L3-INFO Algorithmique et Modélisation

Université Joseph Fourier GRENOBLE UFR IM²AG

Quick – 7 avril 2015 – durée 1 h

Sont interdits : les documents, les ordinateurs, les téléphones (incluant smartphone, tablettes,... tout ce qui contient un dispositif électronique).
Seuls les dictionnaires papier pour les personnes de langue étrangère sont autorisés.

La clarté des raisonnements et la qualité de la rédaction interviendront pour une part importante dans l'appréciation des copies.

Exercice 1 : Complexité récursive

Pour un problème donné, un algorithme naïf a une complexité en $O(n^3)$. On a aussi trouvé deux solutions de type diviser pour régner :

Div1 Découper le problème de taille n en deux sous-problèmes de taille $n/2$ et les recombiner en temps $O(n^2)$.

Div2 Découper le problème en quatre sous-problèmes de taille $n/3$ et les recombiner en temps $O(n)$.

1. Quelle est la complexité de Div1 ?
   Réponse :
   La complexité C_n de Div1 se calcule par récurrence :
   $$C_n = 2C_{n/2} + O(n^2).$$
   D'après le master theorem*, la complexité de Div1 est $O(n^2)$.*
   Remarque : le temps pour recombiner domine les appels récursifs car sans prendre en compte le temps de reconstruction, la complexité serait $O(n)$.
2. Quelle est la complexité de Div2 ?
   Réponse :
   La complexité C_n de Div2 se calcule par récurrence :
   $$C_n = 4C_{n/3} + O(n).$$
   D'après le master theorem*, la complexité de Div2 est $O(n^{\log_3(4)})$ (la complexité des appels récursifs domine le temps de reconstruction).*
3. Quelle solution choisir : naïf, div1 ou div2 ?
   Réponse :
   $\log_3(4)$ est plus petit que $\log_2(4) = 2$. On a donc $n^{\log_3(4)} < n^2 < n^3$. La solution Div2 est donc asymptotiquement plus rapide que les deux autres.

Exercice 2 : Programmation dynamique

On se donne une matrice C de taille $n \times m$ (n et m sont deux entiers supérieurs ou égaux à 1). On chercher à calculer la quantité $D_{n,m}$, définie par récurrence par la formule suivante :

$$D_{i,j} = \begin{cases} i \text{ si } j = 0 \\ j \text{ si } i = 0 \\ \min(D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1} + C_{ij}) \text{ sinon} \end{cases}$$

1. Écrire un algorithme récursif qui prend en entrée une matrice C de taille $n \times m$ et calcule $D_{n,m}$. Quelle est sa complexité ?

   Réponse :

   On peut par exemple définir la fonction `D(C,i,j)` *suivante :*

   Entrées *: une matrice C, deux entiers i et j*
   si *i==0 ou j==0* **alors**
   Retourner i+j;
   sinon
   Retourner $\min(D(C, i-1, j) + 1, D(C, i, j-1) + 1, D(C, i-1, j-1 + C_{ij}))$.

   que l'on appelle `D(C,n,m)`.

   Soit $F_{i,j}$ la complexité de l'algorithme pour calculer $D_{i,j}$. $F_{i,j}$ vérifie la relation de récurrence suivante :

   $$F_{i,j} = \begin{cases} 1 & \text{si } i = 0 \text{ ou } j = 0 \\ F_{i-1,j} + F_{i,j-1} + F_{i-1,j-1} + O(1) & \text{sinon.} \end{cases}$$

   On vérifie par récurrence que $F_{i,j} \leq O(3^{i+j})$ et $F_{i,j} \geq O(2^i + 2^j)$. L'algorithme récursif a donc une complexité exponentielle en i et j (mais le calcul exact est difficile).

2. La relation de récurrence précédente se prête particulièrement bien à l'expression d'une implémentation par programmation dynamique. Écrire l'algorithme correspondant à cette solution. Quelle est la complexité de votre algorithme ?

   Réponse :

   Le calcul de D demande d'avoir calculé les valeurs pour $(i-1, j)$, $(i, j-1)$ et $(i-1, j-1)$. On peut donc remplir un tableau à deux dimensions que l'on parcourt pour i et j croissant.

   Créer une matrice D de taille $(n+1) \times (m+1)$;
   pour $i = 0$ *à* n **faire**
   $D(i, 0) \leftarrow i$
   pour $j = 0$ *à* $m - 1$ **faire**
   $D(0, j) \leftarrow j$
   pour $i = 1$ *à* n **faire**
   pour $j = 1$ *à* $m - 1$ **faire**
   $D(i, j) \leftarrow \min(D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1 + C_{ij}))$
   Retourner $D(n, m)$;

Exercice 3 : Plus courts chemins

On se donne un graphe orienté (S, A) ayant $|S| = n$ sommets et $|A| = m$ arcs. Soit $o \in S$ un sommet de ce graphe. On cherche à calculer l'ensemble des sommets pour lesquels il existe un chemin partant de o. Pour cela, on utilise la propriété suivante :

(P) il existe un chemin de o à j de longueur inférieure ou égale à k si :

- il existe un chemin de o à j de longueur inférieure ou égale à $k-1$;

ou

- il existe un arc $(i, j) \in A$ et un chemin de o à i de longueur inférieure ou égale à $k-1$.

On note $C_{j,k}$ une variable qui vaut *vrai* s'il existe un chemin de o à j de longueur inférieure ou égale à k et qui vaut faux sinon.

1. Formaliser une relation de récurrence sur $C_{j,k}$ utilisant la propriété (P).

   Réponse :

   La propriété se traduit directement par la relation de récurrence suivante :

$$C_{j,k} = \begin{cases} Vrai & \text{si } k = 0 \text{ et } j = o \\ Faux & \text{si } k = 0 \text{ et } j \neq o \\ C_{j,k-1} \text{ ou il existe } (i,j) \in A \text{ telle que } C_{i,k-1} = vrai & \text{sinon} \end{cases}$$

2. La relation de récurrence précédente se prête particulièrement bien à l'expression d'une implémentation par programmation dynamique. Écrire l'algorithme correspondant à cette solution. L'algorithme prendra en entrée un sommet o et l'ensemble des arcs A et rendra en sortie un vecteur C tel que $C_i = vrai$ s'il existe un chemin entre o et i et $C_i = faux$ sinon.

   Réponse :

   Tous les chemins du graphe ont une longueur inférieure ou égale à $n-1$. On peut donc appliquer la propriété (P) $n-1$ fois :

   Initialiser un vecteur C de taille n avec la valeur faux*;*
   $C_o := vrai$*;*
   pour $k = 1$ *à* $n-1$ **faire**
   pour ***chaque arc*** $(i, j) \in A$ **faire**
   $C_j \leftarrow C_j$ *ou* C_i
   Retourner C;

3. Calculer la complexité de votre algorithme en fonction de n et m.

   Réponse :

   Il y a deux boucles imbriquées. Le coût de l'algorithme est majoré par $n \times m$

4. On associe à chaque arc $(i, j) \in A$ un poids $P_{ij} \geq 0$ et on cherche à calculer un chemin de poids minimal entre deux sommets o et d. Écrire un algorithme utilisant la propriété (P) qui affiche un chemin de poids minimal de o à d.

   Réponse :

L'algorithme précédent s'adapte en replaçant la ligne "$C_j \leftarrow C_j$ ou C_i" par un test pour savoir si le nouveau chemin calculé est plus court que le précédent. On obtient l'algorithme suivant :

Initialiser un vecteur C de taille n avec la valeur $+\infty$;
Initialiser un vecteur $Pred$ de taille n avec la valeur -1.;
$C_o := 0$;
pour $k = 1$ *à* n **faire**
  pour ***chaque arête*** $(i, j) \in A$ **faire**
    si $C_i + P_{i,j} < C_j$ **alors**
      $C_j \leftarrow C_i + P_{i,j}$;
      $Pred(j) \leftarrow i$
si $C[d] == +\infty$ **alors**
  Imprimer "pas de chemin entre o et d".
sinon
  $i = d$. **tant que** $i \neq o$ **faire**
    Imprimer "le sommet i est à visiter après Pred[i]";
    $i \leftarrow Pred[i]$;