Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram
Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
An Institute of National Importance
COM 501 Advanced Data Structures and Algorithms - Lecture Notes

Recurrence Relations

In previous lectures we have discussed asymptotic analysis of algorithms and various properties associated with asymptotic notation. As many algorithms are recursive in nature, it is natural to analyze algorithms based on recurrence relations. Recurrence relation is a mathematical model that captures the underlying time-complexity of an algorithm. In this lecture, we shall look at three methods, namely, substitution method, recurrence tree method, and Master theorem to analyze recurrence relations. Solutions to recurrence relations yield the time-complexity of underlying algorithms.

1 Substitution method

Consider a computational problem P and an algorithm that solves P. Let $T(n)$ be the worst-case time complexity of the algorithm with n being the input size. Let us discuss few examples to appreciate how this method works. For searching and sorting, $T(n)$ denotes the number of comparisons incurred by an algorithm on an input size n.

1.1 Case studies: Searching and Sorting

Linear Search

`Input:` Array A, an element x
`Question:` Is $x \in A$
The idea behind linear search is to search the given element x linearly (sequentially) in the given array. A recursive approach to linear search first searches the given element in the first location, and if not found it recursively calls the linear search with the modified array without the first element. i.e., the problem size reduces by one in the subsequent calls. Let $T(n)$ be the number of comparisons (time) required for linear search on an array of size n. Note, when $n = 1$, $T(1) = 1$. Then, $T(n) = 1 + T(n-1) = 1 + \cdots + 1 + T(1)$ and $T(1) = 1$ Therefore, $T(n) = n - 1 + 1 = n$, i.e., $T(n) = \Theta(n)$.

Binary search

`Input:` Sorted array A of size n, an element x to be searched
`Question:` Is $x \in A$
Approach: Check whether $A[n/2] = x$. If $x > A[n/2]$, then prune the lower half of the array, $A[1, \ldots, n/2]$. Otherwise, prune the upper half of the array. Therefore, pruning happens at every iterations. After each iteration the problem size (array size under consideration) reduces by half. Recurrence relation is $T(n) = T(n/2) + 1$, where $T(n)$ is the time required for binary search in an array of size n. $T(n) = T(\frac{n}{2^k}) + 1 + \cdots + 1$

Since $T(1) = 1$, when $n = 2^k$, $T(n) = T(1) + k = 1 + \log_2(n)$.
$\log_2(n) \leq 1 + \log_2(n) \leq 2\log_2(n)$,$\forall\ n \geq 2$.
$T(n) = \Theta(\log_2(n))$.
Similar to binary search, the ternary search compares x with $A[n/3]$ and $A[2n/3]$ and the problem size reduces to $n/3$ for the next iteration. Therefore, the recurrence relation is $T(n) = T(n/3) + 2$, and $T(2) = 2$. Note that there are two comparisons done at each iteration and due to which additive factor '2' appears in $T(n)$.
$T(n) = T(n/3) + 2; \Rightarrow T(n/3) = T(n/9) + 2$
$\Rightarrow T(n) = T(n/(3^k)) + 2 + 2 + ... + 2$ (2 appears k times)
When $n = 3^k$, $T(n) = T(1) + 2 \times \log_3(n) = \Theta(\log_3(n))$
Further, we highlight that for k-way search, $T(n) = T(\frac{n}{k}) + k - 1$ where $T(k-1) = k-1$. Solving this, $T(n) = \Theta(\log_k(n))$.

Remarks:
1. It is important to highlight that in asymptotic sense, binary search, ternary search and k-way search (fixed k) are of the same complexity as $\log_2 n = \log_2 3 \cdot \log_3 n$ and $\log_2 n = \log_2 k \cdot \log_k n$. So, $\theta(\log_2 n) = \theta(\log_3 n) = \theta(\log_k n)$, for any fixed k.

2. In the above analysis of binary and ternary search, we assumed that $n = 2^k$ ($n = 3^k$). Is this a valid assumption? Will the above analysis hold good if $n \neq 2^k$. There is a simple fix to handle input samples which are not 2^k for any k. If the input array of size n is such that $n \neq 2^k$ for any k, then we augment the least number of dummy values so that $n = 2^k$ for some k. By doing so, the size of the modified input array is at most $2n$. For ternary search, the array size increases by at most $3n$. This approximation does not change our asymptotic analysis as the search time would be one more than the actual search time. That is, instead of $\log n$, we get $\log 2n$ which is $\log n + 1$. However, in asymptotic sense, it is still $\theta(\log n)$.

Sorting

To sort an array of n elements using find-max (returns maximum) as a black box.
Approach: Repeatedly find the maximum element and remove it from the array. The order in which the maximum elements are extracted is the sorted sequence. The recurrence for the above algorithm is,

$$T(n) = T(n-1) + n - 1 = T(n-2) + n - 2 + n - 1 = T(1) + 1 + \cdots + n - 1 = \frac{(n-1)n}{2}$$

$T(n) = \Theta(n^2)$

Merge Sort

Approach: Divide the array into two equal sub arrays and sort each sub array recursively. Do the sub-division operation recursively till the array size becomes one. Trivially, the problem size one is sorted and when the recursion bottoms out two sub problems of size one are combined to get a sorting sequence of size two, further, two sub problems of size two (each one is sorted) are combined to get a sorting sequence of size four, and so on. We shall see the detailed description of merge sort when we discuss divide and conquer paradigm. To combine two sorted arrays of size

$\frac{n}{2}$ each, we need $\frac{n}{2}+\frac{n}{2}-1 = n-1$ comparisons in the worst case. The recurrence for the merge sort is,

$T(n) = 2T(\frac{n}{2}) + n - 1 = 2[2T(\frac{n}{2^2}) + \frac{n}{2} - 1] + n - 1$

$\implies 2^k T(\frac{n}{2^k}) + n - 2^k + n - 2^{k-1} + \ldots + n - 1.$
When $n = 2^k$, $T(n) = 2^k T(1) + n + \cdots + n - [2^{k-1} + \ldots + 2^0]$
Note that $2^{k-1} + \ldots + 2^0 = \dfrac{2^{k-1+1} - 1}{2-1} = 2^k - 1 = n - 1$
Also, $T(1) = 0$ as there is no comparison required if $n = 1$. Therefore, $T(n) = n\log_2(n) - n + 1 = \Theta(n\log_2(n))$

Heap sort

This sorting is based on a data structure *max-heap* whose creation incur $O(n)$ time, which we shall discuss in detail at a later chapter. For now, let us assume, a max-heap is given; to sort an array, the approach is to delete the maximum element repeatedly which is at the root, and set right the heap to satisfy the max-heap property. This property maintenance incur $O(\log n)$ time for each iteration. The order in which the elements are deleted gives the sorted sequence. The number of comparisons needed for deleting an element is at most the height of the max-heap, which is $\log_2(n)$. Therefore the recurrence for heap sort is,
$T(n) = T(n-1) + \log_2(n)$, $T(1) = 0$
$\Rightarrow T(n) = T(n-2) + \log_2(n-1) + \log_2(n)$
By substituting further, $\Rightarrow T(n) = T(1) + \log 2 + \log 3 + \log 4 + ... + \log n$
$\Rightarrow T(n) = \log(2 \cdot 3 \cdot 4 \cdots n) \Rightarrow T(n) = \log(n!)$
$\Rightarrow \log(n!) \le n \log n$ as $n! \le n^n$ (Stirling's Approximation)
$\Rightarrow T(n) = O(n\log_2(n))$. Using Stirling's approximation, we can also show that $T(n) = \Omega(n\log n)$.

Finding Maximum

To find the maximum in an array of n elements, we first assume the first element is maximum and start comparing this local maximum with the rest of the elements linearly. Update the local maximum, if required. The base case, $T(2) = 1$ or $T(1) = 0$. Let $T(n)$ denote the worst case number of comparisons to find the maximum, then, $T(n) = T(n-1) + 1$ and the solution is $T(n) = n - 1$. Thus, $T(n) = \theta(n)$.

2 Change of Variable Technique

Problem: 1 $T(n) = 2T(\sqrt{n}) + 1$ and $T(1) = 1$

Introduce a change of variable by letting $n = 2^m$.
$\Rightarrow T(2^m) = 2 \times T(\sqrt{2^m}) + 1$
$\Rightarrow T(2^m) = 2 \times T(2^{m/2}) + 1$
Let us introduce another change by letting $S(m) = T(2^m)$
$\Rightarrow S(m) = 2 \times S(\frac{m}{2}) + 1$
$\Rightarrow S(m) = 2 \times (2 \times S(\frac{m}{4}) + 1) + 1$
$\Rightarrow S(m) = 2^2 \times S(\frac{m}{2^2}) + 2 + 1$

By substituting further,
$\Rightarrow S(m) = 2^k \times S(\frac{m}{2^k}) + 2^{k-1} + 2^{k-2} + ... + 2 + 1$
To simplify the expression, assume $m = 2^k$
$\Rightarrow S(\frac{m}{2^k}) = S(1) = T(2)$. Since $T(n)$ denote the number of comparisons, it has to be an integer always. Therefore, $T(2) = 2 \times T(\sqrt{2}) + 1$, which is approximately 3.
$\Rightarrow S(m) = 3 + 2^k - 1 \Rightarrow S(m) = m + 2$.
We now have, $S(m) = T(2^m) = m+2$. Thus, we get $T(n) = m+2$, Since $m = \log n$, $T(n) = \log n+2$
Therefore, $T(n) = \theta(\log n)$

Problem: 2 $T(n) = 2T(\sqrt{n}) + n$ and $T(1) = 1$

Let $n = 2^m \Rightarrow T(2^m) = 2 \times T(\sqrt{2^m}) + 2^m$
$\Rightarrow T(2^m) = 2 \times T(2^{m/2}) + 2^m$
let $S(m) = T(2^m) \Rightarrow S(m) = 2 \times S(m/2) + 2^m$
$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 2^{m/2}) + 2^m$
$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2 \cdot 2^{m/2} + 2^m$
By substituting further, we see that
$\Rightarrow S(m) = 2^k \times S(m/2^k)2^{m/2^k} + 2^{k-1}2^{m/2^{k-1}} + 2^{k-2}2^{m/2^{k-2}} + ... + 2 \cdot 2^{m/2} + 1 \cdot 2^m$

An easy upper bound for the above expression is;
$S(m) \leq 2^k \times S(m/2^k)2^m + 2^{k-1}2^m + 2^{k-2}2^m + ... + 2 \cdot 2^m + 1 \cdot 2^m$
$S(m) = 2^k \times S(m/2^k).2^m + 2^m[2^{k-1} + 2^{k-2} + ... + 2 + 1]$
To simplify the expression further, we assume $m = 2^k$
$S(m) \leq 2^k S(1)2^m + 2^m(2^k - 1)$
Since $S(1) = T(4)$, which is approximately, $T(4) = 4$.
$S(m) \leq 4m2^m + 2^m(m - 1)$
$S(m) = O(m2^m), T(2^m) = O(m2^m), T(n) = O(n \log n)$.

Is it true that $T(n) = \Omega(n \log n)$? Answer: NO

Reason: From the first term of the above expression, it is clear that $S(m) \geq 2^m$, therefore, $T(n) = \Omega(2^m) = \Omega(n)$. In the later section, we show that the solution to this recurrence is $\theta(2^m) = \theta(n)$

Problem: 3 $T(n) = 2T(\sqrt{n}) + \log n$ and $T(1) = 1$

let $n = 2^m$
$\Rightarrow T(2^m) = 2T(\sqrt{2^m}) + \log(2^m)$
$\Rightarrow T(2^m) = 2T(2^{m/2}) + m$
let $S(m) = T(2^m)$
$\Rightarrow S(m) = 2S(m/2) + m$
$\Rightarrow S(m) = 2(2S(m/4) + m/2) + m$
$\Rightarrow S(m) = 2^2 S(m/2^2) + m + m$
By substituting further,
$\Rightarrow S(m) = 2^k S(m/2^k) + m + m + ... + m + m$
let $m = 2^k \Rightarrow S(m/2^k) = S(1) = T(2) = 2$
$\Rightarrow S(m) = 2 + m(k - 1) + m$

$\Rightarrow S(m) = 2 + mk$
$\Rightarrow S(m) = 2 + m \log m$
$\Rightarrow S(m) = O(m \log m)$
$\Rightarrow S(m) = T(2^m) = T(n) = O(\log n \log \log n)$

3 Recursion Tree Method

While substitution method works well for many recurrence relations, it is not a suitable technique for recurrence relations that model divide and conquer paradigm based algorithms. Recursion Tree Method is a popular technique for solving such recurrence relations, in particular for solving unbalanced recurrence relations. For example, in case of modified merge Sort, to solve a problem of size n (to sort an array of size n), the problem is divided into two problems of size $n/3$ and $2n/3$ each. This is done recursively until the problem size becomes 1.

Consider the following recurrence relation.

1. $T(n) = 2T(n/2) + 1$
   Here the number of leaves $= 2^{\log n} = n$ and the sum of effort in each level except leaves is

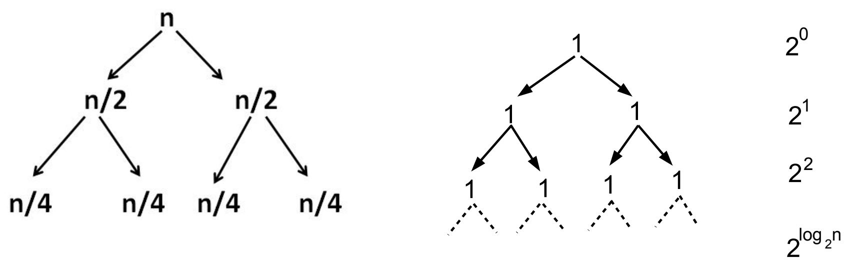

Fig. 1. *The Input size reduction tree* *The Corresponding Computation tree*

   $$\sum_{i=0}^{\log_2(n)-1} 2^i = 2^{\log_2(n)} - 1 = n - 1$$
   Therefore, the total time $= n + n - 1 = 2n - 1 = \Theta(n)$.

2. $T(n) = 2T(n/2) + 1$
   $T(1) = \log n$

   Solution: Note that $T(1)$ is $\log n$.
   Given $T(n) = 2T(n/2) + 1$
   $=2[2T(n/4) + 1] + 1 = 2^2T(n/2^2) + 2 + 1$
   $= 2^2[2T(n/2^3) + 1] + 2 + 1 = 2^3T(n/2^3) + 2^2 + 2 + 1 = 2^kT(n/2^k) + (2^{k-1} + 2^{k-2} + ... + 2 + 1)$
   We stop the recursion when $n/2^k = 1$.
   Therefore, $T(n) = 2^kT(1) + 2^k - 1 = 2^k \log n + 2^k - 1$
   $= 2^{\log_2 n} \log n + 2^{\log_2 n} - 1$
   $= n \log n + n - 1$

Clearly, $n \log n$ is the significant term. $T(n) = \theta(n \log n)$.

3. $T(n) = 3T(n/4) + cn^2$
 Note that the number of levels $= \log_4 n + 1$

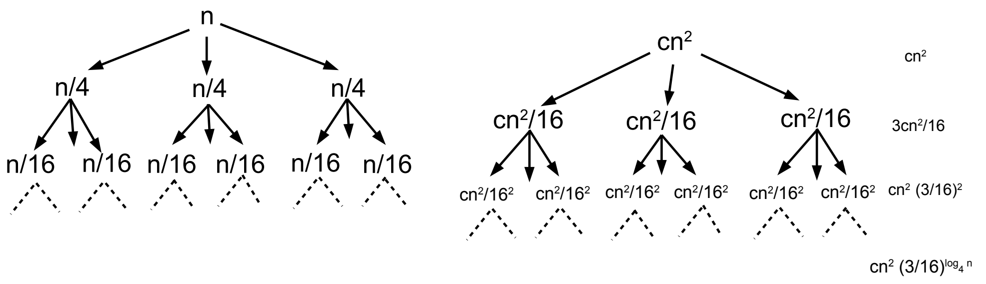

Fig. 2. *Input size reduction tree* *The Corresponding Computation tree*

Also the number of leaves $= 3^{\log_4 n} = n^{\log_4 3}$
The total cost taken is the sum of the cost spent at all leaves and the cost spent at each subdivision operation. Therefore, the total time taken is $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\log_4(n)-1} +$ number of leaves $\times T(1)$.
$T(n) = \sum_{i=0}^{\log_4(n)-1} cn^2(\frac{3}{16})^i + n^{\log_4(3)} \times T(1)$
$= \dfrac{\frac{3}{16}^{\log_4(n)} - 1}{\frac{3}{16} - 1} cn^2 + n^{\log_4(3)} \times T(1)$
$= \dfrac{1 - \frac{3}{16}^{\log_4(n)}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$
$= \dfrac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$
$= d \cdot cn^2 \cdot (1 - n^{\log_4(\frac{3}{16})}) + n^{\log_4(3)} \times T(1)$ where $d = \frac{1}{1-\frac{3}{16}}$
$= dcn^2(1 - \dfrac{n^{\log_4 3}}{n^2}) + n^{\log_4(3)} \times T(1)$
$= dcn^2 - dcn^{\log_4 3} + n^{\log_4(3)} \times T(1)$
$\leq dcn^2$. Therefore, $T(n) = O(n^2)$
Since the root of the computation tree contains cn^2, $T(n) = \Omega(n^2)$. Therefore, $T(n) = \theta(n^2)$

4. $T(n) = T(n/3) + T(2n/3) + O(n)$
 Note that the leaves are between the levels $\log_3 n$ and $\log_{3/2} n$
From the computation tree, it is clear that the maximum height is $\log_{3/2} n$. Therefore, the cost is at most $\log_{3/2} n \cdot cn = O(n \log n)$. Similarly, the minimum height is $\log_3 n$. Therefore, the cost is at least $\log_3 n \cdot cn = \Omega(n \log n)$. Thus, $T(n) = \theta(n \log n)$.

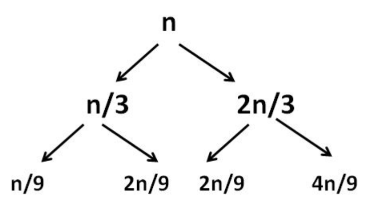

Fig. 3. *Recursion Tree*

Aliter: $T(n) \leq 2T(2n/3) + O(n)$ and $T(n) \geq 2T(n/3) + O(n)$. The solution to the former is $O(n^{\log_{\frac{3}{2}} 2})$ and the latter is $\Omega(n)$.

5. $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$
   Note that the leaves of the computation tree are found between levels $\log_{10} n$ and $\log_{\frac{10}{9}} n$
   Assume all the leaves are at level $\log_{10} n$
   Then $T(n) \geq n \log_{10} n \implies T(n) = \Omega(n \log n)$
   Assume all the leaves are at level $\log_{\frac{10}{9}} n$
   Then $T(n) \leq n \log_{\frac{10}{9}} n \implies T(n) = O(n \log n)$.
   Therefore, we conclude $T(n) = \theta(n \log n)$.

4 Guess method

One can guess the solution to recurrence relation $T(n)$ and verify the guess by simple substitution.

1. $T(n) = T(n/3) + T(2n/3) + O(n)$
   For $T(n) = T(n/3) + T(2n/3) + O(n)$, Guess $T(n) \leq dn \log(n)$.
   Substituting the guess, we get
   $T(n) = dn/3 \log n/3 + dn2/3 \log 2n/3 + cn$
   $T(n) = dn/3 \log n - dn/3 \log 3 + d2n/3 \log 2n - d2n/3 \log 3 + cn$
   $T(n) = dn \log n - dn \log 3 + d2n/3 + cn$
   Since $T(n)$ is at most $dn \log n$, we get
   $dn \log n - dn \log 3 + d2n/3 + cn \leq dn \log n \implies cn \leq dn(\log 3 - 2/3)$
   $c \leq d(\log 3 - 2/3)$ Choose c and d such that the inequality is respected.
   $\therefore T(n) \leq dn \log n = O(n \log n)$

2. $T(n) = 2T(n/2) + n$, $T(1) = 1$
   Solution: Guess $T(n) = O(n^2)$. $T(n) \leq 2cn^2/4 + n = cn^2/2 + n \leq cn^2$ (Possible)
   Therefore, $T(n) = O(n^2)$.

   Guess : $T(n) = O(n \log n)$
   $T(n) \leq 2cn/2 \log(n/2) + n = cn \log n - cn + n$
   $cn \log n - cn + n \leq cn \log n$ (Possible)

Therefore, $T(n) = O(n \log n)$

<u>Guess</u> : $T(n) = O(n)$
$T(n) \leq 2cn/2 + n$
$= cn + n$
$= cn + n \leq cn$ (not possible)
Therefore, $T(n) \neq O(n)$

3. $T(n) = 2T(n/2) + 1$
<u>Guess</u> : $T(n) = O(\log n)$
$T(n) \leq 2c \log n/2 + 1$
$= 2c \log n - 2c + 1$
$2c \log n - 2c + 1 \leq c \log n$ (Not possible)
Therefore, $T(n) \neq O(\log n)$. Note that, $T(n) \neq O(\log n)$, even if the guess is $T(n) \leq c \log n - d$, for some integer $d > 0$.

<u>Guess</u> : $T(n) = O(n)$
$T(n) \leq 2cn/2 + 1 = cn + 1$
$cn + 1 \leq cn$ (Not possible)
The guess that $T(n) = O(n)$ may not be an appropriate guess. A careful fine tuning on the above guess shows that $T(n)$ is indeed $O(n)$.

<u>Guess</u> : $T(n) = n - d$, for some constant $d > 0$
$T(n) \leq 2(n/2 - d) + 1 = n - 2d + 1$
$n - 2d + 1 \leq n - d$ (for $d \geq 1$))
Therefore, $T(n) = n - d = O(n)$

5 Master Theorem : A Ready Reckoner

We shall now look at a method 'master theorem' which is a 'cook book' for many well-known recurrence relations. It presents a framework and formulae using which solutions to many recurrence relations can be obtained very easily. Almost all recurrences of type $T(n) = aT(n/b) + f(n)$ can be solved easily by doing a simple check and identifying one of the three cases mentioned in the following theorem. By comparing $n^{\log_b a}$ (the number of leaves) with $f(n)$, one can decide upon the time complexity of the algorithm.

Master Theorem Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a non negative function, and let $T(n)$ be defined on the non-negative integers by the recurrence
$T(n) = aT(n/b) + f(n)$
where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:
Case 1: If $f(n)$=$O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ Then $T(n) = \theta(n^{\log_b a})$
Case 2: If $f(n)$=$\theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a} \log n)$
Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant ϵ>0, and if $af(n/b) \leq cf(n)$ for some constant c<1 and all sufficiently large n, then $T(n) = \theta(f(n))$.

In the next section, we shall solve recurrence relations by applying master theorem. Later, we discuss a proof of master's theorem and some technicalities to be understood before applying master theorem.

1. $T(n) = 9T(n/3) + n$

   $n^{\log_b a} = n^{\log_3 9} = n^2$
   $f(n) = n$
   Comparing $n^{\log_b a}$ and $f(n)$, we get $n = O(n^2)$
   Satisfies Case 1 of Master's Theorem, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$
2. $T(n) = T(2n/3) + 1$

   $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = f(n)$
   Comparing $n^{\log_b a}$ and $f(n)$; $n^{\log_b a} = \Theta(f(n))$
   Satisfies Case 2 of Master's Theorem, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$
3. $T(n) = 2T(n/2) + n$

   $n^{\log_b a} = n^{\log_2 2} = n^1 = n = f(n)$
   Comparing $n^{\log_b a}$ and $f(n)$; $n^{\log_b a} = \Theta(f(n))$
   Satisfies Case 2 of Master's Theorem, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$
4. $T(n) = 3T(n/4) + n \log n$

   $n^{\log_b a} = n^{\log_4 3}$
   $f(n) = n \log n$
   Comparing $n^{\log_b a}$ and $f(n)$, we see that Case 3 of Master's Theorem applies,
   Checking the regularity condition
   $af(n/b) \leq cf(n)$ (for some constant $c < 1$)
   $(3n/4) \log n/4 \leq cn \log n$
   $(3/4)n[\log n - \log 4] \leq (3/4)n \log n$ where $(c = 3/4)$, implies that the regularity condition is satisfied
   Thus, $T(n) = \Theta(f(n)) = \Theta(n \log n)$
5. $T(n) = 4T(n/2) + n$

   $n^{\log_b a} = n^{\log_2 4} = n^2$
   $f(n) = n$
   Comparing $n^{\log_b a}$ and $f(n)$
   $n = O(n^2)$
   Satisfies Case 1 of Master's Theorem
   This implies $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$
6. $T(n) = 4T(n/2) + n^2$

   $n^{\log_b a} = n^{\log_2 4} = n^2 = f(n)$
   Comparing $n^{\log_b a}$ and $f(n)$
   $n^{\log_b a} = \Theta(f(n))$
   Satisfies Case 2 of Master's Theorem
   This implies $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^2 \log n)$
7. $T(n) = 2T(\frac{n}{2}) + 2^n$.
   $g(n) = n^{\log_a b} = n^{\log_2 2} = n$ and $f(n) = 2^n$. Clearly, $f(n) = 2^n = \Omega(n^{\log_a b + \epsilon}) = \Omega(n^{\log_2 2 + \epsilon})$, for any $\epsilon > 0$. Therefore, case 3 of master theorem applies. Regularity condition check: $af(\frac{n}{b}) \leq$

$cf(n) = 2 \cdot 2^{\frac{n}{2}} \leq c \cdot 2^n$. Clearly, there exists $c < 1$ for $n \geq 4$, for example, $c = \frac{1}{2}$. Therefore, $T(n) = \theta(f(n)) = \theta(2^n)$.

8. $T(n) = 2T(\frac{n}{2}) + n3^n$.
   Since $n^{\log_a b} = n^{\log_2 2} = n$ and $f(n) = n3^n = \Omega(n^{\log_2 2+\epsilon})$, for any $\epsilon > 0$, case 3 of master's theorem is applicable. Regularity condition check: $af(\frac{n}{b}) \leq cf(n) = 2 \cdot \frac{n}{2} \cdot 3^{\frac{n}{2}} \leq c \cdot n3^n$. Clearly, there exists $c < 1$ for $n \geq 2$, for example, $c = \frac{1}{3}$. Therefore, $T(n) = \theta(f(n)) = \theta(n3^n)$.

Technicalities:

– From the above examples, it is clear that in each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \theta(n^{\log_b a})$.

– If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \theta(f(n))$.
– If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor which comes from the height of the tree, and the solution is T(n)=$\theta(n^{\log_b a} \log n)$.

– Also, in the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$.
– In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the regularity condition that $af(n/b) \leq cf(n)$ for some $c < 1$.

– Note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller.
– Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, we cannot use the master method to solve the recurrence.

– We shall next discuss recurrence relations that fall into the gaps between case 1 and 2, and case 2 and case 3. We also look at a recurrence relation that satisfies the case 3 of master theorem but fails to satisfy regularity condition.

1. $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$
   Here, $n^{\log_a b} = n^{\log_2 2} = n$. If we compare n and $f(n) = \frac{n}{\log n}$, then n is asymptotically larger than $\frac{n}{\log n}$ but not polynomially larger. That is $\frac{n}{\frac{n}{\log n}} \neq n^\epsilon, \epsilon > 0$. $\frac{n}{\frac{n}{\log n}} \neq \Omega(n^\epsilon)$, for any $\epsilon > 0$. Therefore, this recurrence relation falls into the gap between case 1 and 2 and hence the master theorem is not applicable. This can be solved using recurrence tree method which will be discussed later in this section.
   The next two recurrences, on the first look suggests that it falls into the gap between case 1 and 2, however, has a solution through case 1 of master's theorem.

2. $T(n) = 2T(\frac{n}{2}) + \frac{n}{n^4}$
   Here, $n^{\log_a b} = n^{\log_2 2} = n$. If we compare n and $f(n) = \frac{n}{n^4}$, then n is asymptotically larger than $\frac{n}{n^4}$ and also polynomially larger. That is $\frac{n}{\frac{n}{n^4}} = n^3, \epsilon = 3$. Therefore, this recurrence

relation satisfies case 1 of master theorem and the solution is $T(n) = \theta(n)$.

3. $T(n) = 2T(\frac{n}{2}) + \frac{n}{2^n}$
Here, $n^{\log_a b} = n^{\log_2 2} = n$. If we compare n and $f(n) = \frac{n}{2^n}$, then n is asymptotically larger than $\frac{n}{2^n}$ and also polynomially larger. That is $\frac{n}{\frac{n}{2^n}} = 2^n = \Omega(n^\epsilon)$, for any $\epsilon > 0$. Therefore, this recurrence relation satisfies case 1 of master theorem and the solution is $T(n) = \theta(n)$.

4. $T(n) = 2T(n/2) + n \log n$

   $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
   Comparing $n^{\log_b a} = n$ and $f(n) = n \log n$
   Does not satisfy either Case 1 or 2 or 3 of the Master's theorem
   Case 3 states that $f(n)$ should be polynomially larger but here it is asymptotically larger than $n^{\log_b a}$ only by a factor of $\log n$
   Note: If $f(n)$ is polynomially larger than $g(n)$ (in our discussion, $g(n) = n^{\log_b a}$), then $\frac{f(n)}{g(n)} = n^\epsilon, \epsilon > 0$. That is, $\frac{f(n)}{g(n)} = \Omega(n^\epsilon)$, for some $\epsilon > 0$. Note that in the above recurrence $n \log n$ is asymptotically larger than $n^{\log_b a}$ but not polynomially larger as per the above definition. I.e., $n = O(n \log n)$ whereas $\frac{n \log n}{n} \neq \Omega(n^\epsilon)$, for any $\epsilon > 0$

5. $T(n) = 4T(n/2) + n^2 \log n$

   $n^{\log_b a} = n^{\log_2 4} = n^2$
   Comparing $n^{\log_b a} = n^2$ and $f(n) = n^2 \log n$
   Does not satisfy either Case 1 or 2 or 3 of the Master's theorem
   Case 3 states that $f(n)$ should be polynomially larger but here it is asymptotically larger than $n^{\log_b a}$ by a factor of $\log n$
   If recurrence relations fall into the gap between Cases 1 and 2 or Case 2 and 3, Master theorem cannot be applied, and such recurrences can be solved using recurrence tree method.

6. $T(n) = 2T(\frac{n}{2}) + n^2 n^{(1+sin(n))}$
Since $sin(n)$ ranges from $[-1, 1]$, $f(n) = n^2 n^{(1+sin(n))}$ ranges from $[n^2, n^4]$. Since $f(n)$ is polynomially larger than $n^{\log_2 2} = n$, this recurrence satisfies case 3 of the master theorem. However, it fails to satisfy the regularity condition. For example, when $n = 270^\cdot$, $sin 270 = -1$ and $sin 135 = \frac{1}{\sqrt{2}}$. Clearly, the condition $af(\frac{n}{b}) \leq cf(n)$ fails. Consider, $2(\frac{n}{2})^2(\frac{n}{2})^{1+sin(\frac{n}{2})} \leq cn^2 n^{1+sin(n)}$. On simplification and further substituting $n = 270$, we get $\frac{1}{2}(\frac{270}{2})^{1+\frac{1}{\sqrt{2}}} \leq c$. Note that we must find a $c < 1$ satisfying the above expression to satisfy regularity condition. However, no such $c < 1$ exists. Similarly for n which are a multiple of 270. Therefore, the regularity condition fails and master theorem cannot be used to solve this recurrence. Here again, recurrence tree method is used to solve recurrences of this type.

1. Solve $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$

`Solution:` We shall compute the cost of recurrence tree as follows: the contribution from the first level is $\frac{n}{\log n}$, the second level is $\frac{n}{\log \frac{n}{2}}$, the third level is $\frac{n}{\log \frac{n}{4}}$, and so on. The total cost $T(n) = \frac{n}{\log n} + \frac{n}{\log n-1} + \frac{n}{\log n-2} + \ldots$. Clearly, the cost is lower bounded by $\frac{n}{\log n} + \ldots + \frac{n}{\log n}$ ($\log n$ times). This implies that $T(n) = \Omega(n)$. Further, the cost is upper bounded by $\frac{n}{\log n-h} + \ldots + \frac{n}{\log n-h}$, where h is the maximum height, $h = \log n$. $T(n) \leq (\frac{n}{\log n - h}) \log n$. Since $h = \log n$, on simplifying, we get, $T(n) \leq (\frac{n}{1 - \frac{h}{\log n}})$, which is $T(n) = O(n)$. Thus, $T(n) = \theta(n)$.

2. Solve $T(n) = 2T(\frac{n}{2}) + \frac{n}{2+sin(n)}$

The total cost of the tree, $T(n)$ is $\frac{n}{2+sin(n)} + \frac{n}{2+sin\frac{n}{2}} + \frac{n}{2+sin\frac{n}{4}} + \ldots$. Note that $2 + sinn \in [1, 3]$ and there are $\log n$ terms in the above expression. Thus, $T(n) \leq n \log n$ and $T(n) \geq \frac{n}{3} \log n$. Therefore, $T(n) = \theta(n \log n)$.

3. Solve $T(n) = 2T(\frac{n}{2}) + n^2 n^{(1+sin(n))}$

Since $sinn \in [-1, 1]$, $n^2 n^{(1+sin(n))} \in [n^2, n^4]$. Therefore, $T(n) = \Omega(n^2)$ and $T(n) = O(n^4)$.

Remarks

1. Consider the recurrence relation $T(n) = 16T(\frac{n}{4}) + n$. Clearly, case 1 of master's theorem applies and hence $T(n) = \theta(n^2)$. Let us get a deeper understanding of how $n^{\log_b a}$ dominates $f(n)$, $af(\frac{n}{b})$, and so on. For this recurrence, $f(n)$ is smaller than $n^{\log_b a}$ by $n^{\epsilon=1}$. Interestingly, the contribution from the subsequent levels is at most $n \cdot n^{\epsilon=1}$. That is, the cost of computation tree is
$T(n) = n + 4n + 4^2 n + \ldots + n4^{\log n - 1} + n^{\log_4 16}$
$= n[\frac{4^{\log_4 n} - 1}{4 - 1}] + n^{\log_4 16}$
$\leq cn \cdot n^{\epsilon=1} + n^{\log_4 16} = O(n^2)$
This shows that $n^{\log_4 16}$ is the dominant factor and the total contribution from all other levels is at most the number of leaves. This observation is proved in the next section as part of the proof.

2. Let $T(n) = 4T(\frac{n}{4}) + n$. This satisfies case 2 of master's theorem and hence, the solution is $\theta(n \log n)$. From the computation tree, it is easy to see that the number of levels is $\log_4 n$ and each level incurs n, thus $T(n) = \theta(n \log n)$. Moreover, the contribution from the leaves is just n which is same as $f(n) = n$. Since the contribution from each level cannot be overlooked in this case, the solution includes the number of levels and their contribution. This observation is revisited as part of the proof.

3. Consider $T(n) = 2T(\frac{n}{2}) + n^2$. In this case, the contribution from each level is $cf(n)$, where $c < 1$ and the contribution from the leaves is polynomially smaller than $f(n)$. This shows that, the solution is $f(n)$. This fact is established in the next section as part of the proof of the master's theorem.

4. Note that the regularity condition check demands a constant $c < 1$; if $c = 1$ or any fixed value then, the recurrence falls into case 2, if $c > 1$ is a constant and increases with polynomial growth $(c, c^2, c^3 ...)$ at each stage, then the recurrence falls into case 1 of master's theorem.

5.1 Proof of Master's Theorem

We shall next present the proof of master's theorem using two Lemmas. As mentioned earlier, the time complexity $T(n)$ consists of the sum of the cost of leaves and the cost at internal nodes. The cost at internal nodes is captured using the function $g(n)$ as defined in Lemma 2. Further, the bounds for $g(n)$ is discussed in Lemma 2 by comparing $g(n)$ and $n^{\log_b a}$

Lemma 1 Let $a \geq 1$ and $b > 1$, let $f(n)$ be a non-negative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

Then, $T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$
Proof:

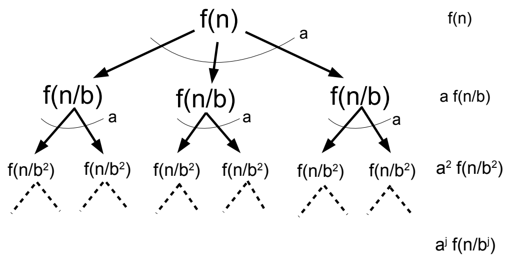

Fig. 4. *Recursion Tree*

From the recursion tree, it is clear that, at level i , there are a^i subproblems each of complexity $f(n/b^i)$.
The last level contains the set of leaves and the number of leaves is $a^{\log_b n} = n^{\log_b a}$
Therefore, the total cost is = number of leaves + $\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$, which is
$T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$
Lemma 2 Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a non-negative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

can be then bounded asymptotically for exact powers of b, as follows.

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$
2. If $f(n) = \theta(n^{\log_b a})$, then $g(n) = \theta(n^{\log_b a} f(n))$
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$, and $\forall n \geq b$, then $g(n) = \theta(f(n))$

Proof:
Here we analyze the recurrence relation, under the simplifying assumption that $T(n)$ is defined only on exact powers of $b > 1$, that is, for $n = 1, b, b^2 \ldots$ From Lemma 1,

$$T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

We now consider three cases to complete the proof.
Case 1: $f(n) = O(n^{\log_b a-\epsilon})$

Recall that $g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Since $f(n/b^j)$=$O((n/b^j)^{\log_b a-\epsilon})$, $g(n) = O(\sum\limits_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a-\epsilon})$

On rearranging, $g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j n^{\log_b a-\epsilon}/(b^{\log_b a-\epsilon})^j \implies g(n) = n^{\log_b a-\epsilon} \sum\limits_{j=0}^{\log_b n-1} a^j (b^\epsilon)^j/((b^{\log_b a})^j)$

$\implies g(n) = n^{\log_b a-\epsilon} \sum\limits_{j=0}^{\log_b n-1} a^j (b^\epsilon)^j/a^j \implies g(n) = n^{\log_b a-\epsilon} \sum\limits_{j=0}^{\log_b n-1} (b^\epsilon)^j$

$\implies g(n) = n^{\log_b a-\epsilon}[(b^\epsilon)^{\log_b n} - 1/(b^\epsilon - 1)] \implies g(n) = n^{\log_b a-\epsilon}[(n^\epsilon - 1/(b^\epsilon - 1)]$

$\implies$ Since $b > 1$, and ϵ >1 $\implies b^\epsilon - 1$ is constant(say c)

$g(n) \leq c.n^{\log_b a-\epsilon}.n^\epsilon \implies g(n) = O(n^{\log_b a})$

Finally, $T(n) = \theta(n^{\log_b a}) + O(n^{\log_b a})$ and $T(n) = \theta(n^{\log_b a})$

Case 2: $f(n) = \theta(n^{\log_b a})$

Recall, $g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Since $f(n/b^j)$=$\theta((n/b^j)^{\log_b a})$, $g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a}$

$g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j n^{\log_b a}/((b^j)^{\log_b a}) \implies g(n) = n^{\log_b a} \sum\limits_{j=0}^{\log_b n-1} a^j 1/((b^{\log_b a})^j)$

$\implies g(n) = n^{\log_b a} \sum\limits_{j=0}^{\log_b n-1} a^j/a^j \implies g(n) = n^{\log_b a} \sum\limits_{j=0}^{\log_b n-1} 1$

$\implies g(n) = n^{\log_b a}\theta(\log_b n) \implies g(n) = \theta(n^{\log_b a} \log_b n)$

Therefore, $T(n) = \theta(n^{\log_b a}) + \theta(n^{\log_b a} \log_b n)$ and $T(n) = \theta(n^{\log_b a} \log_b n)$

Case 3: $f(n) = \Omega(n^{\log_b a+\epsilon})$

Recall, $g(n) = \sum\limits_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Given $af(n/b) \leq cf(n) \implies f(n/b) \leq c/af(n)$

After j iterations, $f(n/b^j) \leq (c/a)^j f(n)$

$g(n) \leq \sum\limits_{j=0}^{\log_b n-1} a^j (c^j/a^j) f(n) \implies g(n) \leq \sum\limits_{j=0}^{\log_b n-1} c^j f(n)$

$\implies g(n) \leq \sum_{j=0}^{\infty} c^j f(n) \implies g(n) \leq f(n) \times 1/(1-c)$

$\implies \therefore g(n) = O(f(n))$

Since $g(n) = f(n) + \sum_{j=1}^{\log_b n-1} a^j f(n/b^j)$, $g(n) > f(n)$ and $g(n) = \Omega(f(n))$.
$\therefore g(n) = \theta(f(n))$

$T(n) = \theta(n^{\log_b a}) + \theta(f(n))$

Since, $f(n) = \Omega(n^{\log_b a+\epsilon})$, $T(n) = \theta(f(n))$

This completes a proof of the Master's theorem.

Not all recurrence relations can be solved using recurrence tree, masters theorem and substitution method. We here mention a method from difference equation to solve homogeneous recurrence relations. That is, recurrence relations which depends on r previous iterations (terms). Particularly, recurrence relations of the form $T(n) = c_1T(n-1) + c_2T(n-2) + \cdots + c_rT(n-r)$. In the next section, we shall discuss a method for solving well-known recurrences like 'Fibonacci series' using 'characteristic equation' based approach.

5.2 Solution using Characteristic Equation Approach

1. $a_n - 7a_{n-1} + 10a_{n-2} = 4^n$
   $T(n) - 7T(n-1) + 10T(n-2) = 4^n$
   Characteristic equation is $x^2 - 7x + 10 = 0$
   $(x-5)(x-2) = 0$; $x = 5, 2$
   Complementary function (C.F.) is $c_1 5^n + c_2 2^n$
   Solution is $a_n = c_1 5^n + c_2 2^n +$ Particular integral (P.I)
   For finding P.I, guess $a_n = d4^n$
   $d4^n - 7d4^{n-1} + 10d4^{n-2} = 4^n$
   $d - \frac{7}{4}d + \frac{10}{16}d = 1 \implies d = -8$
   Therefore solution is $T(n) = c_1 5^n + c_2 2^n - 8.4^n$

2. $a_n - 4a_{n-1} + 4a_{n-2} = 2^n$
   $T(n) - 4T(n-1) + 4T(n-2) = 2^n$
   Characteristic equation is $x^2 - 4x + 4 = 0$ and the roots are $x = 2, 2$
   Complementary function (C.F.) is $c_1.2^n + c_2.n.2^n$
   For P.I, note that the guesses $d.2^n$ and $d.n.2^n$ will not work as the roots and base of the exponent (non-homogeneous) term are same. Therefore we guess $d.n^2.2^n$
   $d.n^2.2^n - 4d(n-1)^2 2^{n-1} + d.(n-2)^2 2^{n-2} = 2^n$
   $d.n^2 - 2d(n^2 - 2n + 1) + d.(n^2 - 4n + 4) = 1$
   Simplifying we get $d = \frac{1}{2}$
   Therefore $T(n) = c_1 2^n + c_2.n.2^n + \dfrac{n^2.2^n}{2}$

Exercise

1. Solve the following
   $T(n) = 4T(\frac{n}{4}) + n^2$
   $T(n) = 4T(\frac{n}{2}) + 4^n$
   $T(n) = T(n+5) + n$
   $T(n) = 3T(\frac{n}{4}) + n$
   $T(n) = 2T(\frac{n}{2}) + 1$
   $T(n) = 2T(\frac{n}{2} + 17) + n$

2. Can Master method be applied to $T(n) = 8T(n/2) + n^3 \log n$. If not why?

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thank all of them.

References:
1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.