Cours 3 : De la récursivité à la programmation dynamique

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

Quelle différence entre ces deux programmes ?

```
function fact (n : CARDINAL): CARDINAL;
begin
  if n = 1 then
    fact := 1
  else
    fact := n * fact(n-1);
end {fact};
```

```
function somme (n : CARDINAL, r : CARDINAL): CARDINAL;
begin
  if n = 1 then
    somme := r + 1
  else
    somme := somme (n-1, r + n);
end {somme};
```


Récursivité terminale

Se dit d'une récursivité dont l'appel récursif est la toute dernière instruction réalisée.

- Ce n'est pas le cas dans `fact` : après l'appel récursif il faut faire une multiplication : `n * fact(n-1)`

  Tous les résultats des appels à `fact(n-1)`, `fact(n-2)`, etc. doivent être stockés dans une pile.

$$\begin{array}{ccccccccc} \text{fact}(5) & \leftarrow & \text{fact}(4) & \leftarrow & \text{fact}(3) & \leftarrow & \text{fact}(2) & \leftarrow & \text{fact}(1) \\ 120 & \times 5 & 24 & \times 4 & 6 & \times 3 & 2 & \times 2 & 1 \end{array}$$

- C'est le cas dans `somme` : l'addition `r + n` est faite avant l'appel.

  Pas de stockage de résultats intermédiaires. Les appels successifs sont considérés comme des égalités.

$$\text{som}(5,1) = \text{som}(4,5) = \text{som}(3,9) = \text{som}(2,12) = \text{som}(1,14) = 15$$

Paramètre d'accumulation

Application du même principe pour le calcul de la factorielle :

```
function fact (n : CARDINAL, r : CARDINAL): CARDINAL;
begin
  if n = 1 then
    fact := r
  else
    fact := fact(n-1,r*n);
end {fact};
```

Le paramètre ajouté qui stocke le résultat est appelé paramètre d'accumulation.

Avantages :

- théoriquement plus de nécessité de garder en mémoire la pile d'appels récursifs
- ces programmes peuvent être écrits de manière itérative

"Dérécursivation"

```
function fact (n : CARDINAL, r : CARDINAL): CARDINAL;
begin
  if n = 1 then
    fact := r
  else
    fact := fact(n-1,r*n);
end {fact};
```

```
function fact_iter (n : CARDINAL): CARDINAL;
var
  i, r : CARDINAL
begin
  i := n;
  r := 1;
  while i >= 2 do begin
    r := r * i;
    dec(i);
  end {while};
  fact := r;
end {fact_iter};
```


Un exemple connu : la suite de Fibonacci

Définition du problème :

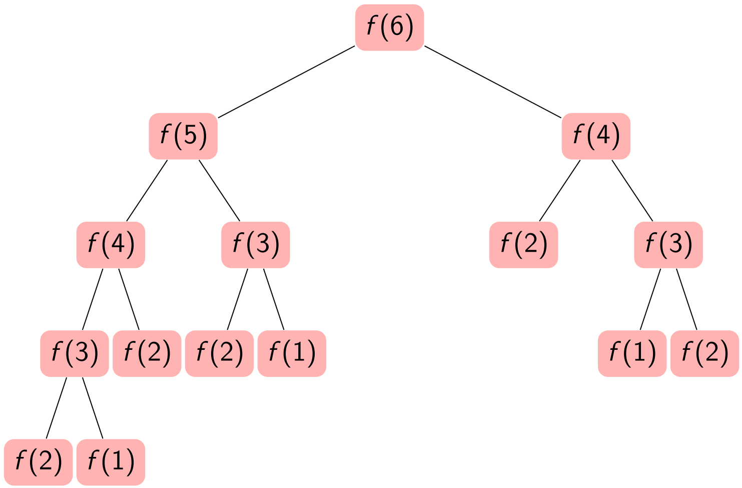
$$\begin{cases} f(1) &= 1 \\ f(2) &= 1 \\ f(n) &= f(n-1)+f(n-2), n > 1 \end{cases}$$

Programmation directe :

```
function fibonacci (n : CARDINAL) : CARDINAL;
begin
   if (n = 2) or (n = 1) then
      fibonacci := 1
   else
      fibonacci := fibonacci(n-1) + fibonacci(n-2);
end {fibonacci};
```


Un exemple connu : la suite de Fibonacci

Arbre des appels récursifs



Nombre d'appels récursifs : 15

Un exemple connu : la suite de Fibonacci

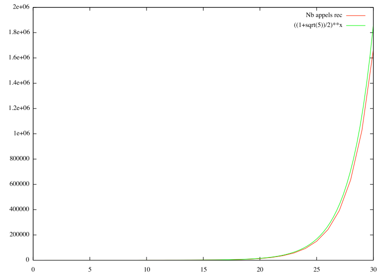
Complexité

Expression du nombre d'appels à la fonction :

$$\begin{cases} c(1) &= 1 \\ c(2) &= 1 \\ c(n) &= 1+c(n-1)+c(n-2) \end{cases}$$

Solution vue au cours précédent :

$$c(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$



Un exemple connu : la suite de Fibonacci

Propriétés

- le calcul de $f(n)$ nécessite de connaître $f(n-1)$ et $f(n-2)$
- on connaît les valeurs pour n=1 et n=2 (les conditions d'arrêt)
- idée : utiliser un tableau contenant les valeurs déjà calculées pour éviter de les recalculer
- utilisation 1 : remplacer les appels récursifs par un accès au tableau lorsque la valeur est calculée
- utilisation 2 : programmer le remplissage du tableau de manière itérative, en partant des valeurs connues
  - on utilise toujours la propriété que si $f(n-1)$ et $f(n-2)$ sont connus, alors on obtient $f(n)$,
  - il suffit de répéter le processus jusqu'au n souhaité.

Un exemple connu : la suite de Fibonacci

Version récursive utilisant un tableau

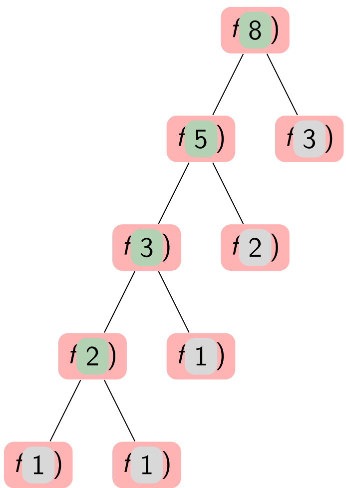
```
function fibonacci_tab_rec (n : CARDINAL) : CARDINAL;
var
   t : array of CARDINAL;
   i : CARDINAL;

   function fibonacci (n : CARDINAL) : CARDINAL;
   begin
      if t[n] = 0 then
         t[n] := fibonacci(n-1) + fibonacci(n-2);
      fibonacci := t[n];
   end {fibonacci};

begin
   setlength(t,n+1);
   t[1] := 1;
   t[2] := 1;
   for i := 3 to n do t[i] := 0;
   fibonacci_tab_rec := fibonacci(n);
end {fibonacci_tab_rec};
```


Un exemple connu : la suite de Fibonacci

Arbre des appels récursifs de `fibonacci_tab_rec`



t :

1	1	2	3	5	8

Nombre d'appels récursifs : 9

Un exemple connu : la suite de Fibonacci

Analyse de la complexité

Cette fois, parmi les 2 appels récursifs effectués à chaque étape, il n'y en a qu'un pour lequel le résultat n'est pas encore stocké dans le tableau.

Expression du nombre d'appels récursifs :

$$\begin{cases} c(1) &= 1 \\ c(2) &= 1 \\ c(n) &= 1 + c(n-1) + 1 \end{cases}$$

La solution est :

$$c(n) = 2 \times n - 3, n \geq 2$$

Un exemple connu : la suite de Fibonacci

Propriétés

- le calcul de $f(n)$ nécessite de connaître $f(n-1)$ et $f(n-2)$
- on connaît les valeurs pour n=1 et n=2 (les conditions d'arrêt)
- idée : utiliser un tableau contenant les valeurs déjà calculées pour éviter de les recalculer
- utilisation 1 : remplacer les appels récursifs par un accès au tableau lorsque la valeur est calculée
- utilisation 2 : programmer le remplissage du tableau de manière itérative, en partant des valeurs connues
  - on utilise toujours la propriété que si $f(n-1)$ et $f(n-2)$ sont connus, alors on obtient $f(n)$,
  - il suffit de répéter le processus jusqu'au n souhaité.

Un exemple connu : la suite de Fibonacci

Version itérative utilisant un tableau

```
function fibonacci_tab_iter (n : CARDINAL) : CARDINAL;
var
   t : array of CARDINAL;
   i : CARDINAL;
begin
   setlength(t,n+1);
   t[1] := 1;
   t[2] := 1;
   for i := 3 to n do
      t[i] := t[i-1] + t[i-2];
   fibonacci_tab_iter := t[n];
end {fibonacci_tab_iter};
```

Complexité :

- en temps : nombre d'additions de la boucle `for` = n-2
- en espace : la taille du tableau de stockage des valeurs intermédiaires = n+1

Note : il est possible d'avoir une complexité en espace constant.

Rappelez-vous les nombres de Catalan

$$\begin{cases} \text{catalan}(0) & = 1 \\ \text{catalan}(1) & = 1 \\ \text{catalan}(n) & = \sum_{k=0}^{n-1} \text{catalan}(n-k-1) \times \text{catalan}(k) \end{cases}$$

Valeur à calculer	Valeurs à connaître
2	$0, 1$
3	$0, 1, 2$
4	$0, 1, 2, 3$
$\vdots$	$\vdots$
n	$0, \ldots, n-1$

Catalan et programmation dynamique

```
type
   TABLE = array [0..50] of CARDINAL;

   function catalan : CARDINAL;
   var
      t : TABLE;
      n, k : CARDINAL;
   begin
      t[0] := 1;
      t[1] := 1;
      for n := 2 to high(t) do begin
         t[n] := 0;
         for k := 0 to n-1 do
            t[n] := t[n] + t[n-k-1] * t[k];
      end {for};
      catalan := t[high(t)];
   end;
```

Complexité :

- en temps : $\Theta(n^2)$ ($\sum_{k=2}^{n} k$ multiplications)
- en espace : $\Theta(n)$ ($n+1$ pour le tableau)

La programmation dynamique

Principe : utiliser une table pour stocker les résultats intermédiaires correspondants aux sous-problèmes

Mise en oeuvre :

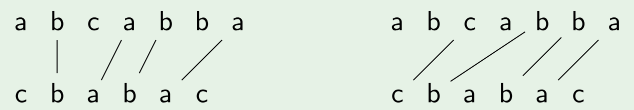
- remplir la table grâce aux valeurs des cas de base (les conditions d'arrête de la récursivité)
- déterminer un sens de remplissage de la table suivant les solutions des sous-problèmes à connaître pour résoudre le problème de taille juste supérieure
- remplir les autres cases de la table, soit avec un parcours itératif, soit un modifiant la version récursive naïve de l'algorithme

Caractéristiques des problèmes traitables

- avoir un problème dont la solution optimale est obtenue par combinaison de solutions optimales de sous-problèmes (principe d'optimalité)
- avoir un algorithme récursif qui nécessite de calculer un grand nombre de fois les mêmes sous-problèmes

La plus longue sous-séquence commune

But : calculer la plus longue sous-séquence commune entre deux chaînes de caractères (sous-séquence = une chaîne dont on efface certains caractères)



baba cbba

Comment formuler le problème ?

Découpage en sous-problèmes

On note u et v les deux chaînes de caractères données en entrée, de longueur respective n et m. On note PLSC(u,v) la longueur de la plus longue sous-séquence commune.

Considérons u' et v' les mots tels que $u'x = u$ et $v'y = v$, x et y sont les dernières lettres de u et v.

Cas 1 si $x = y$ alors on pourra apparier x et y et cette lettre fera partie de la sous-séquence la plus longue, on en déduit :

$$\text{PLSC}(u'x, u'y) = 1 + PLSC(u', v')$$

Cas 2 si $x \neq y$ alors la sous-séquence la plus longue ne peut contenir à la fois la lettre x et la lettre y : x ou y ou aucun des deux ne sera apparié, on en déduit :

$$\text{PLSC}(u'x, v'y) = max \begin{cases} \text{PLSC}(u', v'y) \\ \text{PLSC}(u'x, v') \end{cases}$$

Si l'une des deux chaînes est vide, la PLSC est de longueur nulle.

Version récursive

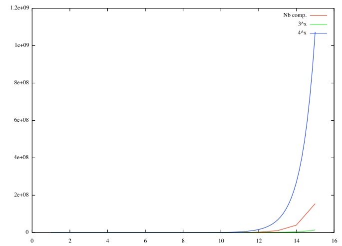
```
function plsc_rec(u,v : STRING) : CARDINAL;
var
   x, y : CHAR;
   uu, vv : STRING;
begin
   if (length(u) = 0) or (length(v) = 0) then
      plsc := 0
   else begin
      x := u[length(u)];
      y := v[length(v)];
      uu := copy(u,low(u),length(u)-1);
      vv := copy(v,low(v),length(v)-1);
      if x = y then
         plsc_rec := 1 + plsc_rec(uu,vv)
      else
         plsc_rec := max (plsc_rec(u,vv), plsc_rec(uu,v));
   end {if};
end {plsc_rec};
```


Complexité de la version récursive

Dans le pire des cas, on réalise toujours le max entre les deux appels récursifs.
Expression du nombre de comparaisons :

$$\begin{cases} c(0,m) &= 0 \\ c(n,0) &= 0 \\ c(n,m) &= 1 + c(n-1,m) + c(n,m-1), n \neq 0, m \neq 0 \end{cases}$$

Si $n = m$,
$c(n,n) = 1 + c(n-1,n) + c(n,n-1)$,
attention, c'est différent de $2 \times c(n-1,n)$



Meilleur des cas laissé en exercice.

Arbre d'appels recursifs

De l'arbre à la table de programmation dynamique :

- repérer les appels redondants
- trouver la dimension de la table
- trouver le sens de remplissage
- obtenir les valeurs initiales

Voir animation multimédia : craie + tableau.

Programmation dynamique

table$[i,j]$ contient la PLSC pour les préfixes de u et v de longueur respective i et j

- conditions initiales : une des deux chaînes est vide
  $$\text{table}[i,0] = 0 \quad \forall i \text{ et table}[0,j] = 0 \quad \forall j$$
- sens de remplissage : si on connaît table$[i-1,j-1]$, table$[i,j-1]$, table$[i-1,j]$ alors on connaît table$[i,j]$
  remplissage des indices les plus petits vers les indices les plus grands, ici peu importe de parcourir d'abord les i ou les j
- résultat : où est stocké le calcul de PLSC(u,v) :
  table$[n,m]$, n longueur de u et m longueur de v

Table de programmation dynamique

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

b a b a

Version programmation dynamique

```
function plsc_dynamique (u,v : STRING) : CARDINAL;
var
   table : array of array  of CARDINAL;
   i, j : CARDINAL;
begin
   setlength(table,length(u)+1);
   for i := 0 to length(u) do setlength(table[i],length(v)+1);

   // initialisation
   for i := 0 to length(u) do table[i][0]  := 0;
   for j := 0 to length(v) do table[0][j]  := 0;
   // remplissage
   for i := 1 to length(u) do
      for j := 1 to length(v) do
         if u[i] = v[j] then
            table[i][j]  := table[i-1][j-1]  + 1
         else
            table[i][j]  := max  (table[i-1][j],table[i,j-1]);
   // resultat
   plsc_dynamique := table[length(u)][length(v)];
end {plsc_dynamique};
```


Complexité de la version programmation dynamique

En espace :

- une table de la taille le produit de la longueur des deux chaînes + 1,
- donc en $\Theta(n \times m)$

En temps (toujours en nombre de comparaisons) :

- le test `u[i] = v[j]` est réalisé pour toutes les cases sauf celles d'indice zéro,
- donc en $\Theta(n \times m)$

La complexité en temps n'est pas systématiquement la taille de la table, il se peut que le calcul d'une case de la table ne s'obtienne pas en temps constant.

Reconstruction de la solution optimale

On a la longueur, maintenant on voudrait obtenir la suite de lettres correspondante.

On parcourt la table en sens inverse une fois qu'elle est calculée :

- si $u[i] = v[j]$ alors le résultat de table$[i][j]$ provient de la case table$[i-1][j-1]$
- sinon, le résultat de table$[i][j]$ provient du max entre la case table$[i][j-1]$ et table$[i-1][j]$

Table de programmation dynamique

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0 ⟵	0	0	1	1	1	1	1
2	b	0	0	1 ⟵	1	1	2	2	2
3	a	0	1	1	1	2 ⟵	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

b a b a

Version avec reconstruction

```
table := ...;
res := '';
i := length(u);
j := length(v);
while (i > 0) and (j > 0) do begin
   if u[i] = v[j] then begin
      res := u[i] + res;
      i := i - 1;
      j := j - 1;
   end else begin
      if table[i][j] = table[i-1][j] then begin
         i := i - 1;
      end else begin
         j := j - 1;
      end {if};
   end {if};
end {while};
```