IFT2125 - Introduction à l'algorithmique

Programmation dynamique (B&B chapitre 8)

Pierre McKenzie

DIRO, Université de Montréal

Automne 2017

COEFFICIENT BINOMIAL

DONNÉE: entiers $0 \leq k \leq n$

CALCULER: $\binom{n}{k}$

Voici un algo diviser-pour-régner pour COEFFICIENT BINOMIAL :

function $C(n, k)$
  if $k = 0$ **or** $k = n$ **then return** 1
  else return $C(n-1, k-1) + C(n-1, k)$

Deux usages inefficaces de diviser-pour-régner

2) Probabilité de l'emporter en série mondiale, B&B Section 8.1.2

<u>SÉRIE MONDIALE</u>

DONNÉE: probabilité $0 \leq p \leq 1$ que A batte B lors d'un seul match, entiers $n > 0$ et $0 \leq i, j < i + j < 2n$

CALCULER: probabilité $P(i, j)$ que A gagne n matchs avant B, sachant qu'il manque à A i victoires et à B j victoires

Quel serait un algo diviser-pour-régner ?

Deux usages inefficaces de diviser-pour-régner

2) Probabilité de l'emporter en série mondiale, B&B Section 8.1.2

<u>SÉRIE MONDIALE</u>

DONNÉE: probabilité $0 \leq p \leq 1$ que A batte B lors d'un seul match, entiers $n > 0$ et $0 \leq i, j < i + j < 2n$

CALCULER: probabilité $P(i, j)$ que A gagne n matchs avant B, sachant qu'il manque à A i victoires et à B j victoires

Quel serait un algo diviser-pour-régner ?

function $P(i, j)$
  if $i = 0$ **then return** 1
  else if $j = 0$ **then return** 0
  else return $pP(i-1, j) + qP(i, j-1)$

Bien meilleure solution pour Série mondiale
Par “programmation dynamique”

$$\underbrace{P(i,j) = pP(i-1,j) + qP(i,j-1)}_{\text{suggère de remplir un tableau une diagonale à la fois de haut en bas}}$$

function *series*(n, p)
    array $P[0..n, 0..n]$
    $q \leftarrow 1 - p$
    {Fill from top left to main diagonal}
    for $s \leftarrow 1$ **to** n **do**
        $P[0, s] \leftarrow 1$; $P[s, 0] \leftarrow 0$
        for $k \leftarrow 1$ **to** $s - 1$ **do**
            $P[k, s-k] \leftarrow pP[k-1, s-k] + qP[k, s-k-1]$
    {Fill from below main diagonal to bottom right}
    for $s \leftarrow 1$ **to** n **do**
        for $k \leftarrow 0$ **to** $n - s$ **do**
            $P[s+k, n-k] \leftarrow pP[s+k-1, n-k] + qP[s+k, n-k-1]$
    return $P[n, n]$

Rendre la monnaie
B&B Section 8.2

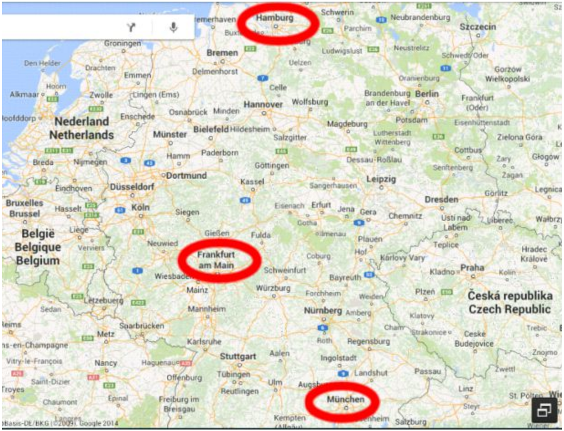
- Rappel : bien qu'efficace, l'approche vorace parfois ratait une solution
- L'approche programmation dynamique fonctionne, quelles que soient les valeurs des pièces :
  - démo du 15 novembre.
  - L'idée :
    $c[i,j] =$ nombre min de pièces pour rendre j en i dénominations
  - Alors :
    $$\underbrace{c[i,j] = \min(c[i-1,j], 1 + c[i, j - \text{dénom}[i]])}_{\text{suggère à nouveau de remplir par diagonales, de haut en bas}}$$

Principe d'optimalité

B&B Section 8.3

La programmation dynamique est à privilégier lorsque

- le problème à résoudre se décompose en sous-problèmes semblables
- ces sous-problèmes ont tendance à se chevaucher
- le principe d'optimalité s'applique : chaque sous-séquence d'une séquence de choix optimale est optimale



- chemin le plus court : oui
- chemin le plus rapide : non
- chemin simple le plus long : non, ex : graphe complet

<u>SAC À DOS</u>

DONNÉE: capacité $W \in \mathbb{R}^{\geq 0}$ et objets $1, 2, \ldots, n$ de poids $w_1, \ldots, w_n \in \mathbb{R}^{\geq 0}$ et de valeurs $v_1, \ldots, v_n \in \mathbb{R}^{\geq 0}$

CALCULER: objets de valeur maximale et de poids n'excédant pas W

- Rappel : bien qu'efficace, l'approche vorace ne parvenait à résoudre que SAC À DOS FRACTIONNAIRE
- L'approche programmation dynamique résoud SAC À DOS
Comment ?

Sac à dos
(suite)

- L'idée :
  $V[i,j] =$ valeur max avec objets $\{1, 2, \ldots, i\}$ et capacité $\leq j$
- On cherche : $V[n, W]$

- Alors :

Sac à dos
(suite)

- L'idée :
  $V[i, j] =$ valeur max avec objets $\{1, 2, \ldots, i\}$ et capacité $\leq j$
- On cherche : $V[n, W]$

- Alors :
  $$\underbrace{V[i, j] = \max(V[i-1, j], v_i + V[i-1, j-w_i])}_{\text{suffit donc de remplir ligne par ligne, de haut en bas}}$$
- L'algorithme détaillé coûtera $\Theta(nW)$ opérations (accès au tableau, sommes, comparaisons)

PLUS COURTS CHEMINS

DONNÉE: graphe (N, A) avec longueurs non négatives (ou ∞) aux arcs
CALCULER: chemins les plus courts de chaque sommet i à chaque sommet j

- Rappel : Dijkstra (vorace) calculait en $\Theta(|N|^2)$ les distances d'un sommet fixé à tous les autres,
  donc résoud PLUS COURTS CHEMINS en $\Theta(|N|^3)$
- Floyd (programmation dynamique) fournit une solution alternative
  Comment ?

Plus courts chemins
(suite)

- L'idée :
  $D_k[i,j] =$ longueur du pcc de i à j restreint aux sommets $\leq k$
- On cherche : les n^2 entrées de D_n

- Alors $\forall i,j$: $D_k[i,j] = \min(\ D_{k-1}[i,j]\ ,$

Plus courts chemins
(suite)

- L'idée :
  $D_k[i,j] =$ longueur du pcc de i à j restreint aux sommets $\leq k$
- On cherche : les n^2 entrées de D_n

- Alors $\forall i,j$: $D_k[i,j] = \min(\ D_{k-1}[i,j]\ ,\ D_{k-1}[i,k] + D_{k-1}[k,j]\ )$

- L'algo :

Plus courts chemins
(suite)

- L'idée :
  $D_k[i,j] =$ longueur du pcc de i à j restreint aux sommets $\leq k$
- On cherche : les n^2 entrées de D_n

- Alors $\forall i, j$: $D_k[i,j] = \min(\ D_{k-1}[i,j]\ ,\ D_{k-1}[i,k] + D_{k-1}[k,j]\ )$

- L'algo :
  - remplir D_1, puis D_2, puis D_3, puis $\cdots$, puis D_n
  - coup de bol : un seul tableau peut stocker $D_1, D_2, \ldots, D_n$

Plus courts chemins
(suite)

```
function Floyd(L[1..n,1..n]): array [1..n,1..n]
    array D[1..n,1..n]
    D ← L
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min(D[i,j], D[i,k]+D[k,j])
    return D
```

Produit chaîné de matrices

B&B Section 8.6

ORDONNANCEMENT DE PRODUITS MATRICIELS

DONNÉE: matrices $M_1, M_2, \ldots, M_n$ de dimensions compatibles

CALCULER: l'ordre optimal dans lequel effectuer les produits pour obtenir $\prod_{i=1}^{n} M_i$

En classe.

```python
def matrices(D):
    m = len(D)-1
    T = [ [0]*m for i in range(m)]
    P = [[-1]*m for i in range(m)]

    for i in reversed(range(m)):
        for j in range(i+1, m):
            c, pos = float("inf"), -1

            for k in range(i, j):
                d = T[i][k] + T[k+1][j] + D[i]*D[k+1]*D[j+1]

                if (d < c):
                    c, pos = d, k

            T[i][j] = c
            P[i][j] = pos

    return P
```

Transformer en récursion...

B&B Section 8.7

Idée :

- Au lieu d'un tableau comme $T[i, j]$, une fonction récursive $fT[i, j]$
- Au lieu d'accéder à $T[i, j]$, appeler récursivement $fT[i, j]$

Pourquoi ?

- Pour remplir “top down” au lieu de “bottom up”

Le malheur ?

- Même inconvénient de recoupements abusifs d'exemplaires que les usages inefficaces de diviser-pour-régner !

...et employer les fonctions à mémoire

B&B Section 8.8

L'idée :

- Un tableau mtab qui mémorise les valeurs $fT[i,j]$ déjà calculées
- Avant de recalculer $fT[i,j]$, vérifier mtab[i.j]
- On récupère (presque) l'efficacité de la programmation dynamique

Exemple pour Ordonnancement de produits matriciels :

function *fm-mem*(i, j)
  if $i = j$ **then return** 0
  if $mtab[i,j] \geq 0$ **then return** $mtab[i,j]$
  $m \leftarrow \infty$
  for $k \leftarrow i$ **to** $j-1$ **do**
    $m \leftarrow \min(m, \textit{fm-mem}(i,k) + \textit{fm-mem}(k+1,j) + d[i-1]d[k]d[j])$
  $mtab[i,j] \leftarrow m$
  return m