Université de Lille

FST- Département Informatique

ACT-Cours

2018–2019
Master d'Informatique-S1
S.Tison

Programmation dynamique

La programmation dynamique est un "paradigme" simple de conception d'algorithmes souvent utilisé pour les problèmes d'optimisation. On peut l'appliquer si:

1. La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) (d'un "petit" nombre) de problèmes de taille inférieure à n: c'est *le principe d'optimalité énoncé par R. Bellman*, une stratégie optimale se base sur des sous-stratégies optimales.

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

Comment? L'idée est simplement d'éviter de calculer plusieurs fois la solution du même sous-problème. On définit une table pour mémoriser les calculs déjà effectués: une case correspondra donc à un et un seul problème intermédiaire et contiendra la solution correspondante, 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, soit de façon "bottom-up" (version itérative), soit de façon "top-down" (version récursive:

- *Version itérative -la classique-*: 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.

- Version récursive et *memoization*: A chaque appel, on vérifie si la valeur a déjà été calculée (donc un sous-problème 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 et on stocke la valeur correspondante. Donc si la fonction f est définie par:

```
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 pour calculer f sera:

```
//tablecalcul est un dictionnaire contenant les valeurs déjà calculées
//on peut  utiliser un tableau, ou une table de hachage
// le sous-problème (ou les paramètres le déterminant) est la clé
fonction fdynrec(parrametres p) {
si non tablecalcul.contains(p)  alors //on calcule et on mémorise
   {val = (si casdebase(p) alors g(p) sinon h(fdynrec(p_1),...,fdynrec(p_k)));
   tablecalcul.ajouter(p,val);}
retourner tablecalcul.valeur(p); }
```

Quelques Remarques:

- *L'art du découpage* L'essentiel du travail conceptuel réside dans la définition des paramètres qui caractérisent un problème et l'expression d'une solution d'un problème en fonction de celles de problèmes "plus petits". Ensuite, on peut analyser ce qui se passe dans une implémentation récursive "naïve": si la solution de mêmes problèmes est recalculée plusieurs fois, on est dans le cadre de la programmation dynamique. La définition des paramètres doit naturellement conduire à la définition de la table: un élément de la table correspond à un sous-problème -des valeurs de paramètres-. Ensuite, il faut encore réfléchir un tout petit peu pour savoir dans quel ordre effectuer le remplissage de la table, tout du moins dans le cas itératif.

- *Itératif versus récursif* La version récursive, souvent un peu plus simple à écrire, peut parfois être plus efficace que l'itérative : en effet, on ne calcule que les appels nécessaires, alors que dans le cas itératif, pour certains problèmes on calcule beaucoup de valeurs inutiles, tout du moins si on n'optimise pas.

- *Complexité en temps* Attention, le nombre de sous-problèmes peut être grand: l'algorithme obtenu n'est pas forcément polynomial.

- *Complexité en espace* A priori la complexité en espace correspond à la table. Il n'est pas toujours nécessaire de mémoriser toutes les valeurs calculées. On a par contre souvent besoin de stocker toutes les valeurs calculées quand on doit effectuer une remontée dans la table (voir ci-dessous). Dans les versions récursives, on pourra utiliser des tables de hachage pour adapter le mieux possible l'espace mémoire consommé à celui réellement nécessaire.

Le principe d'optimalité de Bellman

Bien que naturel, ce principe n'est pas toujours applicable! Prenons l'exemple des chemins dans un graphe:

- le principe est respecté 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);

- il ne l'est plus 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 terme et le concept de programmation dynamique ont été introduits par Richard Bellman pour résoudre des problèmes d'affectation et d'optimisation (recherche opérationnelle) dans les années 50. Le terme "programmation dynamique" peut paraître un peu étrange. Il repose sur le fait que les différentes valeurs sont mises à jour dynamiquement. R. Bellman, alors chercheur à la Rand Corporation, devait rendre des comptes au secrétaire à la Défense et aurait choisi ce nom dans un souci de "marketing". Un film documentaire, "The Bellman Equation", retrace la vie de Richard Bellman.

Programmation dynamique et Optimisation

La programmation dynamique est souvent utilisée dans le cadre de problèmes d'optimisation: on cherche une solution optimale par rapport à un certain coût (ou fonction objectif) et non seulement le meilleur coût. Or, la table construite contient des coûts et non des solutions. Dans une première phase, on calcule donc seulement le coût d'une solution optimale; il est souvent facile ensuite de récupérer la (une) solution optimale, toujours en utilisant l'information contenue dans la table (éventuellement en enrichissant un petit peu la table pou mémoriser d'où vient une valeur); grosso modo, on parcourt la table à l'envers par rapport à son remplissage, i.e. en partant de la case correspondant au problème principal et en "remontant" vers les plus petits problèmes: la solution -qui peut être associée à une suite de choix- correspond au chemin dans la table. On appelle ce processus "la remontée dans la table".

Quelques exemples classiques

Beaucoup d'algorithmes dans des domaines divers -recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov....- utilisent le principe de la programmation dynamique. Citons-en quelques exemples: l'algorithme de Warshall-Floyd (clôture transitive d'une relation, plus court chemin dans un graphe), l'algorithme de Viterbi, le calcul de la distance de deux chaînes (cf *diff* d'Unix) et plus généralement les problèmes d'alignement de séquences, l'algorithme de Cocke-Younger-Kasami pour vérifier si un mot est généré par une grammaire algébrique, l'algorithme d'Earley (analyse syntaxique de phrases), les arbres binaires optimaux, la triangulation d'un polygone ...

Un 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. Dans un premier temps, on ne calcule que $lcs(u, v)$. Puis on calculera une sous-suite de longueur maximale. Ce problème peut être vu comme une version simple du problème d'alignement de deux séquences.
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 $u_i \neq v_j$, $LCS(i, j) = max(LCS(i - 1, j), LCS(i, j - 1))$

On en déduit donc 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)));}
```

Analyse de la complexité de la version récursive naÏve: La complexité dans le pire des cas (en nombre d'appels) est au moins de l'ordre de $2^{min(u.length(),v.length())}$. Pourtant, le nombre d'appels récursifs différents (i.e. le nombre de valeurs différentes possibles pour les paramètres des appels intermédiaires) est au plus $(|u| + 1) * (|v| + 1)$, les paramètres d'un sous-problème étant i , $0 <= i <= |u|$ et j , $0 <= j <= |v|$.

On est donc typiquement dans le cas de la programmation dynamique! On utilisera donc une table d'entiers à deux dimensions de taille $(|u| + 1) * (|v| + 1)$ pour stocker les résultats des sous-problèmes.

Version itérative dynamique avec un tableau:

```
..int sol (PbLCS pb){
   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])
   //cas de base:
   for (int i=0;i<=pb.u.length();i++) T[i][0]=0; // cas i=0
   for (int j=0;j<=pb.v.length();j++) T[0][j]=0;  //cas 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()];}
```

Version récursive avec deux tables, une pour mémoriser si une valeur est déjà calculée, une pour les valeurs:

```
   ... int TCalc[][];  //TCalc(i,j) mémorisera LCS(i,j)
   ... boolean DejaCalc[][];
   ...
   //précondition: si DejaCalc[x][y] TCalc[x][y]==LCS(x,y)
   ... int calc(int i,int j) { //calcule, mémorise et retourne LCS(i,j)
       if (DejaCalc[i][j])  return TCalc[i][j];
       else { DejaCalc[i][j]=true;
              if ((i==0) || (j==0)) return (TCalc[i][j]=0);
              else if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
                          return (TCalc[i][j]=1+calc(i-1,j-1));
              else return (TCalc[i][j]=max(calc(i-1,j),calc(i,j-1)));}}
```

Version récursive avec une seule table, utilisant une valeur sentinelle:

```
   //précondition: si  TCalc[x][y]>=0 alors TCalc[x][y]==LCS(x,y)
   //on initialisera TCalc à la valeur sentinelle -1
    ... int calc(int i,int j) { //calcule, mémorise et retourne LCS(i,j)
        if (TCalc[x][y]>=0)  return TCalc[i][j];
        else { if ((i==0) || (j==0)) return (TCalc[i][j]=0);
                   else if (pb.u.charAt(i-1)==pb.v.charAt(j-1))
                               return (TCalc[i][j]=1+calc(i-1,j-1));
                    else return (TCalc[i][j]=max(calc(i-1,j),calc(i,j-1)));}}
```

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

```
//Précondition TCalc est remplie,
// i.e. TCalc[i][j]=LCS(i,j) pour les valeurs ‘‘nécessaires’’ (mériterait une formalisation...)
  String s=""; // s contiendra une  sous-suite maxi
  int i=pb.u.length(), j=pb.v.length();
  //(i,j) représentent les coordonnées de 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;
```

Evaluation Paresseuse et Programmation dynamique

Dans des langages fonctionnels non stricts (comme Haskell), on peut utiliser l'évaluation paresseuse: une valeur n'est calculée que si vraiment on en a besoin ("call by need"). Cela permet de programmer de façon très simple des algorithmes de type "programmation dynamique". On définit la table avec les valeurs d'initialisation et la formule de récurrence; pour le "calcul" d'une valeur, ne seront calculées -et une seule fois- que les valeurs nécessaires. Pour l'exemple précédent, un programme Haskell pourrait être:

```
pluslonguesoussuite a b =  lcs m n
 where (m, n) = (length a, length b)
       a'     = Array.listArray (1, m) a
       b'     = Array.listArray (1, n) b
       lcs i 0 = 0
       lcs 0 j = 0
       lcs i j
         | a !! (i - 1) ==  b !! (j - 1) = lcst ! (i - 1, j - 1) +1
         | otherwise = maximum [ lcst ! (i - 1, j)
                                              , lcst ! (i, j - 1)
                                             ]

       lcst = Array.listArray bounds
              [lcs i j | (i, j) <- Array.range bounds]
       bounds = ((0, 0), (m, n))
```

Pour aller plus loin sur ce point, voir par exemple le blog http://jelv.is/blog/Lazy-Dynamic-Programming/.