Programmation Dynamique

ACT

Sophie Tison
Master Informatique
Université de Lille

Trois paradigmes d'algorithmes

Algorithmic Design Pattern

- Rappels
- Paradigmes d'algorithme
  - Diviser pour régner
  - Programmation Dynamique
  - Algorithmes Gloutons
- Complexité des problèmes
- Algorithmique Avancée

Un dernier exemple: La paire de points les plus proches

Le problème :
Soit P un ensemble de n points du plan.
On cherche la paire de points les plus proches (pour la distance euclidienne).

Pour simplifier, on supposera que tous les points ont une abscisse différente (pas nécessaire mais plus simple pour la présentation)

Récapitulatif

1. Trier les points selon x
2. Partager en deux parties de même cardinal (à 1 près) par une droite verticale $x = med$, G, D.
3. Résoudre à gauche et à droite. cela donne une distance minimale min_G à gauche, min_D à droite.
4. Trouver, si il existe un point de G et un point de D, à distance inférieure à $\delta = min(min_G, min_D)$; si oui, soit la distance minimale min_{GD} entre un point de G et un point de D
5. La distance minimale est donc le minimum de min_G, min_D, ou min_{GD} selon la réponse du 4.

Coût du point 4?

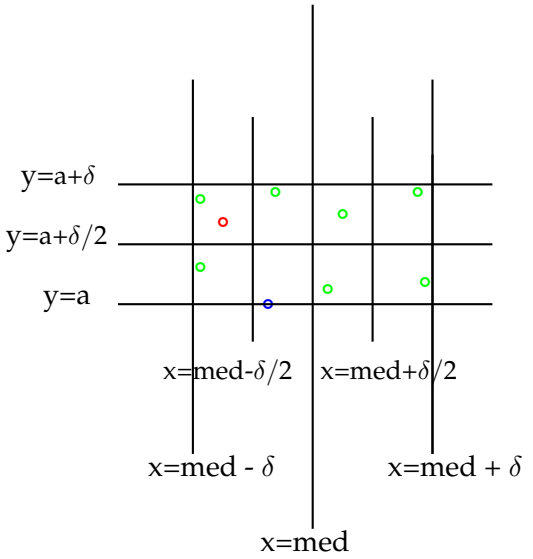
UN LEMME...

Remarque: si deux points sont à distance $< \delta$ et de part et d'autre de la droite $x = med$, ils sont à distance au plus δ de la droite.
On peut donc limiter la recherche aux points de S l'ensemble des points (x, y) tels que $med - \delta \leq x \leq med + \delta$.

Lemma

Soit $\delta = min(min_G, min_D)$
Soit S trié par y croissant.
Alors si deux points de chaque côté de la droite sont séparés par au moins 7 points dans la liste, ils sont à distance $> \delta$.

Au plus un point par "carré"



RÉCAPITULONS

1. Initialisation
   - 1.1 trier les points selon x pour construire S_x :
   - 1.2 trier les points selon y pour construire S_y
2. Partager en deux parties de même cardinal (à 1 près) par une droite verticale $x = med$: on obtient G (G_x, G_y), D (D_x, D_y).
3. Résoudre à gauche et à droite. Cela donne une distance minimale min_G à gauche, min_D à droite.
4. Trouver, si il existe un point de G et un point de D, à distance inférieure à $\delta = min(min_G, min_D)$; on notera alors la distance minimale min_{GD} entre un point de G et un point de D
5. La distance minimale est donc le minimum de min_G, min_D, resp. min_{GD}.

La complexité est donnée par l'équation $C(n) = 2C(n/2) + O(n)$
L'algorithme sera en $O(n \log n)$.

DIVISER POUR RÉGNER: QUELQUES EXEMPLES CLASSIQUES

- les algorithmes de recherche dichotomique,
- le tri fusion
- les algorithmes de multiplication: entiers (Karatsuba), matrices (Strassen)
- La transformée de Fourier rapide, FFT
- Géométrie algorithmique: Enveloppe convexe.

Diviser pour régner: le Bilan

- Diviser un problème en problèmes de taille nettement plus petite (si possible divisée par une constante), attention à la recombinaison!
- Appliquer si possible le *Master Theorem* pour la complexité
- Souvent plus facile à écrire en récursif... même si cela peut bien sûr se faire en itératif
- Peut être intéressant à paralléliser.

Trois paradigmes/schémas d'algorithmes

Algorithmic Design Pattern

- Diviser pour régner
- Programmation Dynamique
- Algorithmes Gloutons

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 :

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

Quel est le nom de cette suite?

La suite de Fibonacci

LA SUITE DE FIBONNACI

Leonardo Bonacci (c. 1170 – c. 1250)

LA SUITE DE FIBONNACI

$F_n = \frac{\Phi^n - \frac{1}{\Phi^n}}{\sqrt{5}}$ avec $\Phi = \frac{1+\sqrt{5}}{2}$ le nombre d'or.

LA SUITE DE FIBONNACI

L'ALGORITHME "NATUREL"

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

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
if $(n \leq 1)$ **then**
 | **return** 1;
else
 | **return** (Fib(n-1)+Fib(n-2));
Complexité ? (en coût uniforme)

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?

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
if $(n \leq 1)$ **then**
 | **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?

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n) L'appel principal
if $(n \leq 1)$ **then**
 | **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?

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n) L'appel principal
if $(n \leq 1)$ **then**
 | **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?

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n) L'appel principal
if $(n \leq 1)$ **then**
 | **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
Méthode 1

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

Donc, c'est une récurrence linéaire d'ordre 2 qu'on sait résoudre.

Calculer le Nombre d'appels
Méthode 2

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

Donc:

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

Calculer le Nombre d'appels
Méthode 2

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

Donc:

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

Calculer le Nombre d'appels
Méthode 3

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

Donc:
$A(0) = A(1) = 1$ et si $n \geq 2$, $A(n) \geq 2 * A(n-2)$
Donc:

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

(Exo: le montrer par récurrence.)

Arbre des appels?

Méthode 4

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
if $(n \leq 1)$ **then**
 | **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.

$$N_f ??? = 1 + N_i$$

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

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

Nombre d'appels?

Méthode 4

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
if $(n \leq 1)$ **then**
 | **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?

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
if $(n \leq 1)$ **then**
 | **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:
$B(0) = B(1) = 0$
$1 < n : B(n) = 2 + B(n-1) + B(n-2)$


Bien définir ce qu'on calcule pour poser l'équation.

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". Il est inutile de calculer précisément son ordre de grandeur.

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

Input: Précondition: n entier ≥ 0
Output: retourne F_n
int Fib (int n)
int F[] =new int[n+1];
F[0]=1; F[1]=1;
for *int* $i = 2; i \leq n; i++$ **do**
| $F[i] = F[i-1] + F[i-2]$;
end
return F[n];
Complexité? : en $\Theta(n)$

D'où vient le miracle?

Algo récursif "naïf" en $\Omega(2^{n/2})$

Algo itératif en $\Theta(n)$

Dans la version récursive, de nombreux appels sont effectués avec les mêmes paramètres!

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!

La programmation dynamique est

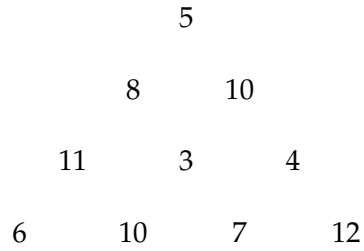
Un schéma d'algorithme exhibé dans les années 1950 par Bellman et basé sur deux idées simples

- 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.

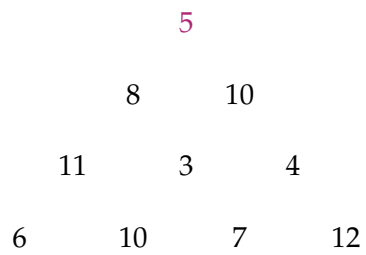
Un exemple de décomposition d'une solution:

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.

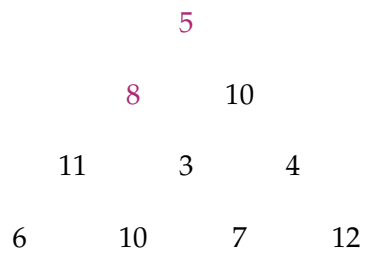
Un exemple:



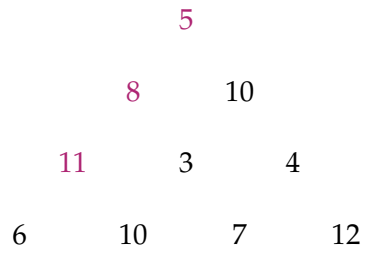
UN EXEMPLE:



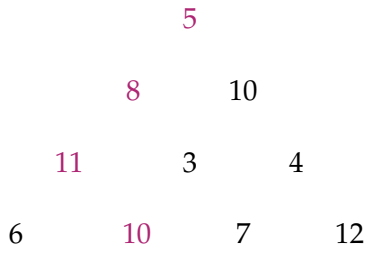
UN EXEMPLE:



UN EXEMPLE :



UN EXEMPLE :



Combien de chemins possibles?

2^{n-1}

La décomposition d'une solution

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

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

D'où l'algorithme:

Input: *val* est un tableau "triangle" de n lignes
Output: retourne le gain maxi
int Gain(int l, int r) {
if *(l==n-1)* **then**
  return val(l,r);
else
  return (val(l,r)+max(Gain(l+1,r),Gain(l+1,r+1)));
Complexité?
Complexité en $\Theta(2^n)$ donc impraticable!

Est-on dans le cadre de la programmation dynamique?

Environ 2^n appels.

Il ne peut y avoir plus d'appels "différents" que de (l,r), avec $0 \leq l \leq n-1, 0 \leq r \leq l$ 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.

Intuition: dans la solution naïve, on parcourt tous les chemins "sans partager"; en utilisant la programmation dynamique, on ne parcourt pas deux fois le même tronçon de chemin.

Solution dynamique itérative

Input: *val* est un tableau "triangle" de n lignes
Output: retourne le gain maxi
int Gain[][] = new int[n][n];
// init: cas de base
for $(int\ r = 0; i \leq n-1; i++)$ **do**
  Gain[n-1][r]=Val[n-1][r];
end
//remplissage selon récurrence
for $(int\ l = n-2; l \geq 0; l--)$ **do**
  for $(int\ r = 0; r \leq l; r++)$ **do**
    Gain[l][r] = Val[l][r]+
      max(Gain[l+1][r],Gain[l+1][r+1]);
  end
end
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.

Input: *val* est un tableau "triangle" de n lignes
Input: Gain le tableau rempli des gains optimaux
int l=0, r=0; //on se positionne à la case de départ
while $(l < n-1)$ **do**
  if $(Gain(l+1, r+1) > Gain(l+1, r))$ **then**
    r++; // on va à droite!
  // sinon à gauche, r ne change pas
  l++;
  Sortir (l,r);
end

Une solution récursive dynamique

Input: *val* est un tableau "triangle" de n lignes
// table G pour stocker les valeurs calculées
// DejaCalc table de booléens
// qui indique si une valeur est calculée
int Gain (l,r)
if *(l==n-1)* **then**
  return (val(l,r));
if *!DejaCalc[l][r]* **then**
  DejaCalc[l][r]=true;
  (G[l][r]=val(l,r)+ max(Gain(l+1,r),Gain(l+1,r+1)));
  //on calcule et stocke
return (G[l][r]);
//on retourne la valeur stockée
Complexité? en $\Theta(n^2)$

Une autre solution récursive dynamique

Input: *val* est un tableau "triangle" de n lignes
// table G pour stocker les valeurs calculées, initialisée à 0
// on suppose les entiers strictement positifs
// 0 est valeur sentinelle
int Gain (l,r) **if** $(l == n-1)$ **then**
  return (val(l,r));
else if $(G[l][r] > 0)$ **then**
  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-.

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

Le principe d'optimalité

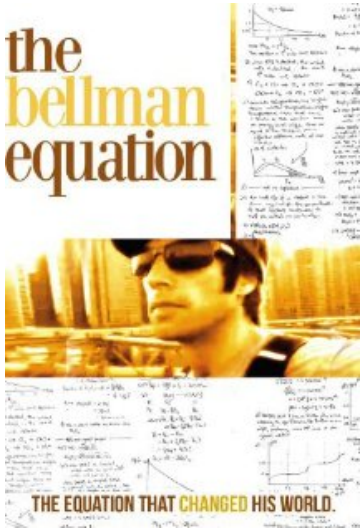
- si on cherche le plus court chemin entre deux points (cf algorithme de Floyd): si le chemin le plus court entre A et B passe par C, le tronçon de A à C (resp. de C à B) est le chemin le plus court de A à C (resp. de C à B); le principe est respecté.
- si on cherche le plus long chemin sans boucle d'un point à un autre: si le chemin le plus long sans boucle de A à C passe par B, le tronçon de A à B n'est pas forcément le plus long chemin sans boucle de A à B! le principe n'est pas respecté.

Richard Bellman

Le concept de programmation dynamique a été introduit par Richard Bellman pour résoudre des problèmes de recherche opérationnelle dans les années 50.
Le terme "programmation dynamique" - les différentes valeurs sont mises à jour dynamiquement- a été choisi dans un souci de "marketing".

The Bellman Equation

retrace la vie de Richard Bellman.



Richard Bellman

extrait du site du film

"One of the most influential mathematicians of the 20th century, Mr. Bellman was a pioneer in the field of computer science. Bellman's work changed the perception of the application of mathematics within science. The term 'Bellman Equation' is a type of problem named after its discoverer, in which a problem that would otherwise be not possible to solve is broken into a solution based on the intuitive nature of the solver. Applied in control theory, economics, and medicine, it has become an important tool in using math to solve really difficult problems. A fiercely independent and controversial figure, Mr. Bellman published more than 40 books and 600 papers before his early death of a brain tumor. The inventor of the field of dynamic programming was persecuted by McCarthy, worked on the Manhattan project, and remained as much a mystery as the problems he conquered."

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.
- 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 si la valeur a déjà été calculée (dsoit en utilisant une table de booléens ou une valeur "sentinelle" dans la table des valeurs).
- Si oui, on récupère la valeur mémorisée.
- Si non, on la calcule, on mémorise qu'on l'a calculée et la valeur correspondante.

La version récursive (tabulation, memoizing)

Donc si la version récursive naïve est

```
fonction f(parametres p)
if cas_de_base(p) then
  | return (g(p))
else
  | return h(f(p_1),...,f(p_k))
```

le schéma de l'algorithme récursif dynamique sera

```
//tabcalcul: dictionnaire des valeurs calculées
// le sous-problème-ses paramètres- est la clé
fonction fdynrec(parametres p)
if cas_de_base(p) then
  | return g(p);
if non (tabcalcul.contains(p)) then
  | val = h(fdynrec(p_1),...,fdynrec(p_k)));
  | tabcalcul.ajouter(p,val);
return 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"!

Principe utilisé dans beaucoup d'algorithmes de domaines divers

Recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov....

- algorithme de Warshall-Floyd
- calcul de la distance de deux chaînes (cf *diff* d'Unix), problèmes d'alignement de séquences
- algorithme de Viterbi
- algorithme de Cocke-Younger-Kasami, algorithme d'Earley (analyse syntaxique de phrases)
- calcul d'arbres binaires optimaux
- triangulation d'un polygone, ...

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 effçant 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.

Exemple: si $u = act$ 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:

- 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 $u_i \neq v_j$, $LCS(i, j) = max(LCS(i - 1, j), LCS(i, j - 1))$

Le cœur de l'algorithme

- $LCS(i,0) = 0 = LCS(0,j), 0 \leq i \leq |u|, 0 \leq j \leq |v|$
- $1 \leq i \leq |u|, 1 \leq j \leq |v|$
  si $u_i = v_j$ $LCS(i,j) = 1 + LCS(i-1, j-1)$
  sinon $LCS(i,j) = max(LCS(i-1,j), LCS(i,j-1))$

Version récursive naïve

Input: u,v deux mots
Output: retourne LCS(i,j) pour u et v
int solvpart(int i, int j)
if $((i == 0)||(j == 0))$ **then**
  return (0);
else if $(pb.u.charAt(i-1) == pb.v.charAt(j-1))$ **then**
  return (1+solvpart(i-1, j-1));
else
  return ((solvpart(i-1,j),solvpart(i, j-1)));

Complexité de la version récursive naïve?

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

Dans le meilleur des cas, le nombre d'appels est $min(u.length(), v.length())$.

Complexité de la version récursive naïve?

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:

$$(1 + u.length) \times (1 + v.length)$$

.

On est dans le cadre de la programmation dynamique!

Version dynamique itérative

```
int T[][]=new int[pb.u.length()+1][pb.v.length()+1];
//T[i][j] memorisera LCS(u[0..i-1],v[0,..j-1])
for (int i = 0; i ≤ pb.u.length(); i++) do
  T[i][0] = 0;
end
for (int j = 0; j ≤ pb.v.length(); j++) do
  T[0][j] = 0;
end
for (int i = 1; i ≤ pb.u.length(); i++) do
    for (int j = 1; j ≤ pb.v.length(); j++) do
        if (pb.u.charAt(i − 1) == pb.v.charAt(j − 1)) then
          T[i][j] = 1 + T[i − 1][j − 1];
        else
          T[i][j] = max(T[i][j − 1], T[i − 1][j]);
    end
end
return (T[pb.u.length()][pb.v.length()]);
```

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

Input: TCalc[i][j]=LCS(i,j) pour les valeurs “nécessaires”
String s=””; //s contiendra une sous-suite maxi
int $i = pb.u.length(), j = pb.v.length()$;
//(i,j) représentent la “case courante”
// on part de la case “finale” et remonte jusqu’au cas de base
while $((i > 0)\&\&(j > 0))$ **do**
  if $(pb.u.charAt(i-1) == pb.v.charAt(j-1))$ **then**
    $s = (pb.u.substring(i-1, i)).concat(s)$;
    $i--; j--$;
  else if $(T[i][j] == T[i-1][j])$ **then**
    $i--$;
  else
    $j--$;
end
return (s);