Reading Note: Chapter 4

Xitong Yang

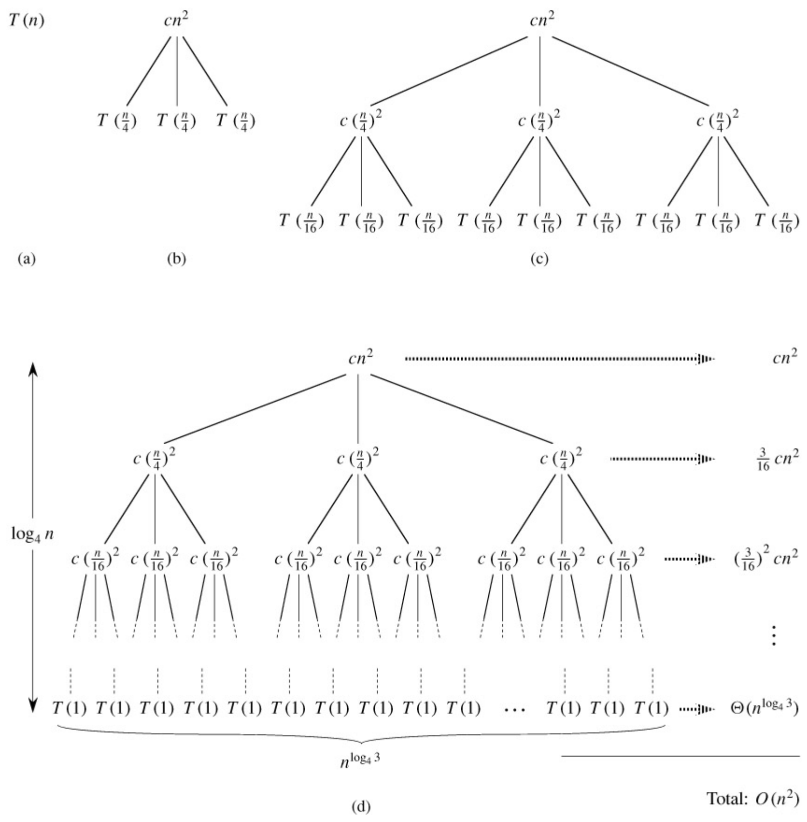
Dec.29 2015

1 Introduction

Chapter 4 mainly introduces a simple but powerful algorithm – **Divide-and-Conquer**. This algorithm is used when the problem can be divided into several similar subproblems and is always solved recursively. Three methods are mentioned to obtain the asymptotic bounds for solving recurrence: substitution, recursion-tree, master theorem.

2 Key Note

- In Divide-and-Conquer algorithm, we solve a problem recursively, applying three steps in each level of the recursion:
  - **Divide** the problem in to subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems recursively. If the subproblem sizes are small enough, just solve it in a straightforward manner.
  - **Combine** the solutions to the subproblems
- The substitution method comprise two steps: 1. *Guess* the form of the solution. 2. Use mathematical *induction* to find the constants and show that the solution works.
  - Revise the guess by subtracting a lower-order term when the inductive assumption is not strong enough to prove the bound.
- The recursion-tree method: is used to generate a good guess, then verified by the substitution method



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
$c\left(\frac{n}{16}\right)^2$
$T(1)$
(a)
(b)
(c)
(d)
$\log_4 n$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$

- The **Master Theorem**.

  For an algorithm that divides a problem of size n into a subproblems, each of size n/b, and the cost of dividing the problem and combining the results of the subproblems is $f(n)$, its running time can be described as

$$T(n) = a(T/b) + f(n)$$

  where $a \geq 1$ and $b \geq 1$ are constants and $f(n)$ is an asymptotically positive function. Then $T(n)$ has the following asymptotic bounds:

  1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.
  2. If $f(n) = O(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} lgn)$.
  3. If $f(n) = O(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

  Notes:

  – Intuitively, in each case we compare $f(n)$ with $n^{log_b a}$ to determine the solution.

  – For case 1(3), $f(n)$ must be *polynomially* smaller(bigger) than $n^{log_b a}$. Otherwise you cannot use the master method to solve the recurrence.

For example, $T(n) = 2T(n/2) + nlogn$ cannot use the master method because $nlogn$ is not polynomially larger than n.

3 Algorithms

- Maximum-subarray problem*
  - Divide-and-Conquer ($\Theta(nlogn)$): divide array to left and right, the maximum subarray lies in one of the places {left, right, crossing the midpoint}
  - Dynamic programming ($\Theta(n)$): loop i from 1 to n, record the maximum subarray ending at index i
- Strassen's algorithm for matrix multiplication (recursion: $T(n) = 7T(n/2) + \Theta(n^2)$, complexity: $\Theta(n^{lg7})$)

* Sample codes implemented in *Codes* folder