Analysis of Algorithm | Set 4 (Solving Recurrences)

In the previous post, we discussed analysis of loops. Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

```
For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn^2

          cn^2
        /      \
    T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

              cn^2
         /           \
     c(n^2)/16      c(n^2)/4
    /      \          /     \
 T(n/16)     T(n/8)  T(n/8)    T(n/4)
Breaking down further gives us following
               cn^2
          /            \
     c(n^2)/16          c(n^2)/4
     /      \            /      \
c(n^2)/256   c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \      /    \    /    \       /    \

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as (n^2)/(1 - 5/16) which is O(n^2)
```

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

```
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

There are following three cases:

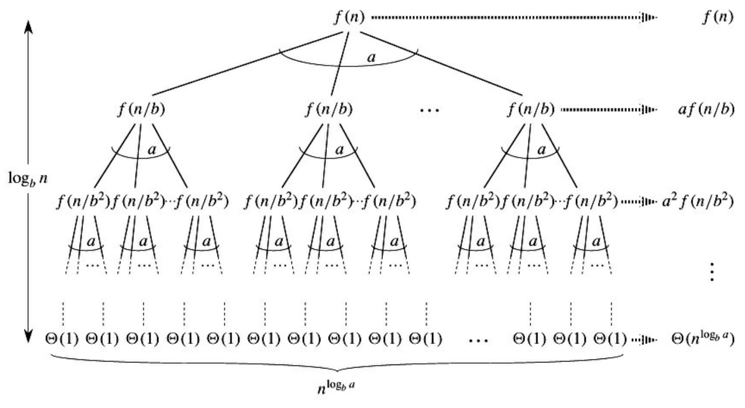
1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3.If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\text{Log}_b a]$ is also 1. So the solution is $\Theta(n \text{ Logn})$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $\text{Log}_b a$ is also 0. So the solution is $\Theta(\text{Logn})$

Notes:

1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\text{Logn}$ cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \text{Log}^k n)$
If $f(n) = \Theta(n^c \text{Log}^k n)$ for some constant $k >= 0$ and $c = \text{Log}_b a$, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$

Practice Problems and Solutions on Master Theorem.

Next – Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

References:

http://en.wikipedia.org/wiki/Master_theorem
MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Trending Content

- Check for Majority Element in a sorted array
- Enumeration of Binary Trees
- Puzzle 67 | Fit Triangle
- Java | Exception Handling | Question 3
- GATE | GATE-CS-2003 | Question 18
- Rearrange array in alternating positive & negative items with O(1) extra space | Set 1
- Setting up the environment in Java
- Swap Kth node from beginning with Kth node from end in a Linked List
- Point arbit pointer to greatest value right side node in a linked list
- k largest(or smallest) elements in an array | added Min Heap method

ProGeek Cup 1.0

Most Visited Posts

- Top 10 Algorithms and Data Structures for Competitive Programming
- Top 10 algorithms in Interview Questions
- How to begin with Competitive Programming?
- Step by Step Guide for Placement Preparation
- How to prepare for ACM-ICPC?
- Insertion Sort , Binary Search , QuickSort , MergeSort , HeapSort

Popular Categories

- Interview Experiences
- Advanced Data Structures
- Dynamic Programming
- Greedy Algorithms
- Backtracking
- Pattern Searching
- Divide & Conquer
- Geometric Algorithms
- Searching
- Sorting
- Analysis of Algorithms
- Mathematical Algorithms
- Randomized Algorithms
- Recursion
- Game Theory

Tags

Advanced Data Structure Amazon Aptitude Aptitude Arrays Bit Magic C C C++ C++ Quiz CPP-Library C Quiz Data Structures Data Structures DBMS Dynamic Programming Experienced GATE-CS-2012 GBlog Graph Hash Internship Interview Experiences Java java- Java Quiz Linked Lists Mathematical Matrix MCQ Microsoft number-digits Program Output Project Puzzles Python QA - Placement Quizzes QA - Placement Quizzes School Programming Searching Sorting STL Strings Technical Scripter Trees

Recent Comments