Université Clermont Auvergne
ISIMA 1A par apprentissage
Algorithmique

Florent Foucaud
2023-2024

TD3 - Complexité en temps, partie II

1 Diviser pour régner

La formule vient de la politique ou de la guerre : *Divide ut regnes*, Philippe II de Macédoine, père d'Alexandre Legrand ; *Divide et impera*, Machiavel.

En algorithmique, il s'agit de couper un problème algorithmique en plusieurs sous-problèmes, généralement en 3 étapes :

- DIVISER : le problème est divisé en sous-problèmes
- RÉGNER : les sous-problèmes sont plus faciles à résoudre
- COMBINER : les solutions des sous-problèmes sont combinées

Exemples classiques : PGCD, Exponentiation rapide, Tri par fusion, Karatsuba (multiplication rapide), Strassen (multiplication de matrices), Tours de Hanoï...

Exercice 1 (Recherche dichotomique récursive)**.**

Quelle est la complexité en temps $T(N)$ de l'algorithme de recherche dichotomique ci-dessous, appelé avec $m = 1$ et $n = N$? Pour simplifier, on supposera que $N = 2^k$ est une puissance de deux.

`rechercheDicho(`TAB, m, n, x`): calcul d'une position de` x `dans` TAB

`Entrée : Un tableau d'entiers` $TAB[1...N]$ `triés par ordre croissant, un entier` x`, deux positions` m `et` n

`Sortie :` 0 `si` $x \notin TAB$`, et une position de` x `dans` TAB `(entre` m `et` n`) sinon`

- `Si` $m == n$ `:`
  - ⋆ `Si` $TAB[m] == x$ `:`
    `Retourner` m
  - ⋆ `Sinon :`
    `Retourner` 0
- `Sinon :`
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ `Si` $TAB[k] < x$ `:`
    `Retourner rechercheDicho(`$TAB, k+1, n, x$`)`
  - ⋆ `Sinon :`
    `Retourner rechercheDicho(`TAB, m, k, x`)`

Exercice 2 (Diviser pour régner : le tri par fusion)**.**

Pour simplifier, on supposera que $N = 2^k$ est une puissance de deux.

1. Estimer la complexité de la sous-fonction `interClassement(`TAB_1, TAB_2`)` en fonction de N_1 et N_2.
2. Dessiner l'arbre des appels récursifs de `triFusion` pour un tableau de taille $N = 2^4$. Quelle est la complexité approximative de l'algorithme sur *chaque niveau* de l'arbre ?
3. En déduire une estimation informelle de la complexité globale de l'algorithme.
4. On va maintenant calculer plus précisément cette complexité. Exprimer la complexité $T(N)$ de `triFusion` en fonction de $T(N/2)$.
5. Appliquer la formule précédente à $T(N/2)$ pour substituer $T(N/2)$ dans la formule. Ainsi on exprime $T(N)$ en fonction de $T(N/4)$.
6. Faire encore une étape en exprimant $T(N)$ en fonction de $T(N/8)$.
7. En déduire $T(N)$ en fonction de $T(N/2^i)$ ($i$ème étape récursive).
8. On rappelle que $N = 2^k$, donc si $i = k$ on a $T(N/2^i) = T(1)$. En déduire l'expression de $T(N)$ uniquement en fonction de N.

`triFusion(`TAB, m, n`): tri du sous-tableau` $TAB[m...n]$ `d'entiers par ordre croissant`

`Entrée : Un tableau d'entiers` $TAB[1...N]$ `et deux positions` m, n `avec` $m \leq n$

`Sortie : Le sous-tableau` $TAB[m...n]$ `trié`

- `Si` $m < n$ `:`
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ TAB_1 = `triFusion(`TAB, m, k`)`
  - ⋆ TAB_2 = `triFusion(`$TAB, k+1, n$`)`
  - ⋆ $TAB[m...n]$ = `interClassement(`TAB_1, TAB_2`)`
- `Retourner` $TAB[m...n]$

interClassement(TAB_1, TAB_2):

Entrée : Deux tableaux d'entiers $TAB_1[1...N_1]$, $TAB_2[1...N_2]$ triés par ordre croissant

Sortie : L'union TAB de TAB_1 et TAB_2 triée par ordre croissant

- $i = 1$, $j = 1$, $k = 1$
- TAB est un tableau vide à $N_1 + N_2$ éléments
- TABant que $i \leq N_1$ et $j \leq N_2$:
  - Si $TAB_1[i] < TAB_2[j]$:
    $TAB[k] = TAB_1[i]$
    $i = i + 1$
  - Sinon :
    $TAB[k] = TAB_2[j]$
    $j = j + 1$
  - $k = k + 1$
- Tant que $i \leq N_1$:
  $TAB[k] = TAB_1[i]$
  $i = i + 1$
  $k = k + 1$
- Tant que $j \leq N_2$:
  $TAB[k] = TAB_2[j]$
  $j = j + 1$
  $k = k + 1$
- Retourner TAB

2 Programmation dynamique

La *programmation dynamique* est une technique algorithmique qui consiste à calculer la solution pour des sous-problèmes, et où la solution du sous-problème suivant dépend directement de un ou plusieurs sous-problèmes précédemment calculés.

Souvent, c'est le moyen de rendre itératif un algorithme récursif, en passant d'une complexité exponentielle à une complexité polynomiale.

Les sous-problèmes sont souvent stockés dans une tableau, ou bien, on peut aussi faire de la programmation dynamique sur des structures arborescentes ou des graphes orientés acycliques.

Exercice 3 (Suite de Fibonacci)**.**

On considère le programme récursif suivant, permettant d'approximer la taille de la $n$ème génération d'une population de lapins.

```
Fibo(n):
Entrée : un entier n
Sortie : la nème valeur de la suite de Fibonacci
  • Si n == 0 ou n == 1:
    Retourner 1
  • Sinon:
    Retourner Fibo(n − 1) + Fibo(n − 2)
```

1. Dessiner l'arbre des appels récursifs pour $Fibo(6)$. Intuitivement, quelle est la complexité du programme ?
2. L'algorithme récursif effectue le même calcul un grand nombre de fois. Pour calculer $Fibo(n)$ de façon plus efficace, on pourrait simplement stocker les valeurs calculées au fur et à mesure dans un tableau. Écrire un tel algorithme.

Exercice 4 (Rendu de monnaie)**.**

On a une liste de dénominations de pièces et billets, par exemple `L=[1,2,5,10,20,50,100,200,500]`. Étant donné un entier `n` de monnaie à rendre en euros, on veut savoir quelle est la meilleure combinaison de pièces/billets (on souhaite minimiser le nombre total de pièces/billets à rendre).

On pourrait proposer un algorithme glouton pour ce problème : on choisit la plus grande dénomination `d` qui ne dépasse pas `n`, et on continue avec `n-d` jusqu'à arriver à 0. Cependant, cet algorithme n'est pas optimal : par exemple pour `L=[5,10,20,25]` et `n=40`, cet algorithme va renvoyer 3 $(25 + 10 + 5)$ alors que l'optimum est 2 $(20 + 20)$. Il faut donc trouver une méthode plus exhaustive...

On note $s[i]$ la solution pour la somme i (plus petit nombre de pièces/billets dont la somme est i).

1. Exprimer $s[i]$ selon deux cas : (1) i est dans la liste L des dénominations, ou sinon (2), en fonction de $s[i - d]$ pour tous les d dans L et $d < i$.
2. En déduire un algorithme récursif `MonnaieRec(n,L)` pour calculer la valeur optimale.
3. Estimer la complexité de l'algorithme récursif en fonction de n et de la taille $|L|$ de L.
4. Le transformer en algorithme itératif `MonnaieProgDyn(n,L)` utlisant le principe de programmation dynamique.
5. Quelle est sa complexité en temps en fonction de n et de la taille $|L|$ de L ? Comme $N = n + |L|$ est la taille de l'entrée, quelle est la complexité en fonction de N ?
6. Quelle modification peut-on faire pour aussi calculer la combinaison optimale (pas juste la valeur) ?

Exercice 5 (Répartition de médicaments)**.**

Une ONG dispose d'un stock de denrées (médicaments) en quantité n. Quand l'ONG distribue une quantité i de denrées à un endroit, l'expérience montre que cela sauve $S[i]$ personnes. Le tableau S a été établi par des experts et mis à disposition à l'ONG. On cherche à calculer le nombre maximal de personnes que l'on pourra sauver avec le stock en le découpant en sous-parties. Par exemple, avec le tableau $S = [1, 3, 2]$ (c'est-à-dire $S[1] = 1$, $S[2] = 3$, $S[3] = 2$) et $n = 3$, on peut sauver :

- soit $1 + 1 + 1 = 3$ personnes en coupant le stock en 3 parties de taille 1 ;
- ou bien, 2 personnes en laissant le stock en une seule partie de taille 3 ;

- ou bien, $1 + 3 = 4$ personnes en coupant le stock en 1 partie de taille 1 et 1 partie de taille 2.

1. On note $m[i]$ la meilleure solution (nombre de personnes sauvées) pour un stock de taille i. Exprimer $m[i]$ récursivement, en fonction des valeurs $m[j]$ avec $j < i$, en utilisant aussi S lorsque nécessaire.
2. En déduire un algorithme récursif `RepartitionRec(n,S)` pour calculer la valeur optimale pour une valeur de n et un tableau $S[1...n]$ donné.
3. Quelle est la classe de complexité (approximative) en temps de cet algorithme ?
4. Transformer cet algorithme en algorithme itératif `RepartitionProgDyn(n,S)` en utilisant le principe de la programmation dynamique.
5. Quelle est sa complexité asymptotique en temps ?

Exercice 6 (Plus grand sous-tableau croissant)**.**

Étant donné un tableau de N entiers, on cherche le plus grand sous-tableau d'éléments croissants (pas forcément consécutifs).

Pour le résoudre on va construire les solutions possibles de façon incrémentale.

On suppose qu'on a un tableau d'entiers $TAB[1...N]$. Soit $d[i]$ la taille d'un plus long sous-tableau de TAB croissant dont le dernier élément est $TAB[i]$.

1. Exprimer $d[i]$ en fonction des valeurs $d[j]$ avec $j < i$.
2. En déduire un algorithme récursif pour calculer *la taille* d'un plus grand sous-tableau de TAB croissant.
3. L'algorithme récursif précédent est néanmoins exponentiel. Le transformer en algorithme itératif utilisant la technique de la programmation dynamique.
4. Quelle est sa complexité en temps ?
5. Quelle modification peut-on faire pour aussi calculer le plus long sous-tableau croissant (pas juste sa taille) ?