Algorithmique Avancée et Complexité: Programmation Dynamique

AAC

Sophie Tison-USTL-Master1 Informatique

La programmation dynamique est :

Un schéma d'algorithme exhibé dans les années 1950 par Bellman et basé sur deux idées simples

La programmation dynamique est :

Un schéma d'algorithme exhibé dans les années 1950 par Bellman et basé sur deux idées simples

- Résoudre un problème grâce à la solution de sous-problèmes

La programmation dynamique est :

Un schéma d'algorithme exhibé dans les années 1950 par Bellman et basé sur deux idées simples

- Résoudre un problème grâce à la solution de sous-problèmes
- Eviter de calculer deux fois la même chose, i.e. la solution du même sous-problème.

Un exemple basique illustrant le deuxième point

On cherche à calculer la suite définie par :

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = F_1 = 1$$

Un exemple basique illustrant le deuxième point

On cherche à calculer la suite définie par :

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = F_1 = 1$$

Quel est le nom de cette suite?

Un exemple basique illustrant le deuxième point

On cherche à calculer la suite définie par :

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = F_1 = 1$$

Quel est le nom de cette suite?

La suite de Fibonacci

L'algorithme "naturel":

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = F_1 = 1$$

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (n <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

Calculer le Nombre d'appels

Soit $A(n)$ le nombre d'appels à *Fib*-y compris le principal- lors de l'évaluation de $Fib(n)$

Comment le calculer?

```
int Fib (int n) {
if ((n<=1)) return 1;
else
return Fib(n-1)+Fib(n-2);}
```

Calculer le Nombre d'appels

Soit $A(n)$ le nombre d'appels à *Fib*-y compris le principal- lors de l'évaluation de $Fib(n)$

Comment le calculer?

```
int Fib (int n) {
```
L'appel principal
```
if ((n<=1)) return 1;
else
return Fib(n-1)+Fib(n-2);}
```

Calculer le Nombre d'appels

Soit $A(n)$ le nombre d'appels à *Fib*-y compris le principal- lors de l'évaluation de $Fib(n)$

Comment le calculer?

```
int Fib (int n) {
```
L'appel principal
```
if ((n<=1)) return 1;
```
pas d'appel interne si p=0 ou p=1
```
else
return Fib(n-1)+Fib(n-2);}
```

Calculer le Nombre d'appels

Soit $A(n)$ le nombre d'appels à *Fib*-y compris le principal- lors de l'évaluation de $Fib(n)$

Comment le calculer?

```
int Fib (int n) {
```
L'appel principal
```
if ((n<=1)) return 1;
```
pas d'appel interne si p=0 ou p=1
```
else
return Fib(n-1)+Fib(n-2);}
```
A(n-1)+A(n-2) appels internes

Calculer le Nombre d'appels

Soit $A(n)$ le nombre d'appels à *Fib*-y compris le principal- lors de l'évaluation de $Fib(n)$

Comment le calculer?

```
int Fib (int n) {
```
L'appel principal
```
if ((n<=1)) return 1;
```
pas d'appel interne si p=0 ou p=1
```
else
return Fib(n-1)+Fib(n-2);}
```
A(n-1)+A(n-2) appels internes

Donc:

$A(0) = A(1) = 1$

$1 < n : A(n) = 1 + A(n-1) + A(n-2)$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Donc:

$$A(n) ? Fib(n)$$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Donc:

$$A(n) \geq Fib(n)$$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Donc:

$$A(n) \geq 2^{n/2}?$$

Calculer le Nombre d'appels

$$A(0) = A(1) = 1$$
$$1 < n : A(n) = 1 + A(n-1) + A(n-2)$$

Donc:

$$A(n) \geq 2^{n/2}?$$

Exo: Par récurrence.

Arbre des appels?

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (p <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

Arbre des appels?

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (p <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

- Arbre binaire
- Longueur minimale d'une branche

Arbre des appels?

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (p <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

- Arbre binaire
- Longueur minimale d'une branche: n/2
- Longueur maximale d'une branche:

Arbre des appels?

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (p <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

- Arbre binaire
- Longueur minimale d'une branche: n/2
- Longueur maximale d'une branche: n-1

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f ??? N_i$$

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f = 1 + N_i$$

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f = 1 + N_i$$

$$? \leq N_f \leq ?$$

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f = 1 + N_i$$

$$2^{l_{min}} \leq N_f \leq ?$$

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f = 1 + N_i$$

$$2^{l_{min}} \leq N_f \leq 2^h$$

Quelques rappels sur les arbres

Soit un arbre binaire: un noeud est soit binaire soit une feuille. Soit N_i son nombre de noeuds internes, N_f son nombre de feuilles, h sa hauteur, l_{min} la longueur minimale d'une branche.

$$N_f = 1 + N_i$$

$$2^{l_{min}} \leq N_f \leq 2^h$$

$$\text{si } h = l_{min}\ N_f = 2^h$$

Nombre d'appels?

```
//Précondition:  n entier >=0,
int Fib (int n) {
  if (p <=1)
     return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

- Arbre binaire
- Longueur minimale d'une branche: n/2
- Longueur maximale d'une branche: n-1
- nombre de noeuds internes= nombre d'appels $\geq 2^{n/2} - 1$
- nombre de noeuds internes= nombre d'appels $\leq 2^{n-1} - 1$

Calculer le Nombre d'appels -Bis

Soit $B(n)$ le nombre d'appels internes à Fib lors de l'évaluation de $Fib(n)$. Comment le calculer?

```
int Fib (int n) { if ((p<=1)) return 1;
else
return Fib(n-1)+Fib(n-2);}
```

Calculer le Nombre d'appels -Bis

Soit $B(n)$ le nombre d'appels internes à *Fib* lors de l'évaluation de $Fib(n)$. Comment le calculer?

```
int Fib (int n) { if ((p<=1)) return 1;
```

0 appel interne si p=0 ou p=1

```
else
return Fib(n-1)+Fib(n-2);}
```

Calculer le Nombre d'appels -Bis

Soit $B(n)$ le nombre d'appels internes à *Fib* lors de l'évaluation de $Fib(n)$. Comment le calculer?

```
int Fib (int n) { if ((p<=1)) return 1;
```

0 appel interne si p=0 ou p=1

```
else
return Fib(n-1)+Fib(n-2);}
```

2+B(n-1)+B(n-2) appels internes

Calculer le Nombre d'appels -Bis

Soit $B(n)$ le nombre d'appels internes à *Fib* lors de l'évaluation de $Fib(n)$. Comment le calculer?

```
int Fib (int n) { if ((p<=1)) return 1;
```

0 appel interne si p=0 ou p=1

```
else
return Fib(n-1)+Fib(n-2);}
```

2+B(n-1)+B(n-2) appels internes

Donc:

$B(0) = B(1) = 0$

$1 < n : B(n) = 2 + B(n-1) + B(n-2)$

Que dire de la complexité de l'algorithme?

L'algorithme est impraticable.

Remarque

On se contente de borner inférieurement le nombre d'appels si on veut juste montrer qu'il est "mauvais"!!!

Inutile de calculer précisément son ordre de grandeur!

Revenons à la programmation dynamique

Pourquoi la complexité de l'algo est-elle mauvaise?

Revenons à la programmation dynamique

Pourquoi la complexité de l'algo est-elle mauvaise?

On recalcule de nombreuses fois les même valeurs!

UNE SOLUTION ITÉRATIVE

```
// 1<=n
int F[] =new int[n+1];
  F[0]=1;
  F[1]=1;
   for (int i=2; i<=n; i++) F[i]=F[i-1]+F[i-2];
```

Complexité?

UNE SOLUTION ITÉRATIVE

```
// 1<n
int F[] =new int[n+1];
  F[0]=1;
  F[1]=1;
   for (int i=2; i<=n;i++) F[i]=F[i-1]+F[i-2];
```

Complexité en $\Theta(n)$

Analyse de cette solution:

- Complexité en $\Theta(n)$

Analyse de cette solution:

- Complexité en $\Theta(n)$
- La taille de la donnée est $\log n$. L'algo est-il polynomial?

Analyse de cette solution:

- Complexité en $\Theta(n)$
- La taille de la donnée est $\log n$. L'algo est-il polynomial? On dit qu'il est pseudo-polynomial.
- Le coût uniforme est-il raisonnable?

Analyse de cette solution:

- Complexité en $\Theta(n)$
- La taille de la donnée est $\log n$. L'algo est-il polynomial? On dit qu'il est pseudo-polynomial.
- Le coût uniforme est-il raisonnable? Pas vraiment ici...
- Complexité spatiale: essentiellement le tableau des $n+1$ valeurs stockées: en $\Theta(n)$, ou plutôt en Θ(n*taille max des entiers manipulés), ce qui donnerait ici $\Theta(n^2)$ (Q? Pourquoi?). Comment l'améliorer?

Analyse de cette solution:

- Complexité en $\Theta(n)$
- La taille de la donnée est $\log n$. L'algo est-il polynomial? On dit qu'il est pseudo-polynomial.
- Le coût uniforme est-il raisonnable? Pas vraiment ici...
- Complexité spatiale: essentiellement le tableau des $n+1$ valeurs stockées: en $\Theta(n)$, ou plutôt en Θ(n*taille max des entiers manipulés), ce qui donnerait ici $\Theta(n^2)$ (Q? Pourquoi?). Comment l'améliorer?
  On n'a besoin que des deux dernières valeurs.
- Il existe d'autres solutions pour calculer la suite de Fibonacci!

Un exemple de décomposition d'une solution:

Le problème:
On a un triangle de n lignes de nombres entiers.

Un exemple de décomposition d'une solution:

Le problème:
On a un triangle de n lignes de nombres entiers.
On part du sommet.

Un exemple de décomposition d'une solution:

Le problème:
On a un triangle de n lignes de nombres entiers.
On part du sommet.
A chaque étape, on choisit à la ligne du dessous un des deux nombres adjacents.

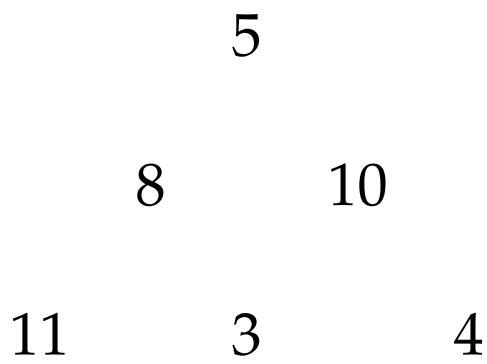
Un exemple de décomposition d'une solution:

Le problème:
On a un triangle de n lignes de nombres entiers.
On part du sommet.
A chaque étape, on choisit à la ligne du dessous un des deux nombres adjacents.
On s'arrête quand on est sur la dernière ligne.

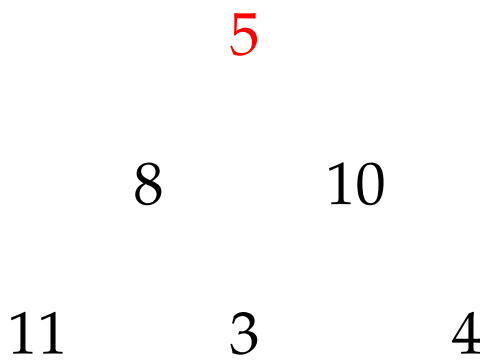
Un exemple de décomposition d'une solution:

Le problème:
On a un triangle de n lignes de nombres entiers.
On part du sommet.
A chaque étape, on choisit à la ligne du dessous un des deux nombres adjacents.
On s'arrête quand on est sur la dernière ligne.
On cherche à maximiser la somme totale des nombres choisis.

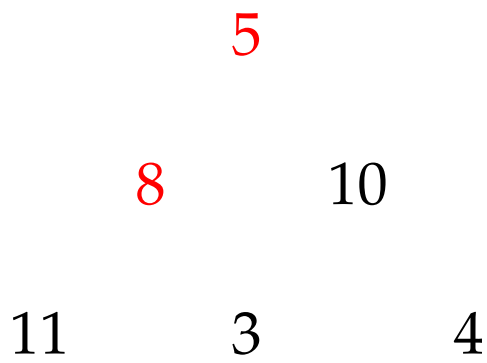
Un exemple:



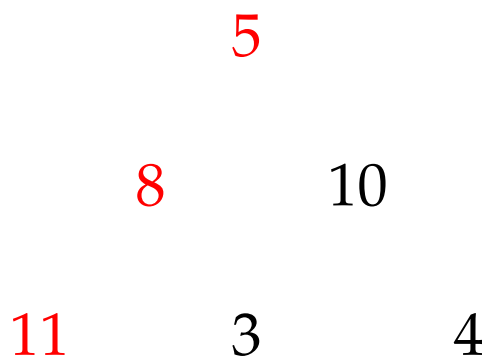
Un exemple:



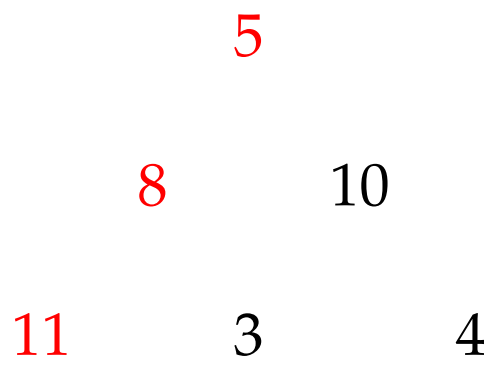
Un exemple:



Un exemple :



UN EXEMPLE :



Combien de chemins possibles?

Un exemple :

$$\begin{array}{ccccc} & & 5 & & \\ & 8 & & 10 & \\ 11 & & 3 & & 4 \end{array}$$

Combien de chemins possibles?

2^{n-1}

La décomposition d'une solution

- Sous-problème:

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet":

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple),

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple?

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple? $l = n - 1$:

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple? $l = n-1 : G(n-1, r) = val(n-1, r)$

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple? $l = n-1 : G(n-1, r) = val(n-1, r)$
- Récurrence?

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple? $l = n-1$: $G(n-1, r) = val(n-1, r)$
- Récurrence? $0 \leq l \leq n-2, 0 \leq r \leq l$:

La décomposition d'une solution

- Sous-problème: le meilleur gain à partir d'un "sommet" quelconque (de bas en haut par exemple)
- Identifier les paramètres d'un sous-problème: un "sommet": son numéro de ligne (de bas en haut par exemple), son rang dans la ligne (de gauche à droite par exemple)
- $G(l, r)$: gain maximum à partir de la case (l, r)
  $0 \leq l \leq n-1, 0 \leq r \leq l$
- Cas simple? $l = n-1$: $G(n-1, r) = val(n-1, r)$
- Récurrence? $0 \leq l \leq n-2, 0 \leq r \leq l$:
  $G(l, r) = val(l, r) + max(G(l+1, r), G(l+1, r+1))$

LA SOLUTION "NAÏVE"

$G(n-1,r) = val(n-1,r)$
$0 \leq l \leq n-2, 0 \leq rl$
$G(l,r) = val(l,r) + max(G(l+1,r), G(l+1,r+1))$

D'où l'algorithme:

```
// val un tableau "triangle" de n lignes
int Gain (l,r) {
  if  l=n-1 return val(l,r)
  else
     return val(l,r)+max(Gain(l+1,r),Gain(l+1,r+1))
```

Complexité?

COMPLEXITÉ DE LA SOLUTION "NAÏVE"

```
// val un tableau "triangle" de n lignes
int Gain (l,r) {
  if  l=n-1 return val(l,r)
  else
     return val(l,r)+max(Gain(l+1,r),Gain(l+1,r+1))
```

Complexité en $\Theta(2^n)$ donc impraticable!

Est-on dans le cadre de la programmation dynamique?

Environ 2^n appels.

Est-on dans le cadre de la programmation dynamique?

Environ 2^n appels.

Il ne peut y avoir plus d'appels "différents" que d'entiers, soit $n(n+1)/2$!

Est-on dans le cadre de la programmation dynamique?

Environ 2^n appels.

Il ne peut y avoir plus d'appels "différents" que d'entiers, soit $n(n+1)/2$!

Donc on recalcule de nombreuses fois les mêmes valeurs!

Est-on dans le cadre de la programmation dynamique?

Environ 2^n appels.

Il ne peut y avoir plus d'appels "différents" que d'entiers, soit $n(n+1)/2$!

Donc on recalcule de nombreuses fois les mêmes valeurs!

On est bien dans le cadre de la programmation dynamique: on va utiliser une table (par exemple) pour mémoriser les valeurs.

LA SOLUTION DYNAMIQUE ITÉRATIVE :

$G(n-1,r) = val(n-1,r)$
$0 \leq l \leq n-2, 0 \leq r \leq l$
$G(l,r) = val(l,r) + max(G(l+1,r), G(l+1,r+1))$

```
// val  un tableau "triangle" de n lignes
int Gain[][] = new int[n][n];
// init: cas de base
  for ( int r=0 ; i<=n-1;i++)
     Gain[n-1][r]=Val[n-1][r];
//remplissage selon récurrence
  for ( int l=n-2; l>=0; l--)
     for ( int r=0 ; r<=l ; r++)
        Gain[l][r]= Val[l][r]
           +Maths.max(Gain[l+1][r],Gain[l+1][r+1]);
  return Gain[0][0];
```

Complexité?

LA SOLUTION DYNAMIQUE ITÉRATIVE :

```
// val  un tableau "triangle" de n lignes
int Gain[][] = new int[n][n];
// init: cas de base
  for ( int r=0 ; i<=n-1;i++)
     Gain[n-1][r]=Val[n-1][r];
//remplissage selon récurrence
  for ( int l=n-2; l>=0; l--)
     for ( int r=0 ; r<=l ; r++)
        Gain[l][r]= Val[l][r]
           +Maths.max(Gain[l+1][r],Gain[l+1][r+1]);
  return Gain[0][0];
```

Complexité en $\Theta(n^2)$

RÉCUPÉRER LA STRATÉGIE?

Une fois la table *Gain* remplie, on fait une "remontée" (en fait ici une descente!) dans la table pour récupérer la stratégie.

```
// val  un tableau "triangle" de n lignes
// Gain le tableau rempli des gains optimaux
 int l=0;
 int r=0;
 //on se positionne à la case de départ
 while (l <n-1) {
     if  Gain(l+1,r+1) > Gain (l+1, r) r++;
                                 // on va à droite!
     l++;
     "sortir l,r";
     }
```

UNE SOLUTION RÉCURSIVE DYNAMIQUE

```
// val un tableau "triangle" de n lignes
// G une table qui stocke les valeurs calculées
//  DejaCalc table de booléens
//          qui indique si une valeur est calculée
int Gain (l,r) {
  if  l==n-1 return val(l,r)
  else if DejaCalc[l][r]   return G[l][r];
       //on retourne la valeur stockée
  else
   {DejaCalc[l][r]=true;
     return G[l][r]=val(l,r)
         +max(Gain(l+1,r),Gain(l+1,r+1));
      //on calcule, stocke et retourne
       }
```

Complexité?

UNE SOLUTION RÉCURSIVE DYNAMIQUE

```
// val un tableau "triangle" de n lignes
// G une table qui stocke les valeurs calculées
//  DejaCalc table de booléens
//          qui indique si une valeur est calculée
int Gain (l,r) {
  if  l==n-1 return val(l,r)
  else if DejaCalc[l][r]   return G[l][r];
       //on retourne la valeur stockée
  else
   {DejaCalc[l][r=true;
     return G[l][r]=val(l,r)
         +max(Gain(l+1,r),Gain(l+1,r+1));
      //on calcule, stocke et retourne
       }
```

Complexité en $\Theta(n^2)$

UNE AUTRE SOLUTION RÉCURSIVE DYNAMIQUE

```
// val un tableau "triangle" de n lignes
// table G stocke les valeurs calculées
// G initialisée à 0
// on suppose les entiers   strictement positifs
// 0 est donc valeur sentinelle
int Gain (l,r) {
  if  l==n-1 return val(l,r)
  else if G[l][r] >0 return G[l][r];
           //on retourne la valeur stockée
  else  return G[l][r]=val(l,r)
             + max (Gain (l+1,r), Gain (l+1,r+1));
           //on calcule, stocke et retourne
```

Complexité en $\Theta(n^2)$

Quand peut-on utiliser la programmation dynamique?

. La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) de problèmes de taille inférieure à n -c'est le principe d'optimalité-.

Quand peut-on utiliser la programmation dynamique?

. La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) de problèmes de taille inférieure à n -c'est le principe d'optimalité-.

. Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

Comment utiliser la programmation dynamique?

On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final.

Comment utiliser la programmation dynamique?

On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final.
Il faut donc qu'on puisse déterminer les sous-problèmes (ou un sur-ensemble de ceux-ci) qui seront traités au cours du calcul (ou un sur-ensemble de ceux-ci) ...

Comment utiliser la programmation dynamique?

On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final.
Il faut donc qu'on puisse déterminer les sous-problèmes (ou un sur-ensemble de ceux-ci) qui seront traités au cours du calcul (ou un sur-ensemble de ceux-ci) ...
Ensuite il faut remplir cette table; il y a deux approches, l'une itérative, l'autre récursive.

La version itérative

On initialise les ”cases” correspondant aux cas de base.

La version itérative

On initialise les ”cases” correspondant aux cas de base.

On remplit ensuite la table selon un ordre bien précis à déterminer: on commence par les problèmes de ”taille” la plus petite possible, on termine par la solution du problème principal: **il faut bien sûr qu'à chaque calcul, on n'utilise que les solutions déjà calculées**.
Le but est bien sûr que chaque élément soit calculé une et une seule fois.

La version récursive (tabulation, memoizing)

A chaque appel, on regarde dans la table si la valeur a déjà été calculée (donc une ”case” correspond à un booléen et une valeur ou à une seule valeur si on utilise une valeur ”sentinelle”).

La version récursive (tabulation, memoizing)

A chaque appel, on regarde dans la table si la valeur a déjà été calculée (donc une "case" correspond à un booléen et une valeur ou à une seule valeur si on utilise une valeur "sentinelle").

Si oui, on ne la recalcule pas: on récupère la valeur mémorisée.

La version récursive (tabulation, memoizing)

A chaque appel, on regarde dans la table si la valeur a déjà été calculée (donc une "case" correspond à un booléen et une valeur ou à une seule valeur si on utilise une valeur "sentinelle").

Si oui, on ne la recalcule pas: on récupère la valeur mémorisée.

Si non, on la calcule, on mémorise qu'on l'a calculée et on stocke la valeur correspondante.

La version récursive (tabulation, memoizing)

Donc si la version récursive naïve est:

```
fonction f(parametres p)
  si cas_de_base(p) alors g(p)
  sinon h(f(p_1),...,f(p_k))
```

le schéma de l'algorithme récursif dynamique sera:

```
//tabcalcul: dictionnaire des valeurs calculées
//...un tableau, ou une table de hachage
// le sous-problème-ses paramètres- est la clé
fonction fdynrec(parametres p)
{si non (tabcalcul.contains(p))
   alors //on calcule et on mémorise
   {val = (si casdebase(p) alors g(p)
    sinon h(fdynrec(p_1),...,fdynrec(p_k)));
    tabcalcul. ajouter(p,val);} ;
 retourner tabcalcul.valeur(p);}
```

Conception d'un algorithme de programmation dynamique

L'essentiel du travail conceptuel réside dans l'expression d'une solution d'un problème en fonction de celles de problèmes "plus petits"!!!

Un autre exemple: la plus longue sous-suite commune

Le problème: Une sous-suite (ou sous-mot) d'un mot u est un mot w obtenu à partir de u en effaçant des lettres:
aac est sous-suite de *arracher*, de *avancer*, de *hamac*...

Un autre exemple: la plus longue sous-suite commune

Le problème: Une sous-suite (ou sous-mot) d'un mot u est un mot w obtenu à partir de u en effaçant des lettres:
aac est sous-suite de *arracher*, de *avancer*, de *hamac*...
Soient deux mots u et v. On cherche la longueur de la (ou d'une) plus longue sous-suite commune des deux mots ainsi qu'une telle sous-suite. On notera $lcs(u, v)$ cette longueur.

Un autre exemple: la plus longue sous-suite commune

Le problème: Une sous-suite (ou sous-mot) d'un mot u est un mot w obtenu à partir de u en effaçant des lettres:
aac est sous-suite de *arracher*, de *avancer*, de *hamac*...
Soient deux mots u et v. On cherche la longueur de la (ou d'une) plus longue sous-suite commune des deux mots ainsi qu'une telle sous-suite. On notera $lcs(u, v)$ cette longueur.
Par exemple, si $u = acc$ et $v = archi$,

Un autre exemple: la plus longue sous-suite commune

Le problème: Une sous-suite (ou sous-mot) d'un mot u est un mot w obtenu à partir de u en effaçant des lettres:
aac est sous-suite de *arracher*, de *avancer*, de *hamac*...
Soient deux mots u et v. On cherche la longueur de la (ou d'une) plus longue sous-suite commune des deux mots ainsi qu'une telle sous-suite. On notera $lcs(u, v)$ cette longueur.
Par exemple, si $u = acc$ et $v = archi$, *ac* est la sous-suite de longueur maximale et donc $lcs(u, v)$ vaut 2.

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$
- la récurrence:

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$
- la récurrence:
  - si $u_i = v_j$,

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$
- la récurrence:
  - si $u_i = v_j$, $LCS(i, j) = 1 + LCS(i - 1, j - 1)$

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$
- la récurrence:
  - si $u_i = v_j$, $LCS(i, j) = 1 + LCS(i - 1, j - 1)$
  - si $u_i \neq v_j$,

Analyse du problème

Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1..u_i$ -les i premières lettres de u- et $v_1..v_j$ -les j premières lettres de v-. On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$
- la récurrence:
  - si $u_i = v_j$, $LCS(i, j) = 1 + LCS(i - 1, j - 1)$
  - si $u_i \neq v_j$, $LCS(i, j) = max(LCS(i - 1, j), LCS(i, j - 1))$

Version récursive naïve

```
....int solvpart(int i, int j) {
    //retourne LCS(i,j)
     if (i==0) return 0;
     else if (j==0) return 0;
     else if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
          return 1+solvpart(i-1, j-1);
     else return
           max (solvpart(i-1,j),solvpart(i, j-1));}
```

Complexité de la version récursive naïve?

```
....int solvpart(int i, int j) {
    //retourne LCS(i,j)
     if (i==0) return 0;
     else if (j==0) return 0;
     else if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
          return 1+solvpart(i-1, j-1);
     else return
         max (solvpart(i-1, j),solvpart(i, j-1));}
```

La complexité dans le pire des cas (en nombre d'appels) est au moins de l'ordre de $2^{min(u.length(),v.length())}$.

Complexité de la version récursive naïve?

```
....int solvpart(int i, int j) {
    //retourne LCS(i,j)
     if (i==0) return 0;
     else if (j==0) return 0;
     else if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
          return 1+solvpart(i-1, j-1);
     else return
          max (solvpart(i-1, j),solvpart(i, j-1));}
```

La complexité dans le pire des cas (en nombre d'appels) est au moins de l'ordre de $2^{min(u.length(),v.length())}$

Le nombre d'appels différents est au plus $(1 + u.length() * (1 + v.length)$.

On est dans le cadre de la programmation dynamique!

Version dynamique itérative

```
..int sol (PbLCS pb){
   int T[][]=new int[pb.u.length()+1][pb.v.length()
   //T[i][j] memorisera LCS(u[0..i-1],v[0,..j-1])
   //cas de base:
   for (int i=0;i<=pb.u.length();i++) T[i][0]=0;
   for (int j=0;j<=pb.v.length();j++) T[0][j]=0;
   //la récurrence:
   for (int i=1;i<=pb.u.length();i++) {
         for (int j=1;j<=pb.v.length();j++) {
            if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
               T[i][j]=1+T[i-1][j-1];
            else  T[i][j]=max(T[i][j-1],T[i-1][j]);
   return T[pb.u.length()][pb.v.length()];}
```

La remontée, ou comment récupérer une sous-suite commune de longueur maxi

```
//Précondition TCalc[i][j]=LCS(i,j) pour les
//   valeurs   ``nécessaires'' (à formaliser...)
  String s=""; //s contiendra une sous-suite maxi
  int i=pb.u.length();
  int j=pb.v.length();
  //(i,j) représentent la ``case courante''
  // on part de la case ``finale''
  // et on remonte jusqu'au cas de base
  while ((i>0)&&(j>0)){
    if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
        {s= (pb.u.substring(i-1,i)).concat(s);
        i--;j--;}
    else if (T[i][j]==T[i-1][j])   i--;
    else j--; }
  return s;
```