1 Limite à l'utilisation des algorithmes récursifs

Les algorithmes récursifs offrent souvent une solution élégante à des problèmes variés. Cependant, nous avons vu qu'il fallait tenir compte de certains aspects relatifs à leurs complexité.

1.1 Rappels

La complexité spatiale de tels algorithmes est en général un premier frein à leur utilisation (croissance notamment de la pile d'appels, notamment). Il arrive qu'on puisse se dégager de cette contrainte en rendant l'algorithme récursif terminal, au prix d'une conception plus pénible.

La complexité temporelle est liée au rapport entre le nombre d'appels récursifs et la taille des paramètres de tels appels.

— Avec un appel récursif, on a en général des résultats corrects :
si la complexité $C(n)$ vérifie une relation de récurrence $C(n) = C(n-1) + O(n^\alpha)$, alors
$$C(n) = O(n^{\alpha+1})\,;$$
si la complexité $C(n)$ vérifie une relation de récurrence $C(n) = C(n/2) + O(n^\alpha)$, alors
$$C(n) = O(n^\alpha) \text{ si } \alpha > 0,$$
$$C(n) = O(\log n) \text{ si } \alpha = 0\,;$$

— Avec deux appels récursifs, ça se complique :
si la complexité $C(n)$ vérifie une relation de récurrence $C(n) = 2C(n/2) + O(n^\alpha)$, alors
$$C(n) = O(n^\alpha) \text{ si } \alpha > 1,$$
$$C(n) = O(n \log n) \text{ si } \alpha = 1,$$
$$C(n) = O(n) \text{ si } \alpha < 1\,;$$
mais une relation du type $C(n) = C(n-a)+C(n-b)+O(n^\alpha)$ conduit invariablement à une explosion combinatoire :
$$C(n) \sim K.r^n$$
où r est un réel strictement supérieur à 1

Or on rencontre souvent des situations de ce genre, quand il faut appeler récursivement la fonction selon des cas différents et choisir ensuite le cas selon les résultats des appels, on n'a alors pas d'autre stratégie que de devoir tout tester.

1.2 Des mauvais exemples

Par exemple, on a vu un exemple de rendu de monnaie dans un devoir : la fonction suivante donne un couple $(u, v) \in \mathbb{N}^2$ tel que $n = au + bv$ si c'est possible, $(-1, -1)$ sinon.

```
let rec rendu a b n =
  if (n mod b = 0) then (0, n/b)
  else if n < a then (-1,-1)
  else
    let (u,v) = rendu a b (n-a) in
    if u = -1 then (-1,-1) else (u+1,v);;
```

Mais si on ajoute la condition (farfelue !) que la solution retournée doit maximiser le produit uv, alors il faut calculer toutes les solutions :

```
let rec rendu2 a b n =
  if n = 0 then (0,0)
  else if n < a then (-1,-1)
  else
    let (u,v) = rendu2 a b (n-a) in
    let (w,x) = rendu2 a b (n-b) in
    match u, w with
    | -1,-1 -> (-1,-1)
    | -1, _ -> (w,x+1)
    | _, -1 -> (u+1,v)
    | _, _ -> if u*v < w*x then (w,x+1)
      else (u+1,v);;
```

Avec $a = 3, b = 5, n = 100$, la fonction rendu2 doit calculer rendu2 3 5 97 puis rendu2 3 5 95, ce qui provoque le calcul de rendu2 3 5 94, rendu2 3 5 92, puis le calcul de rendu2 3 5 92, rendu2 3 5 90, etc.

On voit tout de suite là où se situe une source de gain de temps : éviter de réitérer le même appel récursif...

1.3 Mémoïsation

Pour gagner du temps, on stocke les valeurs calculées et avant tout appel récursif, on vérifie si on a déjà calculé la valeur voulue : si oui, on la récupère directement dans la mémoire. On échange de la complexité temporelle avec de la complexité spatiale.

Ce principe permet d'éviter plusieurs appels récursifs identiques : on l'appelle la *mémoïsation* ou *mémoïzation*.

2 Principes de la programmation dynamique

Dans de nombreuses situations, on cherche une situation **optimale** : le plus court chemin dans un graphe (de longueur minimale), le parcours élémentaire qui rapporte le plus de points (score maximal), la transformation d'un objet en un autre en un nombre minimal d'étapes intermédiaires, etc.

La programmation dynamique est une méthode de décomposition d'un problème d'optimisation en un certain nombre de sous-problèmes telle que la solution soit obtenue « facilement » dès que l'on connait celles de tous ces sous-problèmes. La difficulté ici est de déterminer les bons sous-problèmes pour ne pas devoir les traiter tous, ce qui est sans doute coûteux (voir exemple précédent ou algorithme du « compte est bon »), et d'éviter de traiter plusieurs fois les mêmes sous-problèmes.

Remarque. L'idée générale est voisine de la méthode « diviser pour régner » : celle-ci peut s'appliquer en toute généralité, mais est finalement assez peu subtile. Elle explore entièrement l'arbre des appels récursifs, car la division se fait en sous-problèmes disjoints, ce qui est en général faux dans le cadre de la programmation dynamique.

Certains problèmes d'optimisation se traitent bien par programmation dynamique...

2.1 Principe de sous-optimalité de Bellman

Un problème d'optimisation est agréablement résoluble par programmation dynamique dans le cas suivant :

> la solution du problème d'optimisation peut être obtenue par combinaisons de solutions optimales sur les sous-problèmes.

Le principe de sous-optimalité de Bellmann est satisfait lorsque toute partie d'une solution optimale à un problème est elle-même une solution optimale à un sous-problème.

Par exemple, si A, B, C, D sont quatre points tels que le plus court chemin entre A et D passe par B et C, alors la partie du chemin entre B et C est elle-même le plus court chemin entre B et C (évident : preuve par l'absurde).

La programmation dynamique permet, quand on peut l'appliquer selon ce principe, de réduire considérablement la complexité en évitant la recherche de solutions de sous-problèmes qui ne servent à rien.

Les idées générales sont donc les suivantes :
- décomposition du problème en sous-problèmes ;
- mise en mémoire des solutions déjà calculées pour réutilisation éventuelle.

2.2 Quelques exemples

a) Histoire du chamelier

Le temps a effacé les parenthèses dans un calcul algébrique : est-ce que $1+2\times3\times4+5$ doit être lu $(1+2)\times(3\times4+5)$? ou $(1+2\times3)\times4+5$?

Tout cela dépend de ce qu'on veut obtenir, par exemple le plus grand nombre possible ou le plus petit.

Si on cherche à maximiser le nombre, alors la solution est $(1+2)\times(3\times(4+5))$. En revanche, si on veut minimiser, la solution est $1+(((2\times3)\times4)+5)$, équivalente à la solution sans parenthèses.

Disons qu'on cherche à maximiser le nombre.

On indice les caractères de l'expression : $e = 1_0 +_1 2_2 \times_3 3_4 \times_5 4_6 +_7 5_8$. Pour $(i,j) \in \{0,\dots,9\}$ tel que $i<j$, on note $t(i,j)$ la sous-expression comprenant les caractères d'indice k tel que $i \leqslant k \leqslant j$. Par exemple, $t(2,4)$ désigne l'expression $2_2 \times_3 3_4$.

La valeur maximale d'une sous-expression $t(i,j)$ est notée $M(i,j)$. L'objectif est donc de calculer $M(0,8)$.

Pour les sous-expressions simples formées d'un entier unique, on a directement leur valeur maximale : si i est pair, alors $M(i,i)$ est connue :

$$M(0,0)=1,\ M(2,2)=2,\ M(4,4)=3,\ M(6,6)=4,\ M(8,8)=5$$

Puis on note que pour tout couple (i,j) tel que $i<j$, i et j pairs,

$$M(i,j) = \max_{\substack{i\leqslant k\leqslant j-1\\ k \text{ impair}}} \big(M(i,k)\ e_k\ M(k+1,j)\big)$$

(le principe de sous-optimalité de Bellmann s'applique ici).

On reconnaît ici une définition récursive de la fonction M.

Si on l'applique directement, on constate qu'on va appeler plusieurs fois la fonction M avec les mêmes paramètres, donc on augmente inutilement la complexité. Pour éviter cela, on va travailler du bas vers le haut plutôt que récursivement du haut vers le bas.

On commence par ranger les nombres connus dans un tableau rectangulaire, ils sont sur la diagonale inverse :

j						
	8					5
	6				4	+
	4			3	×	
	2		2	×		
	0	1	+			
		0	2	4	6	8
				i		

Puis on calcule les nombres de la sur-diagonale de la précédente :

j						
	8				9	5
	6			12	4	+
	4		6	3	×	
	2	3	2	×		
	0	1	+			
		0	2	4	6	8
				i		

Et on recommence progressivement selon chaque sur-diagonale :

j						
	8			27	9	5
	6		24	12	4	+
	4	9	6	3	×	
	2	3	2	×		
	0	1	+			
		0	2	4	6	8
				i		

j						
	8		54	17	9	5
	6	36	24	12	4	+
	4	9	6	3	×	
	2	3	2	×		
	0	1	+			
		0	2	4	6	8
				i		

j						
	8	81	54	17	9	5
	6	36	24	12	4	+
	4	9	6	3	×	
	2	3	2	×		
	0	1	+			
		0	2	4	6	8
				i		

La valeur maximale est donc 81. Pour la calculer, on s'est servi des valeurs calculées précédemment et stockées dans le tableau : on a donc échangé de la complexité temporelle en complexité spatiale.

Ce principe permet d'éviter plusieurs appels récursifs identiques : on l'appelle la *mémoïsation* ou *mémoïzation*.

b) Parcours monotone dans un quadrillage

On considère un quadrillage à n lignes et p colonnes, avec dans chaque case un entier naturel. Un robot se trouve dans la case en haut à droite et doit descendre dans la case en bas à gauche en ramassant sur son chemin le plus grand nombre de points. Posé comme cela, le problème est trivial : on lui fait parcourir toutes les cases. Mais il est plus intéressant si on impose que les déplacements de case en case du robot ne peuvent être que horizontal vers la gauche $\leftarrow$ ou vertical vers le bas $\downarrow$.

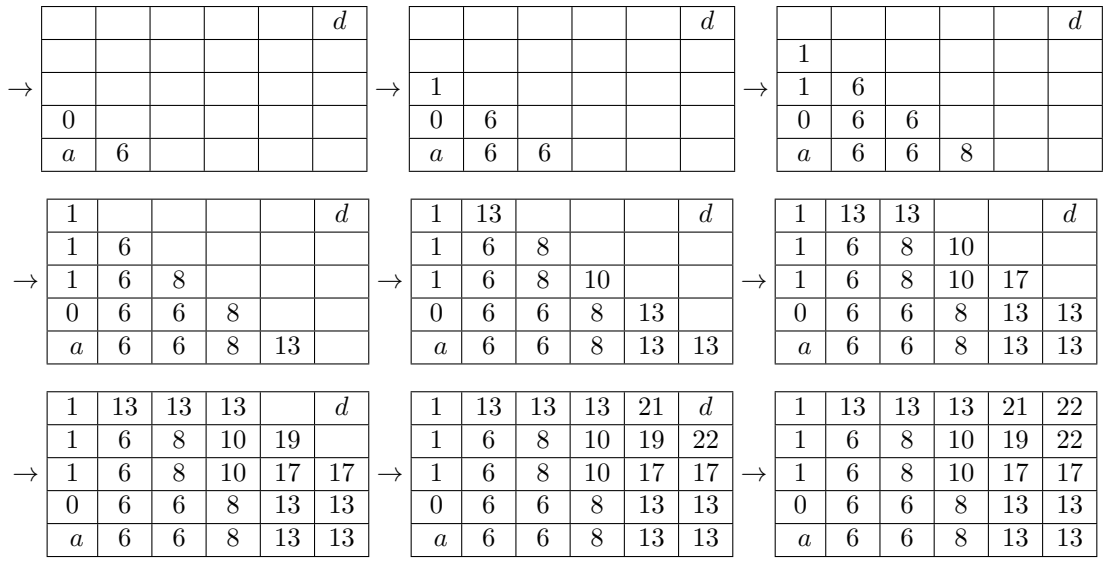
	7			2	d
				2	3
1		2	2	4	
a	6		2	5	

On note $v(x, y)$ la valeur de la case de coordonnées (x, y) et $f(x, y)$ la valeur maximale d'un parcours depuis la case (x, y) vers la case $(0, 0)$. L'objectif est de calculer la valeur de $f(p-1, n-1)$.

Là encore, le principe de sous-optimalité de Bellmann s'applique. On obtient la relation récursive suivante :

$$f(x,y) = \begin{cases} v(x,y) + \max(f(x-1,y), f(x,y-1)) & \text{si } x \neq 0 \text{ et } y \neq 0 \\ v(x,0) + f(x-1,0) & \text{si } x \neq 0 \text{ et } y = 0 \\ v(0,y) + f(0,y-1) & \text{si } x = 0 \text{ et } y \neq 0 \\ v(0,0) & \text{si } x = 0 \text{ et } y = 0 \end{cases}$$

On peut encore calculer $f(x, y)$ en allant du bas vers le haut. On peut remplir un tableau rectangulaire de proche en proche.



De plus, si on regarde bien le tableau, on peut retrouver un chemin qui réalise cette valeur maximale : en partant de la case de départ, on va de proche en proche dans la case accessible maximale.

3 En pratique

Si on reconnaît un problème résoluble par programmation dynamique, on a deux méthodes de résolution.

La première est itérative. C'est en fait ce qui a été réalisé dans les exemples précédents.

- On explicite la formulation récursive du problème ;
- on choisit une structure de données modifiable adaptée au problème (en général un tableau), destinée à recevoir les résultats intermédiaires : c'est une mémoire ;
- on remplit la mémoire selon la formulation précédente, en commençant par les cas de base, puis de proche en proche ;
- la solution est en général dans la dernière case.

La seconde est récursive.

- On explicite d'abord la formulation récursive du problème ;
- on choisit une structure de données modifiable adaptée au problème (en général un tableau), destinée à recevoir les résultats intermédiaires : c'est une mémoire ;
- la fonction récursive est encapsulée dans une fonction d'environnement, qui crée la structure de donnée précédente avant la définition de la fonction récursive ;
- dans la fonction récursive, **avant** de lancer un appel récursif, on vérifie d'abord si la valeur qu'on veut calculer ne l'a pas été avant : elle a été stockée dans la mémoire ; si oui, alors on se contente de lire le contenu de la mémoire ; si non, alors on lance le calcul récursif et avant de retourner la valeur calculée, on la « mémoïse » ;
- la fonction d'environnement lance l'appel initial à la fonction récursive et se contente d'en donner le résultat.

La forme récursive est en général plus simple à mettre en ouvre, car on se contente de transposer la formulation récursive dans la fonction. La forme itérative nécessite en général de savoir organiser la remontée des calculs dans le bon ordre et calcule parfois un peu plus de valeurs que nécessaire.

3.1 Exemples

a) Problème du chamelier

On se donne un tableau `nbr` de n nombres et une chaîne de caractères `sgn` de $n-1$ signes.

```
let applique s a b =
 match s with
 | '+' -> a + b
 | '*' -> a * b;;

let maxi nbr sgn =
 let n = vect_length nbr in
 let res = make_matrix n n 0 in
 let rec val x y =
   if res.(y).(x) != 0 then res.(y).(x)
   else begin
     let r =
     if x = y then nbr.(x)
     else
       let m =ref 0 in
       for k = (x) to (y-1) do
         let s = applique sgn.[k] (val x k) (val (k+1) y) in
         if s > !m then m := s
       done;
       !m
     in
     res.(y).(x) <- r;
     r
   end
 in
 val 0 (n-1);;

let nbr = [|1;2;3;4;5|] and sgn ="+**+" in maxi nbr sgn;;
```

b) Robot

t désigne la matrice représentant le quadrillage initial (par simplicité, il est carré), le point de départ est en $(n-1, n-1)$, le point d'arrivée est en $(0, 0)$.

```
let robot t =
  let n = vect_length t in
  let c = make_matrix n n (-1) in
  let rec f x y =
    if c.(x).(y) <> -1 then c.(x).(y)
    else begin
      let r =
        (if x = n-1 && y = n-1 then 0
        else
          (if x = n-1 then f x (y+1)
          else if y = n-1 then f (x+1) y
          else max (f (x+1) y) (f x (y+1))
          )
        ) + t.(x).(y)
        in
        c.(x).(y) <- r;
        r
  end
  in
  f 0 0;;
```

4 Amélioration

En général, on ne veut pas seulement connaître la valeur numérique de la solution optimale, mais aussi la façon de construire la solution (dans le cas du chamelier, on veut savoir où placer les parenthèses). En étudiant plus finement la mémoire ou en l'enrichissant de données supplémentaires (comme l'origine de la valeur calculée), on peut en la parcourant à l'envers reconstruire une solution au problème.