# TP3 : recursivite terminale, programmation dynamique

## Exercice 1 : Exponentiation rapide

On se propose de définir une fonction exponentielle, $x^y$, sans utiliser la fonction prédéfinie de PASCAL. Pour cela, deux formules de récurrence sont possibles :

$$(A) \quad \begin{cases} \exp(x,0) &= 1 \\ \exp(x,y+1) &= \exp(x,y) * x \end{cases}$$

$$(B) \quad \begin{cases} \exp(x,0) &= 1 \\ \exp(x,2y) &= \exp(x*x,y)(y>0) \\ \exp(x,2y+1) &= \exp(x*x,y) * x \end{cases}$$

**Q 1.1** Implémenter la fonction exponentielle pour chacune des deux récurrences $(A)$ et $(B)$ (et vérifiez que les deux formules donnent bien le même résultat). D'après vous, laquelle des deux récurrences est la plus efficace ?

**Q 1.2** Modifier les deux fonctions de manière à afficher le nombre d'appels récursifs. Avec Gnuplot tracer le nombre d'appels récursifs pour $(A)$ et $(B)$ (choisir une valeur de $x$ appropriée, la complexité de l'algorithme n'en dépend pas, et faire varier $y$). Estimer la complexité des deux algorithmes.

**Q 1.3** Ecrire maintenant une fonction itérative qui calcule l'exponentielle de deux entiers en utilisant la décomposition $(B)$.

## Exercice 2 : Alignement de deux mots

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 subsitutions et une insertion. La distance de aluminium à albumine est 4: une insertion, b, une substitution, i en e et deux suppressions, u et m.

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

$$\begin{aligned} d(\varepsilon, v) &= |v| \text{(on fait } |v| \text{ insertions)} \\ d(u, \varepsilon) &= |u| \text{ (on fait } |u| \text{ suppressions)} \\ d(ua, va) &= d(u, v) \\ d(ua, vb) &= 1 + min(d(u,v), d(ua,v), d(u,vb)) \\ & \quad \text{(la dernière opération est une substitution de la lettre } a \text{ en } b, \\ & \quad \text{ou une insertion de la lettre } b, \text{ ou une suppression de la lettre } a) \end{aligned}$$

où $a$ et $b$ sont deux lettres quelconques distinctes, $\varepsilon$ est le mot vide et où $|u|$ représente la longueur du mot $u$.

**Q 2.1** Écrire une fonction récursive

```
function distance_recursive (u, v :  STRING) : CARDINAL;
```

qui calcule la distance d'édition de deux chaînes de caractères[1]. Testez-la sur carie et durite, aluminium et albumine, puis sur un couple de mots un peu plus longs.

[1] La fonction PASCAL `copy(s :  STRING; index , count :  INTEGER) : STRING;` pourra être utile.

Le problème de la fonction `distance_recursive` 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(U'first..i)` à `V(V'first..j)`. On ajoute une colonne et une ligne pour traiter le mot vide. Cela donne finalement une table indexée par `U'first-1..U'last` et `V'first-1..V'last`. Le résultat est `D(U'last, V'last)`. Par exemple, en supposant que les chaînes de caractères sont indexées à partir de 1, la table pour carie et durite est

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

**Q 2.2** Version programmation dynamique Écrire une fonction

```
function distance_dynamique (u, v :  STRING) : CARDINAL;
```

qui calcule la distance de deux chaînes de caractères sans appels récursifs, en construisant la table `D`.

**Note.** Pour créer la table dans laquelle on va stocker les valeurs intermédiaires, on pourra utiliser les instructions suivantes qui permettent de définir la taille d'un tableau.

Si le tableau est déclaré ainsi :

```
var
    table : array of array  of CARDINAL;
```

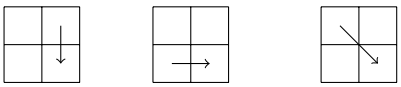
et que `u` et `v` sont deux chaînes de caractères de longueur respective $n$ et $m$, alors l'obtention d'un tableau de taille $n+1 \times m+1$ indicé de 0 à $n$ et de 0 à $m$ se fait ainsi :

```
setlength(table,length(u)+1);
for i := 0 to length(u) do setlength(table[i],length(v)+1);
```

On veut maintenant connaître la suite d'opérations qui mène de `U` à `V`. Pour cela, on peut visualiser les transformations par un petit schéma:

```
c a r i - e          a l - u m i n i u m
      | |   |          |   |   | | | |
d u r i t e          a l b u m i n e - -
```

Deux lettres identiques sont signalées par |. Un espace dans le mot de départ correspond à une insertion, un espace dans le mot d'arrivée correspond à une suppression, et une substitution est représentée par les deux lettres face à face, sans |. Il est possible de connaître la dernière opération appliquée en regardant comment la valeur de `table(length(u),length(v))` a été obtenue. Par exemple, si `table(length(u),length(v))` est égal à `table(length(u),length(v)-1)`+1, alors la dernière opération est une insertion de `V(length(v))`. Ci-dessous est représenté schématiquement d'où provient le résultat et l'opération correspondante :



insertion suppression substitution ou identité

**Q 2.3** Retrouvez à la main à partir de la table de l'exemple la suite des transformations pour passer de carie à durite.

**Q 2.4** Écrire une procédure :

```
procedure afficher (u, v :  STRING);
```

aui affiche la suite d'opérations sous la forme décrite ci-dessus. Pour cela, vous devez utiliser la table et construire l'historique des transformations en remontant dans la table à partir de `(length(u),length(v))`.