Recurrence Relations

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 ana-lyze 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.

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 $T(n) = T(n/2) + O(1)$, 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, $T(n) = \Theta(\log_k(n))$.
It is important to highlight that in asymptotic sense, binary search, ternary search and k-way search (fixed k) are having 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 fixed k.

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 least number of dummy values to make $n = 2^k$. By doing so, we are only increasing the size by 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.

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. The recurrence for the merge sort is,

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

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

Recurrence relation using change of variable technique

Problem: 1 $T(n) = 2 * T(\sqrt{n}) + 1$ 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 $S(m) = T(2^m)$
$\Rightarrow S(m) = 2 \times S(m/2) + 1$

$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 1) + 1$

$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2 + 1$

By substituting further,

$\Rightarrow S(m) = 2^k \times S(m/2^k) + 2^{k-1} + 2^{k-2} + ... + 2 + 1$

To simplify the expression, assume $m = 2^k$

$\Rightarrow S(m/2^k) = S(1) = T(2)$. 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 $T(n) = 2 * T(\sqrt{n}) + n$ 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 $S(m) = T(2^m) \Rightarrow S(m) = 2 \times S(m/2) + 2^m$

$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 2^{m/2}) + 2^m$

$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2.2^{m/2} + 2^m$

By substituting further, we see that

$\Rightarrow S(m) = 2^k \times S(m/2^k).2^{m/2^k} + 2^{k-1}.2^{m/2^{k-1}} + 2^{k-2}.2^{m/2^{k-2}} + ... + 2.2^{m/2} + 1.2^m$

An easy upper bound for the above expression is;

$S(m) \leq 2^k \times S(m/2^k).2^m + 2^{k-1}.2^m + 2^{k-2}.2^m + ... + 2.2^m + 1.2^m$

$S(m) = 2^k \times S(m/2^k).2^m + 2^m[2^{k-1} + 2^{k-2} + ... + 2 + 1]$

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, $T(4) = 4$.

$S(m) \leq 4m2^m + 2^m(m - 1)$

$S(m) = O(m.2^m), T(2^m) = O(m.2^m), T(n) = (n. \log n)$.

Is it true that $T(n) = \Omega(n. \log n)$?

From the first term of the above expression, it is clear that $S(m) \geq m.2^m$, therefore, $T(n) = \Omega(n. \log n)$

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

let $n = 2^m$

$\Rightarrow T(2^m) = 2 * T(\sqrt{2^m}) + \log(2^m)$

$\Rightarrow T(2^m) = 2 * T(2^{m/2}) + m$

let $S(m) = T(2^m)$

$\Rightarrow S(m) = 2 * S(m/2) + m$

$\Rightarrow S(m) = 2 * (2 * S(m/4) + m/2) + m$

$\Rightarrow S(m) = 2^2 * S(m/2^2) + m + m$

By substituting further,

$\Rightarrow S(m) = 2^k * S(m/2^k) + m + m + ... + m + m$

let $m = 2^k \Rightarrow S(m/2^k) = S(1) = T(2) = 2$

$\Rightarrow S(m) = 2 + m(k - 1) + m$

$\Rightarrow S(m) = 2 + m.k$

$\Rightarrow S(m) = 2 + m \log m$

$\Rightarrow S(m) = O(m \log m)$

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

2 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

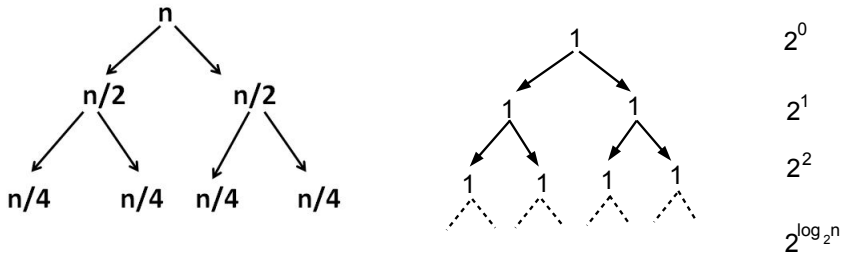

Fig. 1. *The Input size reduction tree* *The Corresponding Computation tree*

$\sum_{i=0}^{\log_2(n)-1} i = 2^{\log_2(n)} - 1 = n - 1$
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 $= \log_4 n + 1$

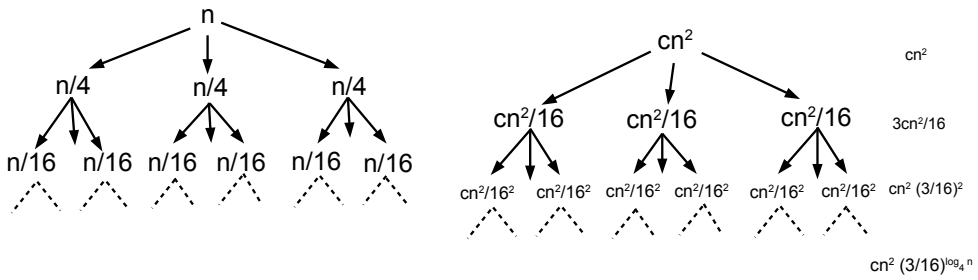

Fig. 2. *Input size reduction tree* *The Corresponding Computation tree*

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 $\times\, T(1)$.
$T(n) = \sum_{i=0}^{\log_4(n)-1} cn^2(\frac{3}{16})^i + n^{\log_4(3)} \times T(1)$
$= \frac{\frac{3}{16}^{\log_4(n)} - 1}{\frac{3}{16} - 1} cn^2 + n^{\log_4(3)} \times T(1)$
$= \frac{1 - \frac{3}{16}^{\log_4(n)}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$
$= \frac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$
$= d'cn^2 + n^{\log_4(3)} T(1)$ where the constant $d' = \frac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}}$
$= d'cn^2 + n^{\log_4(3)} T(1)$. 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, the cost incured in all levels except leaves is at most $\log_{3/2} n * cn = O(n \log n)$
The Number of leaves $= 2^{\log_{3/2} n} = n^{\log_{3/2} 2}$
The Total Time $T(n) = n^{\log_{3/2} 2} + cn(\log_{3/2} n - 1) \geq n \log n + n \log n - cn = \Omega(n \log n)$
$\therefore T(n) = \theta(n \log n)$

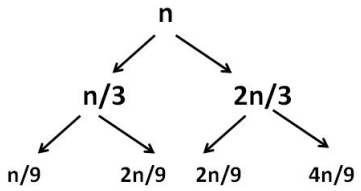

Fig. 3. *Recursion Tree*

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 $T(n) \leq n\log_{\frac{10}{9}}(n) \implies T(n) = O(n\log(n))$
   Solution using Guess method
   One can guess the solution to recurrence relation $T(n)$ and verify the guess by simple substitution.
   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
   $\therefore T(n) \leq dn\log n = O(n\log n)$
6. $T(n) = 2T(n/2) + n$, $T(1) = 1$
   Solution: Guess $T(n) = O(n^2)$. $T(n) \leq 2.cn^2/4 + n = cn^2/2 + n \leq cn^2$ (Possible)
   Therefore, $T(n) = O(n^2)$.

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

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

7. $T(n) = 2T(n/2) + 1$
   Guess : $T(n) = O(\log n)$

$T(n) \leq 2c \log n/2 + 1$
$= 2c \log n - 2c + 1$
$2c \log n - 2c + 1 \leq c \log n$ (Not possible)
Therefore, $T(n) \neq O(\log n)$

Guess : $T(n) = O(n)$
$T(n) \leq 2.cn/2 + 1 = cn + 1$
$cn + 1 \leq cn$ (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 $O(n)$.

Guess : $T(n) = n - d$
$T(n) \leq 2.(n/2 - d) + 1 = n - 2d + 1$
$n - 2d + 1 \leq n - d$ (for $d \geq 1$))
Therefore, $T(n) = n - d = O(n)$

3 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 $b \geq 1$ 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 $\epsilon \geq 0$ 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.

3.1 Deducing Time Complexity using Master's Theorem: Some Examples

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 $f(n)$
   $n = O(n^2)$

Satisfies Case 1 of Master's Theorem
That implies $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

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
That implies $T(n) = \Theta(n^{\log_b a}. \log n) = \Theta(\log n)$

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
That implies $T(n) = \Theta(n^{\log_b a}. \log n) = \Theta(n \log n)$

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 $f(n)$
Satisfies Case 3 of Master's Theorem
Checking the regularity condition
$a.f(n/b) < c.f(n)$ (for some constant $c < 1$)
$(3n/4) \log n/4 < c.n \log n$
$(3/4)n[\log n - \log 4] < (3/4)n \log n$ where (c=3/4)
That implies the regularity condition is satisfied
That implies $T(n) = \Theta(f(n)) = \Theta(n \log n)$

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
That 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
That implies $T(n) = \Theta(n^{\log_b a}. \log n) = \Theta(n^2 \log n)$

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

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$
Comparing $n^{\log_b a} = n$ and $f(n) = n \log n$

Doesn't 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 $g(n)$, then $\frac{f(n)}{g(n)} = n^{\epsilon}, \epsilon > 0$. Note that in the above recurrence $n \log n$ is asymptotically larger than $n^{\log_b a}$ but not polynomially larger. i.e., $n = O(n \log n)$ whereas $\frac{n}{n \log n} \neq n^{\epsilon}$, for any $\epsilon > 0$

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

   $n^{\log_b a} = n^{\log_2 4} = n^2$
   Comparing $n^{\log_b a} = n^2$ and $f(n) = n^2 \log n$
   Doesn't 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 case 1 and case 2 or case 2 and case 3, master theorem can not be applied and such recurrences can be solved using recurrence tree method.

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 $af(n/b) \leq cf(n)$.

- 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, you cannot use the master method to solve the recurrence.

- Here is a recurrence relation which satisfies the case 3 of master theorem but fails to satisfy regularity condition.

  1. $T(n) = 2T(\frac{n}{2}) + n^2(1 + sin(n))$
     Here $f(n) = n^2(1 + sin(n))$, $n^2(1 + sin(n)) \leq c.n^{\log_2 2+\epsilon}$ where $\epsilon = 1$. Therefore, case 3 of master's theorem applies. Now we shall check the regularity lemma.
     2.$(\frac{n}{2})^2(1 + sin(\frac{n}{2})) \leq c.n^2(1 + sin(n))$. Note that for every odd value m, $sin(\frac{2m+1}{2}\pi) = -1$

and $sin(\frac{2m+1}{4}\pi) = -\frac{1}{\sqrt{2}}$. When $n = \frac{2m+1}{2}$, it implies that $\frac{1}{2}.n^2(1 - \frac{1}{\sqrt{2}}) \leq c.n^2(1-1)$ and such a c does not exist.

2. $T(n) = 2T(\frac{n}{2}) + n^2.2^{-n}$
   Here $f(n) = n^2.2^{-n} = O(n^{\log_2 2+\epsilon})$, $\epsilon = 1$. For checking regularity lemma,
   $2.(\frac{n}{2})^2.2^{-\frac{n}{2}} \leq c.n^2.2^{-n}$ It follows that $\frac{1}{2}2^{-\frac{n}{2}} \leq c.2^{-n} \implies c \geq \frac{1}{2}.2^{\frac{n}{2}}$, which is not possible as $c < 1$.