CHAPITRE III

Programmation Dynamique

III.1 Exemple introductif : la suite de Fibonacci

La suite de Fibonacci est la suite d'entier $(u_n)_{n\geq 0}$ définie récursivement par :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \forall n \geq 2 \end{cases}$$

On peut traduire directement cette définition en un algorithme récursif :

Algorithm 1 Fibonacci

```
1: procedure FIBONACCI(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return Fibonacci(n − 1) + Fibonacci(n − 2)
6:     end if
7: end procedure
```

Pour analyser la complexité de cet algorithme, on remarque que chaque appel à Fibonacci() se fait en temps constant (si on ne tient pas compte des appels récursifs lancés). Il suffit donc de compter le nombre d'appels, pour n en entrée, que l'on va noter A_n. La suite A_n vérifie :

$$\begin{cases} a_0 = 1 \\ a_1 = 1 \\ a_n = 1 + a_{n-1} + a_{n-2} \quad \forall n \geq 2 \end{cases}$$

Cette suite ressemble à la suite de Fibonacci. On peut l'étudier mathématiquement et on trouve que $A_n \in \Theta(\phi^n)$, où $\phi = \frac{1+\sqrt{5}}{2} \approx 1,6$ est le nombre d'or. La complexité de l'algorithme est donc exponentielle.

D'où vient une telle complexité ? Si on développe le calcul de u_6, on a :

$$u_6 = u_5 + u_4 = u_4 + u_3 + u_3 + u_2 = u_3 + u_2 + u_2 + u_1 + u_2 + u_1 + u_1 + u_0 = \cdots$$

On remarque que les mêmes calculs sont effectués un nombre important de fois.

Pour améliorer la performance de l'algorithme, on peut stocker les résultats intermédiaires obtenus et les réutiliser directement quand on en a besoin au lieu de les recalculer. C'est l'idée de la programmation dynamique. Cela donne :

Algorithm 2 Fibonacci

```
1: procedure FIBONACCI(n,T[ ])
2:     if n ≤ 1 then
3:         return n
4:     end if
5:     if T[n] n'est pas défini then
6:         T[n] = Fibonacci(n − 1) + Fibonacci(n − 2)
7:     end if
8:     return T[n]
9: end procedure
```

La complexité devient alors linéaire : on ne remplit la case $T[n]$ qu'une fois, donc le nombre d'appels à Fibonacci(i), pour chaque valeur $i \leq n$ est d'au plus 3, une fois pour le calculer, une fois quand on calcule Fibonacci($i + 1$) et une fois quand on calcule Fibonacci($i + 2$).

On peut dé-récursiver l'algorithme. Il s'agit de remplir le tableau T directement case par case. On s'aperçoit qu'il faut faire le contraire de ce que fait l'algorithme récursif : commencer par les petites valeurs.

Algorithm 3 Fibonacci

```
1: procedure FIBONACCI(n)
2:     Allouer T avec n + 1 cases
3:     T[0] = 0
4:     T[1] = 1
5:     for i de 2 à n − 1 do
6:         T[i] = T[i − 1] + T[i − 2]
7:     end for
8:     return T[n]
9: end procedure
```

Il est immédiat que l'algorithme est bien linéaire. Il prend aussi une quantité de mémoire linéaire.

La dernière étape consiste à essayer de gagner en mémoire en ne mémorisant que le nécessaire. Pour notre exemple, il suffit de se rappeler des deux derniers termes pour calculer le nouveau. On peut donc n'utiliser qu'un espace mémoire constant (Voir l'algo 4).

III.2 Principe général

On part d'un problème dont on peut trouver une solution optimale à partir des solutions optimales de sous-problèmes du même type. Quand ses sous-problèmes se recouvrent, c'est-à-dire que leurs résolutions ont des calculs en commun, on stocke en mémoire les calculs déjà effectués afin de ne pas avoir à les refaire dans le traitement d'un autre sous-problème.

Pour résoudre un problème en utilisant la programmation dynamique il faut :

Algorithm 4 Fibonacci

```
1: procedure FIBONACCI(n)
2:     Allouer T avec n + 1 cases
3:     a = 0
4:     b = 1
5:     for i de 2 à n − 1 do
6:         c = b
7:         b = a + b
8:         a = b
9:     end for
10:    return b
11: end procedure
```

1. Trouver un découpage en sous-problèmes plus petits, de sorte que les sous-problèmes se recouvrent.
2. Mémoriser les calculs déjà effectués pour ne pas lancer plusieurs fois le même appel.
3. Essayer de dé-récursiver l'algorithme, pour obtenir un algorithme itératif. Il faut en général commencer par les plus petits objets pour reconstruire des objets de plus en plus grands.
4. Essayer de gagner de l'espace mémoire en "jetant" ce qui ne sert plus, ou plutôt en réutilisant l'espace mémoire qui ne sert plus.

Dans le cadre du cours on vous demande surtout de faire 1. et 2., parfois 3.

III.3 Exemple : le sac à dos avec répétitions

On dispose de n types d'objets différents. Le i-ème objet x_i a un poids $\texttt{poids}[i]$ et un prix $\texttt{prix}[i]$. L'endroit contient plein d'exemplaires de chaque type d'objet. On dispose également d'un sac de capacité C, le poids maximal qui peut être dedans.

Le problème est d'optimiser la valeur totale du sac : on met autant d'exemplaire de chaque objet que l'on veut, tant qu'on ne dépasse pas la capacité, et on veut que la somme des prix soit maximale. On considère que toutes les valeurs sont des entiers.

La solution s'obtient en faisant le découpage suivant : tout objet x_i de poids inférieur ou égale à C est placé dans le sac, il reste donc à traiter le sous-problème avec $C - \texttt{poids}[i]$, ce qu'on fait récursivement, en ajoutant le prix de x_i pour avoir la valeur optimale d'un sac contenant x_i. Une fois qu'on a calculé la valeur optimale d'un sac contenant x_1, la valeur optimale d'un sac contenant x_2, ... on prend la plus grande de toute ces valeurs. Mathématiquement, cela donne, en notant $\texttt{Sac}(C)$ la valeur optimale pour une capacité C :

$$Sac(C) = \begin{cases} 0 & \text{si tous les poids sont supérieurs àC,} \\ \max_{i|\texttt{poids}[i]\leq C} \left(Sac(C - \texttt{poids}[i]) + \texttt{prix}[i]\right) & \text{sinon.} \end{cases}$$

On peut traduire directement l'expression mathématique en algorithme récursif, et utiliser la mémorisation sur les valeurs de la capacité pour faire de la programmation dynamique. La complexité du résultat est en $\mathcal{O}(Cn)$.

Quand on dérécursive on obtient l'algorithme ci-dessous.

Algorithm 5 Sac à dos

```
1: procedure SAC(C)
2:     Allouer S avec C + 1 cases
3:     S[0] = 0
4:     for p de 1 à C do
5:         m = 0
6:         for i de 0 à n − 1 do
7:             if poids[i] ≤ p then
8:                 if S[p − poids[i]] + prix[i] ≥ m then
9:                     m = S[p − poids[i]] + prix[i]
10:                end if
11:            end if
12:        end for
13:        T[p] = m
14:    end for
15:    return T[C]
16: end procedure
```

III.4 Exemple : Distance d'édition

Nous développons à présent un exemple classique d'utilisation de la programmation dynamique : le calcul de la *distance d'édition*.

Soient u et v deux mots sur un alphabet A. La distance d'édition entre u et v est le nombre minimum d'opérations à effectuer pour transformer u en v. Les opérations autorisées sont :

- Insertion : on ajoute une lettre de A dans u, où on veut.
- Suppression : on enlève une lettre de u.
- Substitution : on change une lettre de u en une autre lettre (éventuellement la même, ce qui ne change alors pas le mot).

Exemple : $u = verites$ et $v = eviter$. On peut faire la suite minimale de transformations suivante :

$$verites \mapsto erites \mapsto evites \mapsto eviter$$

Comme il n'y a pas de transformation en moins d'étapes (c'est affirmé, non prouvé, mais on peut tester toutes les possibilités), la distance d'édition entre u et v est 3.

Aligments : on représente les opérations effectuées pour passer d'un mot à l'autre par un *alignement*. Voilà un alignement entre *chien* et *niche* :

$$w = \begin{pmatrix} c & h & i & e & n & - & - \\ - & n & i & - & c & h & e \end{pmatrix}$$

Si on regarde colonne par colonne un alignement, on voit qu'on a une suite de transformations qui permettent de passer du mot en haut au mot en bas. Une colonne de la forme $\binom{\alpha}{-}$ correspond à une suppression de α. Une colonne de la forme $\binom{-}{\beta}$ correspond à une insertion de β. Une colonne de la forme $\binom{\alpha}{\beta}$ correspond à une substitution de α en β. Le coût pour passer du premier mot au deuxième en utilisant ces transformations est exactement le nombre de colonnes où les deux lettres (en haut et en bas sont différentes). Sur l'exemple, il y a 7 colonnes, dont une $\binom{i}{i}$ avec deux fois la même lettre, pour un coût total de 6.

Un *alignment optimal* entre u et v est un alignement qui réalise la distance d'édition, c'est-à-dire qui minimise le coût parmi tous les alignements et qui a donc pour coût $d(u, v)$.

Remarque importante : Soit $w = \binom{x\alpha}{y\beta}$ un alignement optimal de dernière lettre $\binom{\alpha}{\beta}$, alors $\binom{x}{y}$ est également un alignement optimal. Cela donne une piste pour la programmation dynamique, puisqu'on peut se ramener à des alignements de mots plus petits.

On distingue trois cas pour un alignement de ua avec vb :

- Si la dernière colonne est $\binom{\alpha}{-}$, cela signifie que la dernière opération effectuée est une suppression. Donc nécessairement on a supprimé la dernière lettre de ua, c'est-à-dire a : on a donc $\alpha = a$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre u et vb, qui est donc de poids $d(u, vb)$. Le coût total est donc $d(u, vb) + 1$.
- Si la dernière colonne est $\binom{-}{\beta}$, cela signifie que la dernière opération effectuée est une insertion. Donc nécessairement on a inséré la dernière lettre de vb, c'est-à-dire b : on a donc $\beta = b$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre ua et v, qui est donc de poids $d(ua, v)$. Le coût total est donc $d(ua, v) + 1$.
- Si la dernière colonne est $\binom{\alpha}{\beta}$, avec α et β qui ne sont pas des $-$, cela signifie que la dernière opération effectuée est soit une substitution, soit rien. Le "rien" n'arrive que quand $\alpha = \beta$. Ce qui reste après, si l'alignement était optimal, c'est un alignement optimal entre u et v, qui est donc de poids $d(u, v)$. Le coût total est donc $d(u, v) + \delta_{a,b}$, où $\delta_{a,b}$ vaut 1 si $\alpha \neq \beta$ et 0 sinon.

On en déduit la formule récursive pour $d(ua, vb)$:

$$d(ua, vb) = \min\{d(ua, v) + 1, d(u, vb) + 1, d(u, v) + \delta_{a,b}\}.$$

Il nous manque les conditions aux bords, mais on remarque que $d(\varepsilon, v) = |v|$ (en insérant les lettres de v une par une) et $d(u, \varepsilon) = |u|$ (en supprimant les lettres de u une par une).

Pour les algorithmes ci-dessous, les indices des lettres d'un mot u de taille n vont de 0 à $n-1$ et $u[0, i]$ est le préfixe $u_0u_1 \cdots u_i$ de u de longueur $i+1$ (ou le mot vide si i est négatif). On utilise la notation $u[i]$ pour désigner la lettre u_i. Le mot u est de taille n et le mot v de taille m.

Algorithm 6 Distance d'édition

```
1: procedure D(u,v)
2:     if u = ε then
3:         return m
4:     end if
5:     if v = ε then
6:         return n
7:     end if
8:     if u[n − 1] == v[m − 1] then
9:         δ = 0
10:    else
11:        δ = 1
12:    end if
13:    return min(d(u, v[0, m − 2]) + 1, d(u[0, n − 2], v) + 1, d(u[0, n − 2], v[0, m − 2]) + p)
14: end procedure
```

On retombe donc sur un algorithme récursif avec des sous-problèmes qui se recouvrent, on va stocker dans un tableau les calculs intermédiaires, c'est à dire les distance d'édition de tous les préfixes de u et v. On note $T[][]$ ce tableau ; on stocke dans $T[i][j]$ la distance d'édition entre le préfixe de longueur i de u et celui de longueur j de v. Plutôt que de faire des appels récursifs, on va remplir T dans l'ordre de la taille des préfixes.

Algorithm 7 Distance d'édition itératif

```
1: procedure D(u,v)
2:     for i de 0 à n do
3:         T[i][0] = i
4:     end for
5:     for j de 0 à m do
6:         T[0][j] = j
7:     end for
8:     for i de 1 à n do
9:         for j de 1 à m do
10:            if u[i − 1] == v[j − 1] then
11:                δ = 0
12:            else
13:                δ = 1
14:            end if
15:            T[i][j] = min(T[i][j − 1] + 1, T[i − 1][j] + 1, T[i − 1][j − 1] + δ)
16:        end for
17:    end for
18:    return T[n][m]
19: end procedure
```

La complexité est en $\mathcal{O}(n \times m)$ en temps et en espace. On remarque qu'on peut faire le calcul en ne gardant en mémoire que deux lignes ou deux colonnes (puisqu'on ne regarde que dans la colonne d'avant et la ligne d'avant), ce qui permet de ne stocker que $\mathcal{O}(n)$ valeurs.

Exemple : On prend $u = aaba$ et $v = bab$:

	i	0	1	2	3	4
j			a	a	b	a
0		0	1	2	3	4
1	b	1	1	2	2	3
2	a	2	1	1	2	2
3	b	3	2	2	1	2