Complexité (Quatrième année)

TD5

Maud MARCHAL

6 avril 2011

Programmation efficace des formules récursives

6 avril 2011

Plan

Introduction

- Décomposition recursive : principe fondamental en algorithmique.
- Méthode "Diviser pour Résoudre" : conduit à des programmes efficaces.
- **Mais** : la programmation directe de formules récursives peut conduire à des inefficacités fortes.
- Premier exemple : suite de Fibonacci.

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned}$$

Plan

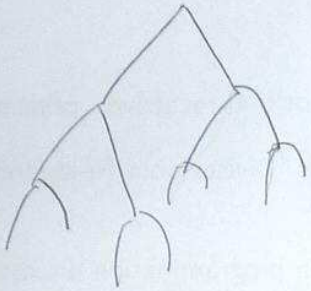
Exemple : suite de Fibonacci

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned}$$

- Programmation directe :

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

suite de Fibonacci : graphe des appels



Complexité exponentielle !

Pas de sous pb. indép
=> pas de diviser par résoudre.

Plan

Elimination des calculs redondants : principe

- L'élimination des calculs redondants est une technique de résolution qui peut être utilisée chaque fois que l'écriture récursive d'un algorithme conduit à effectuer plusieurs fois le même calcul.
- L'approche repose sur une *évaluation ascendante* des résultats intermédiaires.
- Cette technique est employée dans la **programmation dynamique** : résolution de problèmes d'optimisation s'exprimant par une équation de récurrence.

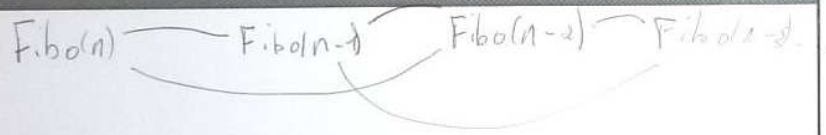
Approche descendante/ascendante

- Approche "Diviser pour Résoudre" : il n'y a pas de problèmes de redondance car le processus de résolution est effectué sur 2 moitiés disjointes des données : approche descendante (top-down).
- Par opposition, dans le cours d'aujourd'hui, on va parler d'une **approche ascendante** (bottom-up).
- L'élimination des calculs redondants conduira généralement à l'écriture d'algorithmes itératifs.

Elimination des calculs redondants : une approche ascendante

- Au lieu d'utiliser les appels d'une procédure récursive comme un arbre, on peut utiliser un graphe en confondant les sommets qui portent la même étiquette : on obtient un **graphe des appels**.
- Ce graphe permet de visualiser la présence de calculs redondants.
- Des techniques de marquage ou d'exécution dans un *ordre topologique* peuvent alors être utilisées pour éliminer les redondances.

Premier exemple : suite de Fibonacci



1. Dessiner le graphe des appels pour la suite de Fibonacci.
2. Quelle technique peut-on utiliser pour éviter de traiter deux fois le même nœud ?

Stocker
On peut utiliser une technique de marquage.
Au premier passage par chaque nœud, on le marque
on évite ainsi de refaire le calcul ultérieurement.

⚠ marquer ne suffit pas, il faut stocker.
On passe d'un arbre à un graphe

Fibonacci : graphe des appels

Fibonacci : algorithme (bis)

```
int TabFib[NMAX]
for (i=0; i<NMAX; i++){
    TabFib[i] = -1;
}
TabFib[0]=0;
TabFib[1]=1;
int Fibo(int n)
{
    if(TabFib[n]>=0)
        return TabFib[n];
    else
        TabFib[n] = Fibo(n-1)+Fibo(n-2);
        return TabFib[n];
}
```

Coût : O(n)
Coût mémoire O(n)

Fibonacci : solution au coût mémoire

1. Question : Proposer une autre méthode pour limiter le coût mémoire.

Analyse le graphe d'appels en regardant le graphe on peut trouver ~~traiter~~ le nb de données à garder en mémoire
Pour tabuler Fibo(n) on a uniquement besoin des valeurs de Fibo(n-1) et Fibo(n-2)

Fibonacci : algorithme comme solution au coût mémoire

```
int Fibo(n){
  int a,b, aux;
  if(n==0 || n==1)
    return n;
  else
    a=0;
    b=1;
    for(i=2; i<=n; i++)
    aux = b
    b = a+b
    a = aux
    }return b;
}
```


Méthodologie et synthèse

1. Caractériser la solution d'un problème ayant une formulation récursive.
2. Montrer qu'il y a des calculs redondants en dessinant le graphe des appels d'une procédure.
3. Eliminer les redondances avec (au choix) 2 types d'algorithmes :
   - Calcul avec marquage et stockage des résultats partiels, de manière à éviter de faire plusieurs fois le même calcul.
   - Calcul selon un ordre topologique sur les noeuds du graphe (économie de place mémoire mais il faut faire attention au coût en temps avec les algorithmes itératifs).

   La première méthode peut être utilisée de manière systématique, la deuxième demande plus de réflexion.

Plan

Combinaisons C_n^p

Les nombres C_n^p sont définis par la formule suivante :

$$C_n^p = \frac{n * \cdots * n-p+1}{p * \ldots 1}$$

Il n'est pas très intéressant de calculer le numérateur et le dénominateur, puis de faire le quotient...
On utilise une formule de récurrence.

Combinaisons C_n^p

Les nombres C_n^p sont définis par la formule de récurrence suivante :

$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1} \text{ pour} 0 < p < n$$
$$C_n^0 = C_n^n = 1$$

1. Dessiner le graphe des appels correspondant à une solution récursive naïve.
2. Donner une solution qui évite les calculs redondants en utilisant une méthode de marquage.
3. Trouver un autre algorithme, basé sur un ordonnancement "intelligent" des tâches, qui permette d'éviter les redondances tout en limitant si possible la place mémoire utilisée.

Combinaisons C_n^p : algorithme naïf

```
int comb(int n, int p) {
  if(p==0 || p==n)
    return 1;
  else return comb(n-1,p)+comb(n-1,p-1);
}
```

$C(2k,k) = \underbrace{\frac{2k*(2k-1)*\cdots*(k+1)}{k*(k-1)*\cdots*1}}_{\geq 2^k}$.

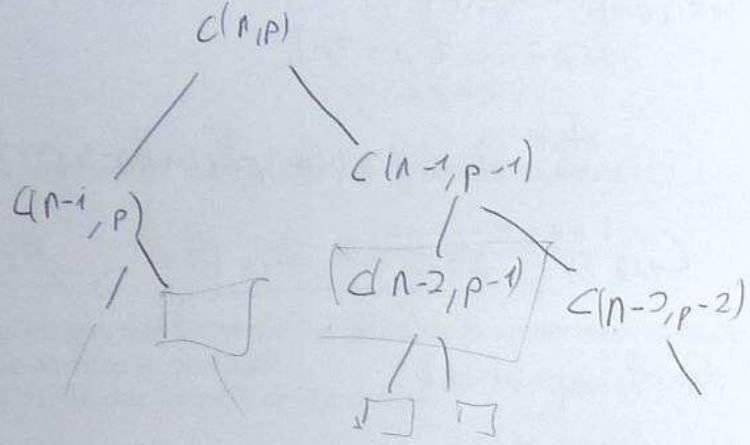
l'algo est impraticable.

$A(n,0) = A(n,n) = 1$

$A(n,p) = 1 + A(n-1,p)\ A(n-1,p-1)$

Combinaisons C_n^p : graphe des appels

Combinaisons C_n^p : complexité de la solution naïve



Parenthèse : 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.
On a :

$$\begin{aligned} N_f &= 1 + N_i \\ 2^{l_{min}} \leq\ & N_f \leq 2^h \end{aligned}$$

Dans le cas des combinaisons :

- La longueur minimale d'une branche est $\min(p, n-p)$
- La hauteur de l'arbre des appels est n
- Donc le nombre d'appels est supérieur ou égal à $2^{\min(p, n-p)}$

Combinaisons C_n^p : algorithme avec marquage

Matrice n×p ou tab (n-p)×p.
initialisation à -1

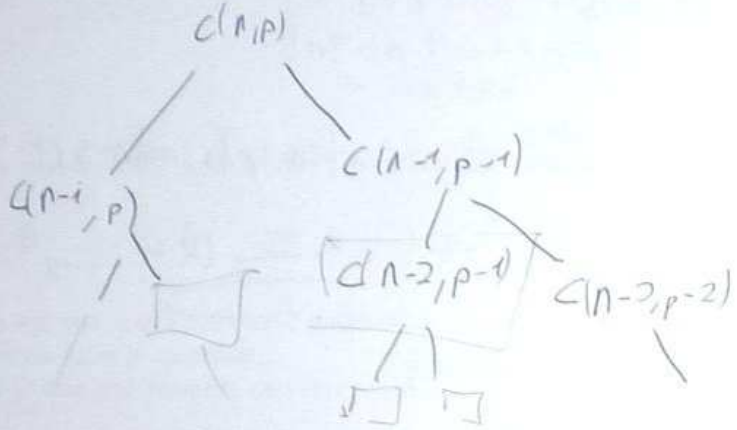
```
int comb(int n, int p){
  if(p==0 || p==n){
    return 1;
  else (tab[n][p]==-1)
    tab[n][p] = comb(n-1,p-1) + comb(n-1,p);
  return tab[n][p];
}
```

coût en $O(np)$ opérations
place mémoire : $(np - p^2) : O(np)$

Combinaisons C_n^p : algorithme itératif

- Objectif : bon ordonnancement des tâches (triangle de Pascal).
- Algorithme :
  Idée : calculs ligne par ligne en remontant dans le graphe des appels ; stocker une ligne de taille p avant de calculer la ligne du dessus.
  - Calculer la ligne d'en dessous jusqu'à C_{n-1}^p, stocker les résultats.
  - Calculer la queue de la ligne courante, de C_{n-p}^0 en remontant jusqu'à C_{n-1}^{p-1} (on peut écraser au fur et à mesure les valeurs stockées dans le début du tableau).
  - Sommer les résultats obtenus pour C_{n-1}^p (case p du tableau) et pour C_{n-1}^{p-1} (case p-1 du tableau).

Combinaisons C_n^p : complexité de la solution naïve



Parenthèse : 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.
On a :

$$\begin{aligned} N_f &= 1 + N_i \\ 2^{l_{min}} \leq N_f &\leq 2^h \end{aligned}$$

Dans le cas des combinaisons :

- La longueur minimale d'une branche est $\min(p, n-p)$
- La hauteur de l'arbre des appels est n
- Donc le nombre d'appels est supérieur ou égal à $2^{\min(p,n-p)}$

Combinaisons C_n^p : algorithme avec marquage

Matrice n×p ou tab (n-p)×p.
initialisation à -1

```
int comb(int n, int p){
  if(p==0 || p==n){
    return 1;
  else (tab[n][p]==-1)
    tab[n][p] = comb(n-1,p-1) +
    comb(n-1,p);
  return tab[n][p];
}
```

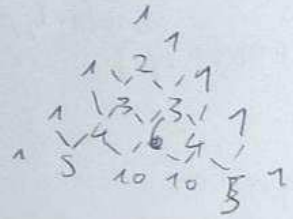
coût en O(np) opérations
place mémoire : $(np-p^2)$: O(np)

Combinaisons C_n^p : algorithme itératif

- Objectif : bon ordonnancement des tâches (triangle de Pascal).
- Algorithme :
  Idée : calculs ligne par ligne en remontant dans le graphe des appels ; stocker une ligne de taille p avant de calculer la ligne du dessus.
  - Calculer la ligne d'en dessous jusqu'à C_{n-1}^p, stocker les résultats.
  - Calculer la queue de la ligne courante, de C_{n-p}^0 en remontant jusqu'à C_{n-1}^{p-1} (on peut écraser au fur et à mesure les valeurs stockées dans le début du tableau).
  - Sommer les résultats obtenus pour C_{n-1}^p (case p du tableau) et pour C_{n-1}^{p-1} (case p-1 du tableau).

Combinaisons C_n^p : algorithme itératif

```
pour i dans 1..p faire tab(i):=1;
pour k dans 1..n-p faire
    pour i dans 1..p faire
        tab(i) = tab(i) + tab(i-1)
return tab(p);
```



Diviser pour résoudre vs programmation dynamique

Ce sont 2 méthodes de conception d'algorithme possédant des points communs :

- Elles partent d'un problème P à résoudre de taille n.
- Ce problème P de taille n peut être solutionné en :
  - séparant P en x sous-problèmes de taille $< n$,
  - résolvant séparément ces x sous-problèmes,
  - combinant les solutions de ces x sous-problèmes.

Complexité de Diviser pour Résoudre

- Lorsque x=2 et que les 2 sous-problèmes sont de taille n/2, la complexité de l'algorithme obtenu est $O(nlog(n))$.
- Lorsque x=2 et que les 2 sous-problèmes sont de taille $n-1$ et $n-2$, la complexité de l'algorithme est $O(2^n)$, mais souvent dans ce cas, les sous-problèmes ne sont pas **indépendants**.

C'est ce constat qui est à la base de la programmation dynamique.

Caractéristiques de la programmation dynamique

- La programmation dynamique s'intéresse essentiellement à des problèmes d'**optimisation** (Richard Bellman 1940).
- En offrant des algorithmes plus efficaces que ceux obtenus par "Diviser pour Résoudre".

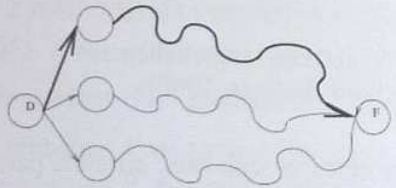
Conditions d'utilisation de la programmation dynamique

C'est une méthode qui permet de réduire la complexité temporelle des algorithmes d'optimisation à condition que :

- Les sous-problèmes ne soient pas totalement indépendants : on parle alors de redondance ou recouvrement. Dans ce cas, on est amené à calculer plusieurs fois la solution des mêmes sous-problèmes.
- La solution optimale du problème s'exprime en fonction des solutions optimales des sous-problèmes.
  C'est le **principe d'optimalité** de Richard Bellman.

Principe d'optimalité

- Le principe d'optimalité signifie que les solutions optimales des sous-problèmes peuvent être utilisées pour construire la solution optimale du problème complet.



- Exemple : la valeur du plus court chemin entre D et F peut être construite en :
  - calculant la valeur du plus court chemin de chaque successeur de D à F,
  - utilisant ces résultats partiels pour déterminer la valeur du plus court.
- Attention : cela ne marche pas si on cherche le chemin le plus long sans boucle d'un point à un autre

Démarche commune aux problèmes d'optimisation

- Ces problèmes admettent généralement un grand nombre de solutions.
- A chaque solution est attachée une valeur,
- Et on cherche à trouver la solution dont la valeur est optimale.

Démarche commune aux problèmes d'optimisation

La conception d'un algorithme d'optimisation utilisant la programmation dynamique comporte les étapes suivantes :

1. Trouver une décomposition du problème en sous-problèmes qui satisfait le principe d'optimalité.
2. Trouver l'expression à optimiser (maximiser ou minimiser) pour obtenir la valeur de la solution optimale à partir des solutions optimales partielles.
3. Organiser le calcul de la valeur de la solution optimale.
4. Construire la solution optimale à partir de l'étape précédente.