

Chapitre 5

Programmation Dynamique

5.1 Présentation générale

Le terme « programmation dynamique » recouvre un ensemble de techniques qui permettent de résoudre les problèmes en combinant les solutions de sous-problèmes, un peu à la manière de l'approche « diviser pour régner ». Cependant, une grande différence sépare ces deux approches :

- Dans le cas de « diviser pour régner », tous les sous-problèmes sont indépendants. Lorsque les sous problèmes ne sont pas indépendant, l'utilisation d'une approche récursive mène souvent à des algorithmes très peu efficace d'une complexité souvent exponentielle.
- La « programmation dynamique » est, elle, applicable surtout les sous-problèmes ne sont pas indépendants, c'est-à-dire même s'il existe des sous-sous-problèmes en commun.

La programmation dynamique aborde ce problème de sous-problèmes non indépendants de la manière suivante:

- l'approche résout les sous-problèmes qu'une seule fois, même si ces sous-problèmes arrivent plusieurs fois;
- cela n'est possible que parce qu'elle mémorise les solutions de tous les sous-problèmes qu'elle rencontre.

On utilise surtout la programmation dynamique pour des problèmes d'optimisation. Souvent, il y a plusieurs solutions optimales, et dans ce cas-là, suivant les implémentations de la programmation dynamique, on peut extraire l'une d'entre elle, ou l'ensemble des solutions optimales.

Exemple inutile : La suite des nombres de Fibonacci est définie comme suit:

$$\begin{cases} f(0) &= 1 \\ f(1) &= 1 \\ f(n+1) &= f(n) + f(n-1) \quad \forall n > 1 \end{cases}$$

La définition même de la suite est récursive. Il s'ensuit qu'il est très facile d'écrire un programme récursif pour le calcul du $(n+1)^{\text{ième}}$ élément de la suite :

- appel récursif pour le calcul de $f(n)$
- appel récursif pour calcul de $f(n-1)$
- calcul de $f(n+1) = f(n) + f(n-1)$

Voilà le pseudo-code de cet algorithme:

```

1  int fibo(n) {
2      si n=0 ou n=1 alors 1

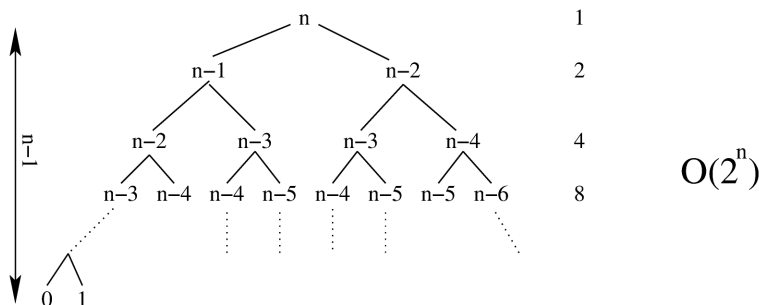
```

```

3     sinon retourner fibo(n-1) + fibo(n-2);
4 }

```

Calcul de la complexité. Intéressons nous à la complexité de cette approche très simple à implémenter dans un langage récursif. La figure suivante développe l'arbre des appels récursifs à la fonction écrite:



A chaque nœud, l'algorithme doit faire une addition. De ce fait la complexité est proportionnelle au nombre de nœuds de l'arbre des appels récursifs.

Si on suit la branche de gauche, il y a n étapes. Si on suit la branche la plus à droite, il y a $n/2$ étapes. Donc si on coupe l'arbre à la profondeur $n/2$, on a un arbre binaire complet (chaque nœud à une distance de la racine strictement inférieure à $n/2$ a exactement deux fils, et les nœuds à une distance $n/2$ sont des feuilles). Au niveau 1, il y a 1 nœud; au niveau 2, il y a 2 nœuds; au niveau 3, il y a 2^2 nœuds; ... et au niveau $n/2$, il y a $2^{n/2-1}$ nœuds. Globalement, il y a donc

$$1 + 2 + \dots + 2^{n/2-1} = \frac{2^{n/2} - 1}{2 - 1} = 2^{n/2} - 1$$

nœuds. La complexité est donc au moins en $\Omega(2^{n/2})$, borne inférieure asymptotique. Si maintenant on considère l'arbre binaire complet de profondeur n , le même raisonnement mène à une complexité en $O(2^n)$.

La complexité est donc exponentielle! Cet algorithme ne peut pas être utilisé pour calculer la valeur de la suite de Fibonacci pour des grandes valeurs de l'argument. Cependant, si cet algorithme est si coûteux en temps, c'est qu'il passe son temps à recalculer des choses qu'il a déjà calculé. Il y a énormément de sous-problèmes en commun: par exemple, on calcule 3 fois $f(n-3)$...

Pour avoir une chance d'améliorer la complexité, il faut essayer de mémoriser les valeurs que l'on doit calculer, afin de ne avoir à les calculer, si on rencontre plus tard le même sous problème. C'est l'essence même de la programmation dynamique.

L'algorithme ci-dessous se décompose en deux fonctions: la première permet d'initialiser la structure de données qui va mémoriser les solutions intermédiaires (ici, un tableau), et la seconde fait les deux appels récursif à la fonction. Cependant, notez bien qu'avant de lancer le calcul récursif, cette deuxième fonction teste si le calcul n'aurait pas déjà été fait.

```

1  fibo1(n)
2      - initialisation du tableau : tous les éléments à 0
3      - Tableau[1] = 1
4      - Tableau[0] = 1
5      - retourner fibo2(n, tableau);
6

```

```

7  fibo2(n,tableau)
8      - si tableau[n] != 0 alors retourner tableau[n]
9      - sinon
10         r1 = fibo2(n-1, tableau)
11         r2 = fibo2(n-2, tableau)
12         r  = r1 + r2
13         tableau(n) = r
14         retourner(r);

```

Complexité :

- La phase d’initialisation est en $\Theta(n)$
- les appels récursifs sont en nombre proportionnel à n , car il n’est fait que si le calcul n’a jamais été fait avant.
- Ainsi la complexité totale est en $\Theta(n)$

Notons tout de même que l’on peut écrire une version itérative de l’algorithme :

```

1  fiboIteratif(n)
2      a = 1;
3      b = 1;
4      Pour i = 2 à n faire
5          c = a + b;
6          a = b;
7          b = c;
8      Fin Pour
9      Retourner (b);

```

Complexité de l’algorithme itératif : $O(n)$

Pour développer un algorithme de programmation dynamique, il y a 4 étapes :

1. caractériser la famille de problèmes à laquelle appartient le problème initial,
2. définir les relations entre les solutions aux problèmes de cette famille,
3. construire un algorithme récursif basé sur ces relations,
4. s’il y a des sous-problèmes en commun, construire un algorithme de programmation dynamique en stockant les résultats des sous-problèmes pour ne pas avoir à les recalculer.

5.2 Exemple paradigmatique : le problème du sac à dos

Il s’agit de maximiser le **bénéfice** pouvant être obtenu en remplissant un sac à dos avec des objets. A chaque objet est associé un bénéfice ($\in \mathbb{R}$) et un poids ($\in \mathbb{N}$). On ne doit pas dépasser un poids total maximal donné au départ.

Si on pouvait fractionner les objets, on pourrait commencer par les objets qui rapportent le plus (par unité de poids) puis s’il reste de la place, continuer avec les objets qui rapportent le plus parmi ceux qui restent... Mais cette approche ne conduit pas à une solution optimale pour un problème non fractionnable.

Soit W le poids maximal pouvant être contenu par le sac. Soit n le nombre total d'objets. On va construire un algorithme de programmation dynamique pour répondre au problème initial. Voici les quatre étapes :

1. Définir la famille de problèmes à laquelle appartient le problème initial :
 $B(k, w)$ va représenter le bénéfice optimal obtenu en remplissant le sac en utilisant uniquement les items $1, 2, \dots, k$ sans jamais dépasser un poids w .
2. Il faut maintenant construire les relations qui peuvent exister entre les différents sous-problèmes, autrement dit les relations qui permettent de décomposer un problème en sous-problèmes de tailles plus petites. Mais il ne faut pas oublier les cas de base pour lesquels une solution est évidente.
 - Si le poids total autorisé est $w = 0$, alors, on ne peut mettre aucun objet dans le sac à doc, ainsi, $B(k, 0) = 0$.
 - Pour résoudre $B(k, w)$, si $poids[k] > w$, on est ramené au cas $B(k - 1, w)$, car il n'est pas possible de choisir l'objet k .
 - Si $poids[k] \leq w$, on a 2 possibilités :
 - inclure l'item k : $bénéfice[k] + B(k - 1, w - poids[k])$
 - ne pas prendre l'item k : $B(k - 1, w)$
 - Et on retourne le maximum des 2 choix précédents.
3. A partir des relations précédentes, il est facile de décrire un algorithme récursif qui réponde au problème.

```

1  BeneficeSac(W : int , n : int, poids : tableau d'int, benef :  tableau d'int)
2      Retourner BS(n,W,poids,benef)
3
4  BS(k : int, w : int , poids : tableau d'int, benef :  tableau d'int)
5      si w=0 alors retourner 0;
6      si poids[k]>w alors
7          retourner BS(k-1,w,poids,benef)
8      sinon
9          b1 = BS(k-1, w-poids[k], poids, benef) + benef[k];
10         b2 = BS(k-1, w, poids, benef);
11         retourner (max(b1,b2));

```

On peut faire tourner le résultat sur l'exemple suivant:

objet	0	1	2	3	4	5
poids	5	2	4	6	3	1
benef	14	6	13	17	10	4

Si on pose $W = 12$, le bénéfice optimal est 37. 3 solutions mènent à cet optimal :

- les objets 6, 5, 4 et 2
- les objets 1, 2, 3 et 6
- les objets 1, 3, 5

Malheureusement, cette version récursive calcule plusieurs fois la solution de sous-problèmes identiques.

4. Il faut donc envisager un stockage des résultats intermédiaires pour ne pas avoir à les recalculer. Comme tout à l'heure pour les nombres de Fibonacci, il est possible de construire une version récursive dans laquelle le tableau des résultats intermédiaires est mis à jour à

la demande. Mais ici, on va préférer une version itérative qui remplit systématiquement ce tableau des résultats intermédiaires.

```
1  BeneficeSac(W: int , n: int, poids: tableau d'int, benef:  tableau)
2      de i=0 à n-1          /* les objets sont numérotés de 0 à n-1 */
3          B[0,i] = 0
4      de w = 1 à W
5          B[w,0] = (w>poids[0])? benef[0] : 0
6          De i = 1 à n-1
7              si poids[i]>w alors
8                  B[w,i] = B[w,i-1]
9              sinon
10                 B[w,i] = max(B[w,i-1], B[w-poids[i],i-1] + benef[i])
11      renvoyer B[W,n-1]
```

Habituellement, pour avoir la (ou les) liste(s) qui mène(ent) à l'optimal, on part de la case du tableau $B[W, n-1]$, et à chaque étape on cherche quelle opération a permis d'obtenir cette valeur. Cette phase de remontée est assez classique, on la retrouvera lors de l'étude des algorithmes d'alignements de séquences.

Suivant les langages de programmation utilisés, il est cependant possible de mémoriser lors de la phase de descente, les ensembles d'objets qui mènent au nombre que l'on inscrit dans le tableau. Lorsque tout le tableau est construit, on a donc directement les listes d'objets qui mènent à l'optimal.