p_k p_j
si q < m[i, j] alors m[i, j] ←q
renvoyer m[i, j]
```

La complexité de cet algorithme est donné par la récurrence :

$$T(n) = \begin{cases} 0 & \text{si } n=1 \\ \sum_{k=1}^{n-1} (T(k) + T(n-k) + 2) & \text{Si } n \geq 2 \end{cases}$$

$$\Rightarrow T(n) = \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k) + 2(n-1)$$

$$= [\, T(1)+T(2)+\ldots\ldots+T(n-1)] + [T(n-1)+T(n-2)+\ldots.+T(2)+T(1)] + 2(n-1)$$

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

Par conséquent $T(n) \geq 2T(n-1)$ (T(2) : $T(1)$; T(3) : $2T(1)+1$)

$$\geq 2(T(1) + (2\, T(1) + 1) + (2 (T(1) + (2\, T(1) + 1) + 2)$$

$$+\ldots\ldots\ldots+ 2\,(T(1) + T(2) +\ldots+T(n-2) + (n-2))$$

$$\geq 2^n \Rightarrow \boxed{T(n) = \Omega (2^n)} \Rightarrow$$

Donc la quantité de travail effectuée par l'appel récursif est au moins exponentielle

II. 3 Calcul des coûts optimaux:

L'algorithme récursif rencontre chaque sous-problème un grand nombre de fois (ici, un nombre exponentiel de fois) dans différentes branches de l'arbre des appels récursifs.
Cette propriété, dite des sous-problèmes superposés (des sous-problèmes qui ont des sous-sous-problèmes en commun), est le deuxième indice de l'applicabilité de la programmation dynamique.

Donc plutôt que d'implémenter de manière récursive notre équation de récurrence on aborde la troisième étape du paradigme de la programmation dynamique : on calcule le coût optimal en utilisant une approche ascendante.

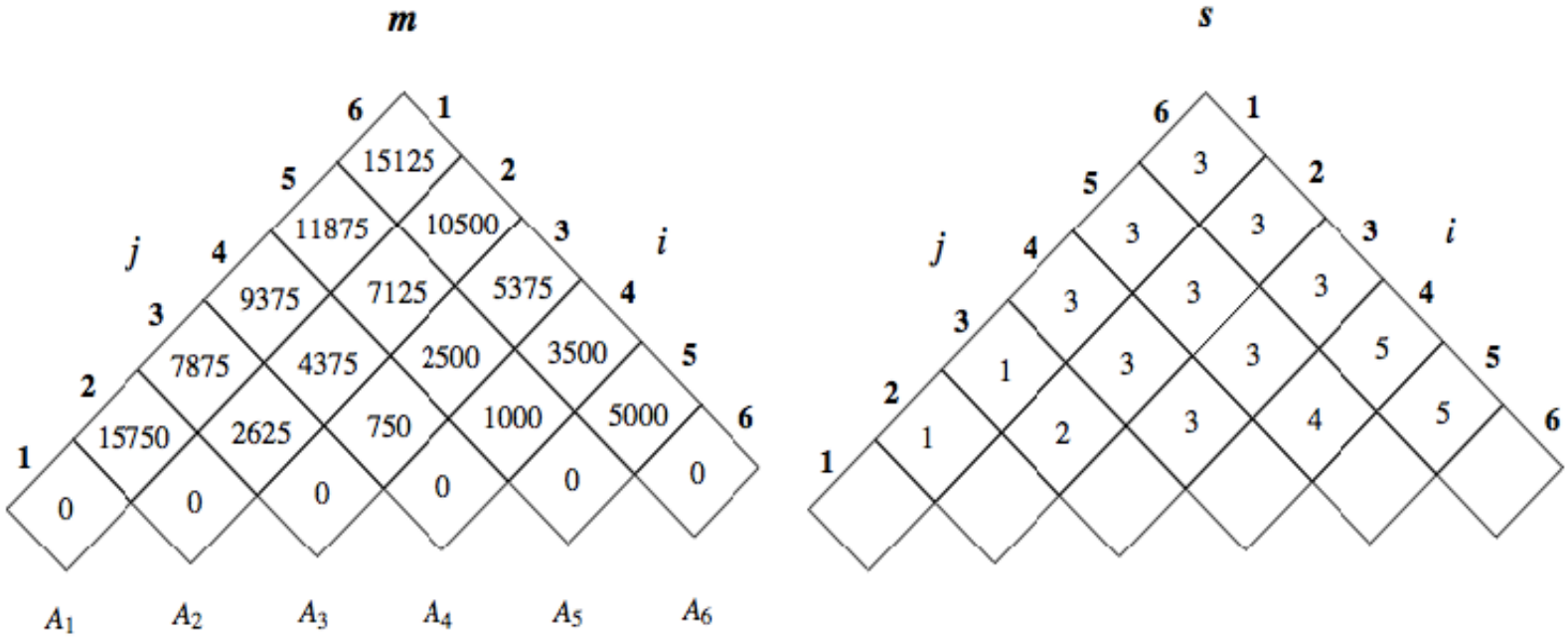
L'entrée de l'algorithme ci-dessous est la séquence $p_0, p_1, \ldots\ldots, p_n$ des dimensions des matrices. Cet algorithme calcule le coût optimal m[i, j] et enregistre un indice s[i, j] permettant de l'obtenir.

```
ORDONNER-CHAÎNEDEMATRICES(p)
Début
   n ← longueur(p) -1
   pour i de 1 à n faire
      m[i, i] ← 0
   finPour
   pour ℓ de 2 à n faire
        pour i de 1 à n − ℓ + 1  faire
                j ← i + ℓ − 1
                m[i, j] ← ∞
                pour k de i à j −1 faire
                   q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                   si q < m[i, j] alors m[i, j] ← q
                                                s[i, j] ← k
                 finPour
        finPour
  finPour
  Retourner m, s
Fin
```

La boucle sur ℓ est une boucle sur la longueur des suites considérées. La figure ci-dessous présente un exemple d'exécution de l'algorithme. Comme m[i, j] n'est défini que pour $i \leq j$, seule la partie du tableau m strictement supérieure à la diagonale principale est utilisée.

Les deux tableaux sont présentés de manière à faire apparaître la diagonale principale de m horizontalement, chaque rangée horizontale contenant les éléments correspondants à des chaînes de matrices de même taille.

L'algorithme calcule les rangées de m du bas vers le haut, et chaque rangée de la gauche vers la droite. Dans notre exemple, un parenthésage optimal coûte 15 125 multiplications scalaires.



Les tableaux m et s sont calculés par ORDONNER-CHAÎNEDEMATRICES pour n = 6 et les dimensions :30; 35; 15; 5; 10; 20; 25.

Complexité

Un simple coup d'œil à l'algorithme montre que sa complexité est en $O(n^3)$. Plus précisément :

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

$$= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} 2(\ell-1)$$

$$= \sum_{l=2}^{n} (n-\ell+1)(\ell-1)$$

$$= \frac{n(n-1)(n+1)}{6} \in O(n^3)$$

La complexité de l'algorithme ORDONNER-CHAÎNEDEMATRICES est donc en $O(n^3)$ ce qui est infiniment meilleur que la solution naïve énumérant tous les parenthésages ou que la solution récursive, toutes deux de complexité exponentielle.

II. 4 Construction d'une solution optimale:

L'algorithme ORDONNER-CHAÎNEDEMATRICES calcule le coût d'un parenthésage optimal, mais n'effectue pas la multiplication de la suite de matrices.

Par contre, l'information nécessaire à la réalisation d'un calcul selon un parenthésage optimal est stockée au fur et à mesure dans le tableau s : s[i; j] contenant une valeur k pour laquelle une séparation du produit $A_iA_{i+1}.....A_j$ entre A_k et A_{k+1} fourni un parenthésage optimal. L'algorithme MULTIPLIER-CHAÎNEDEMATRICES ci-dessous réalise la multiplication et résout donc notre problème.

```
MULTIPLIER-CHAÎNEDEMATRICES(A, s, i, j)
si j > i alors
        X ←MULTIPLIER-CHAÎNEDEMATRICES(A, s, i, s[i, j])
        Y ←MULTIPLIER-CHAÎNEDEMATRICES(A, s, s[i, j] + 1, j)
        renvoyer MULTIPLIER-MATRICES(X, Y)
sinon renvoyer A_i
```

L'algorithme MULTIPLIER-CHAÎNEDEMATRICES (A, s, 1, 6) calcule le produit de la suite de matrices en suivant le parenthésage :

$$((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$$

car s[1, 6] = 3; s[1, 3] = 1 et s[4, 6] = 5.

III. Eléments de la programmation Dynamique

On examine ici les deux caractéristiques principales que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable : une sous-structure optimale et des sous-problèmes superposés.

Sous-structure Optimale

Un problème fait apparaître une sous-structure optimale si une solution optimale au problème fait apparaître des solutions optimales aux sous-problèmes.

La présence d'une sous-structure optimale est un bon indice de l'utilité de la programmation dynamique.

Sous-problèmes superposés :

La seconde caractéristique que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable est « l'étroitesse »de l'espace des sous-problèmes, au sens où un algorithme récursif doit résoudre constamment les mêmes sous-problèmes, plutôt que d'en engendrer toujours de nouveaux.

En général, le nombre de sous-problèmes distincts est polynomial par rapport à la taille de l'entrée.

Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d'optimisation contient des sous-problèmes superposés. A contrario, un problème pour lequel l'approche « diviser pour régner » est plus adaptée génère le plus souvent des problèmes nouveaux à chaque étape de la récursivité.

Les algorithmes de programmation dynamique tirent parti de la superposition des sous-problèmes en résolvant chaque sous-problème une unique fois, puis en conservant la solution dans un tableau où on pourra la retrouver au besoin avec un temps de recherche constant.

Problème

On définit sur les mots trois opérations élémentaires :

- la substitution : on remplace une lettre par une autre,
- l'insertion : on ajoute une nouvelle lettre,
- la suppression : on supprime une lettre.

Par exemple, sur le mot *'carie'*, si on substitue c en d, a en u et si on insère t après le i, on obtient *'durite'*.

La distance d'édition entre deux mots U et V est le nombre minimal d'opérations pour passer de U à V. Ainsi, la distance de *'carie'* à *'durite'* est 3 : deux substitutions et une insertion. La distance de *'aluminuim'* à *'albumine'* est 4 : une insertion, b, une substitution, i en e et deux suppressions, u et m.

Posons

- $\delta_{i,j} = 0$ si la i-ème lettre de U et la j-ème lettre de V sont identiques, sinon 1 ;
- si $U = u_1u_2u_3 \dots u_n$ alors on appelle suffixe $U_i = u_1u_2u_3 \dots u_i$;
- si $V = u_1u_2u_3 \dots u_m$ alors on appelle suffixe $V_j = u_1u_2u_3 \dots u_j$;
- par convention $U_0 = V_0 = 0$, le mot vide.

Une formule de récurrence pour calculer la distance d'édition est

a. $d(\varphi, V_j) = j$ (j insertions) (la distance pour passer du mot vide φ au suffixe V_j) ;
b. $d(U_i, \varphi) = i$ (i suppressions) (la distance pour passer du suffixe U_i au mot vide φ) ;
c. $d(U_i, V_j) = \min(d(U_{i-1}, V_{j-1}) + \delta_{i,j}; d(U_{i-1}, V_j) + 1; d(U_i, V\,j-1) + 1)$ pour $i \in \{1, 2, \dots, n\}, j \in \{1, 2, \dots, m\}$.

1. Justifier cette formule de récurrence.

   Indication : pour (c.), on écrit U_i et V_j sous la forme $U_i = U_{i-1}\alpha$ et $V - j = V_{j-1}\beta$ et on considère les différentes manières de transformer U_i en V_j (il y a trois manières).

La récurrence est construite sur les longueurs des deux mots :
En se basant sur les deux cas de base a. et b. on peut dire que la distance entre deux mots de longueurs respective i et j peut être déduite à partir de la distance entre les deux préfixes des deux mots de longueurs respectives i-1 et j-1 (en supposant qu'on a ajouté un caractère à la fin de chacun des deux préfixes).
On peut envisager 3 possibilités:

a. On retient la distance entre U_{i-1} et V_{j-1} et nous supposons que le $i^{ème}$ caractère de U est substitué par le $j^{ème}$ caractère de V (au cas où les deux caractères sont différents).
b. On retient la distance entre U_{i-1} et V_j et nous supposons qu'un caractère a été supprimé à la fin de U
c. Inversement, on retient la distance entre U_i et V_{j-1}et nous supposons qu'un caractère a été ajouté à la fin de U.

2. Écrire une fonction récursive `Distance` qui calcule la distance d'édition de deux chaînes de caractères de tailles respectives n et m. Testez-la sur *'carie'* et *'durite'*, *'aluminui'* et *'albumine'*.

```
Fonction  Distance( U, V : MOT; i, j : entier ) : entier
Début

Si ( i = 0 )  alors    retourner ( j )
    sinon si ( j = 0) alors retourner ( i )
            sinon
                  retourner ( min ( distance( U, V, i -1, j-1) + diff_car(U[i],V[j]),
                                    distance( U, V, i -1, j) +1,
                                    distance( U, V, i , j-1) + 1 ))
            fsi
     fsi
fsi

Fin
```

```
Fonction diff_car(a, b : caractère) : entier
Début
Si ( a= b) alors  retourner ( 0 )
Sinon  retourner ( 1 )
Fsi
Fin
```

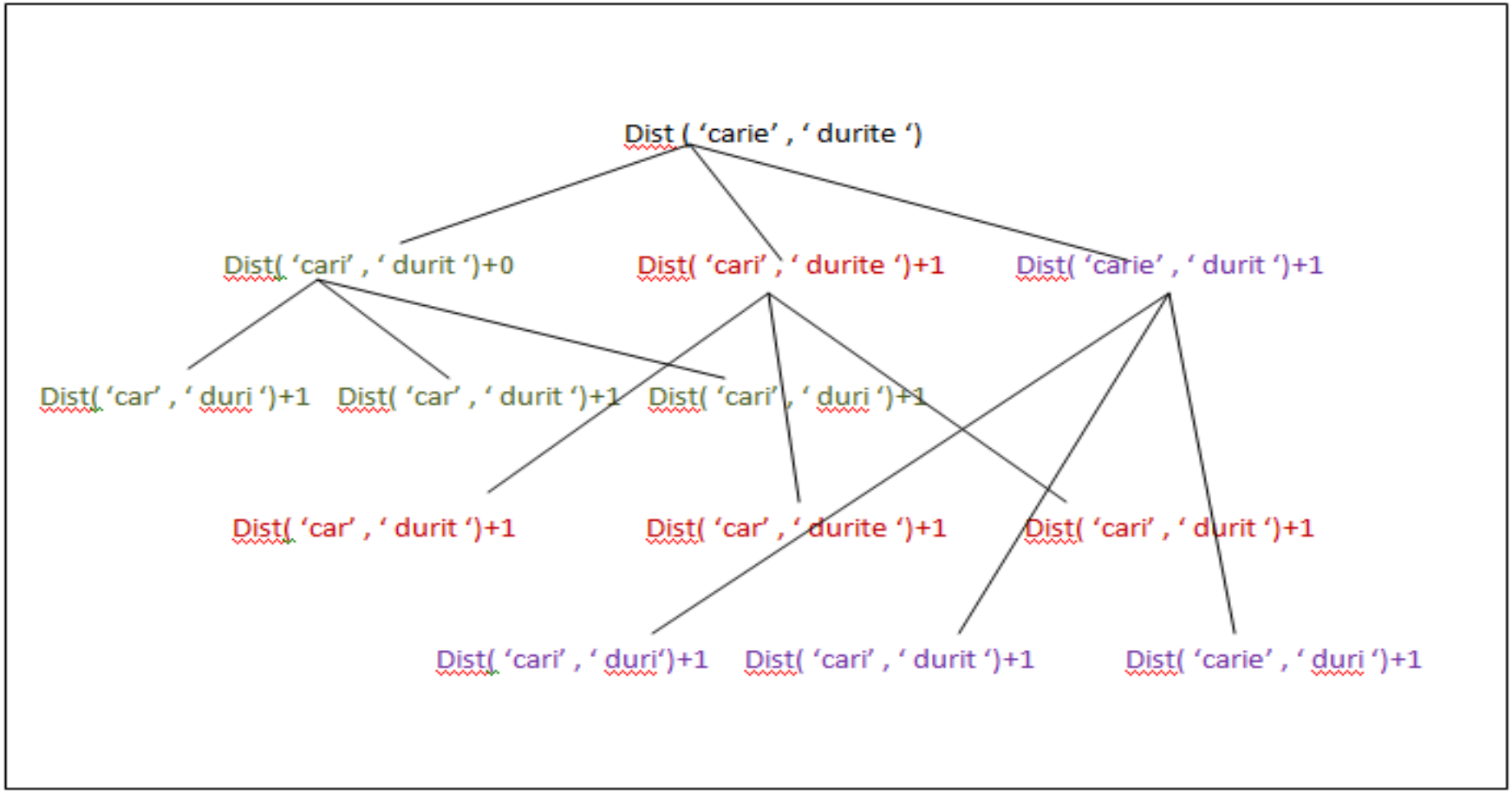
3. Calculer la complexité de votre algorithme Distance.

Indication : On pourra raisonner sur deux mots de mêmes longueurs n. Remarquez que dans cet algorithme les mêmes calculs sont faits plusieurs fois

$$T(n) = 3\,T(n-1) + O(1)$$
$$\geq 3\,T(n-1)$$
$$\geq 3\,(3\,T(n-2))$$
$$\geq 3\,(3\,(3\,T(n-3))) \geq \ldots\ldots\ldots \geq 3^{n-1}\,T(1)$$

$$\Rightarrow\ T(n) = \Omega(3^n)$$

Le problème de la fonction `Distance` est qu'elle effectue de nombreux appels récursifs redondants. Cela conduit à une complexité exponentielle. La solution est de stocker les appels récursifs dans une table D de dimension 2, telle que $D(i, j)$ soit la distance de U_i à V_j. On ajoute une colonne et une ligne pour traiter le mot vide φ. Cela donne finalement une table indexée par 0 à n et 0 à m. Le résultat est $D(n, m)$. Par exemple, en supposant que les chaînes de caractères sont indexées à partir de 1, la table pour *'carie'* et *'durite'* est



Dans l’arbre des appels récursifs on rencontre plusieurs fois les mêmes sous-problèmes (ss-pbs superposés) ce qui génère un algo récursif ayant un temps d’exécution en exponentielle

	φ	d	u	r	i	t	e
φ	0	1	2	3	4	5	6
c	1	1	2	3	4	5	6
a	2	2	2	3	4	5	6
r	3	3	3	2	3	4	5
i	4	4	4	3	2	3	4
c	5	5	5	4	3	3	3

4. Écrire une fonction `Distance_dynamique` qui calcule la distance de deux chaînes de caractères sans appels récursifs, en construisant la table D tout en justifiant l'utilisation du paradigme de la programmation dynamique.

```
Procédure  Distance_dyn( U, V : MOT; n, m : entier , var D :tableau[0…n,0…m] d'entier )
Début
pour i de 0 à n faire
      D[i, 0] ← i
finPour
pour j de 0 à m faire
      D[0, j] ← j
finPour

pour i de 1 à n faire
    pour  j  de 1 à m faire
       D[i, j] ←  min ( D[i −1, j−1] + diff_car(U[i],V[j]),  D[i −1, j] +1, D[i , j −1] +1 )
     finPour
 finPour


Fin
```

5. Calculer la complexité de votre algorithme Distance_dynamique.

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{m} cst$$

$$= O(n \times m)$$

Si $n = m$ alors $T(n) = O(n^2)$

7. (Facultative) Écrire une procédure itérative qui prend arguments deux chaînes de caractères et affiche la suite d'opérations sous la forme décrite ci-dessus.
Pour cela, vous devez utiliser la table D et construire l'historique des transformations en remontant dans la table à partir de la case D(n;m) jusqu'à la case D(0; 0).

```
Procédure  Affiche_Transf( U, V : Mot; n, m : entier; D :tableau[0…n,0…m] d'entier )
Début

                i ← n
                j ← m

Tantque ( i > 0  et  j  > 0)  Faire

Selon D [ i, j]  Faire
 D [ i-1, j] + 1  :  écrire ( U[i], ' ', '-')
                                  i - 1
  D [ i, j-1] + 1  :  écrire ( '-', ' ', V[j])
                                  j - 1
Sinon
  si  (diff_car( U[i] , V[j]) = 1)  alors écrire ( U[i], '  ', V[j])
   sinon
                   écrire ( U[i], '|', V[j])
   fsi

                i ← i - 1
                j ← j - 1
FinSelon
FinTanque
Fin
```

C	D
A	U
R |	R
I |	I
-	T
E |	E

A |	A
L |	L
-	B
U |	U
M |	M
I |	I
N |	N
I	E
U	-
M	-

	φ	d	u	r	i	t	e
φ	0	1	2	3	4	5	6
c	1	1	2	3	4	5	6
a	2	2	2	3	4	5	6
r	3	3	3	2	3	4	5
i	4	4	4	3	2	3	4
c	5	5	5	4	3	3	3