# Lecture 4

- Divide & Conquer Approach
- Recurrences

# Designing Algorithms

- There are many ways to design algorithms.
- In insertion sort we used an incremental approach: i.e. having sorted the sub list of 1…… j-1 , we insert a single element x, at its proper place which yields again the sorted sub list of 1……. j elements.
- In the following section, we are going to study another approach of designing algorithms, known as Divide and Conquer approach.

# Divide and Conquer Approach

- Divide-and-conquer is a technique for designing algorithms that consists of breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), **and**
- Then combine these solutions to create a solution to the original problem.

The divide and conquer paradigm involves three steps at each level of recursion.

- ***Divide*** the problem into number of sub problems
- ***Conquer*** the sub problems by solving them recursively. If the sub problem sizes are small enough then solve them in straight forward manner.
- ***Combine*** the solutions to the sub problems into the solution for the original problem.

# Merge Sort

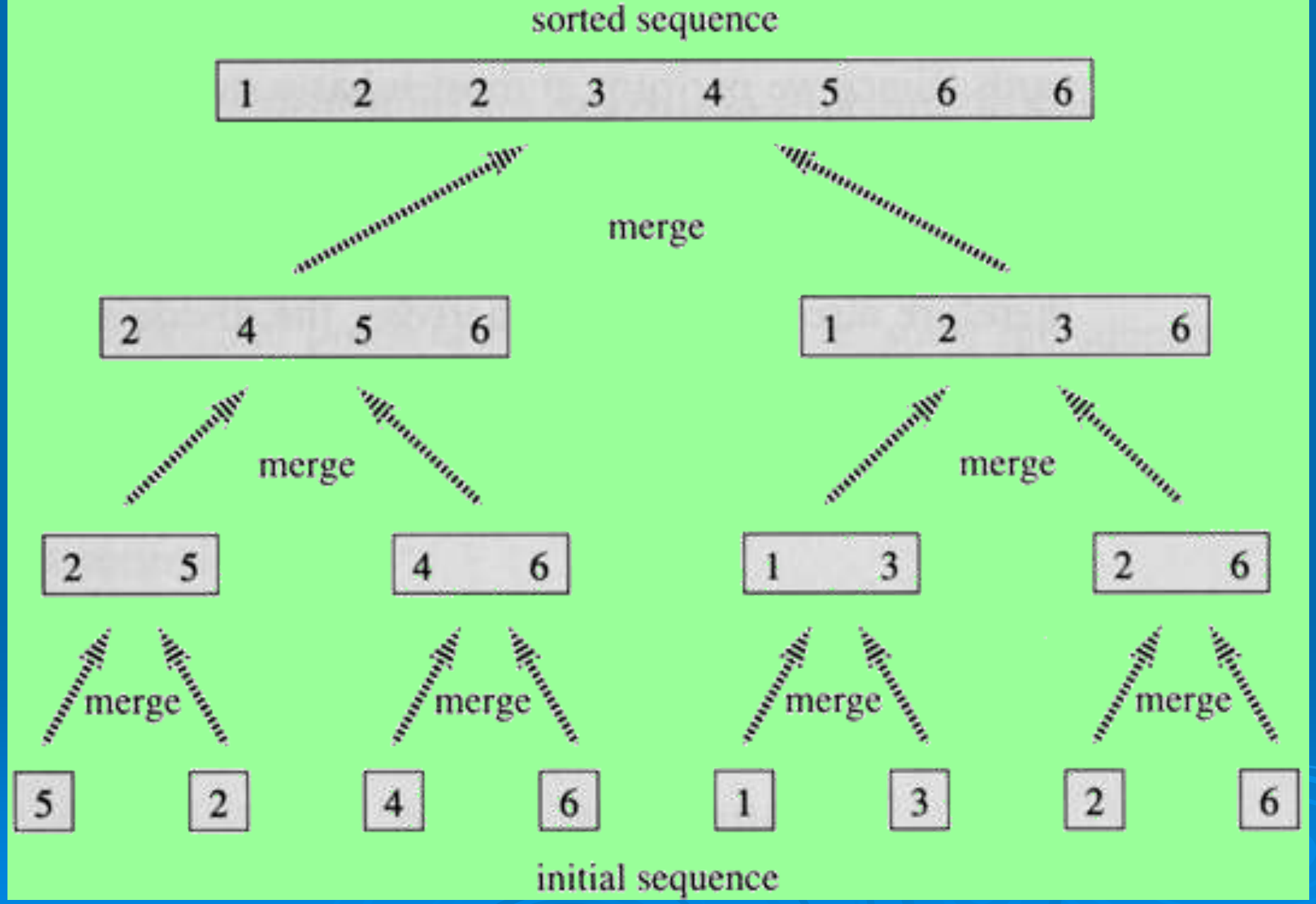
The **Merge sort** algorithm closely follows the divide and conquer paradigm. It works as follows.

- ***Divide*** : Divide the n - element sequence to be sorted into sub sequences each of size n / 2 elements.
- ***Conquer*** : Sort the two sub sequences recursively using merge sort.
- ***Combine*** : Merge the two sorted sub sequences to produce another sorted list.

- The key operation of the merge sort algorithm is the merging of two sub sequences in the “combine” step.

- To perform merging, we use Merge(A, left, mid, right); function, where A is an array of elements and left, right and mid representing leftmost, rightmost and center indices of an array respectively.

- The above function assumes that the sub array A[left…..mid] and A[mid+1 …… right] are in sorted order where n = right - left + 1


sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

# Analyzing Divide and Conquer Algorithms

- When an algorithm contains a recursive call to itself, its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of running time on smaller inputs or ***Recurrence*** is an equation or an inequality that describes the function in terms of its value on smaller inputs.
- We can use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

- A recurrence for the running time of a Divide-and-Conquer algorithm is based on three steps of basic paradigm.
- Let T(n) is the running time of problem of size n. if the problem size is small enough , say n <= c for some constant c, the solution takes constant time, which we write as Θ(1).
- Suppose our division of the sub problem yields **a** sub problems, each of size **1/b** of the size of original. If we take **D(n)** time to divide the problem into sub problems and **C(n)** time to combine the solutions to sub problems into the solution to the original problem, we get the following recurrence.

$$T(n) = \begin{cases} \Theta(1) & if n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise \end{cases}$$

- We will see later how to solve common recurrences of this form.

# Analysis of Merge Sort

- Although the merge sort algorithm works fine when the number of elements are not even, for our simplicity we assume that the original problem size is a power of 2. later we will see that this assumption does not effect the order of growth of the solution to recurrence.

- Merge sort on just one element takes constant time. When $n > 1$ , we break down the running time as follows.

- ***Divide*** : The Divide step just computes the middle of the sub array which takes constant time. Thus $D(n) = \Theta(1)$

- ***Conquer*** : We recursively solve two problems, each of size $n/2$ which contributes

  $T(n/2) + T(n/2) = 2T(n/2)$ to the running time

- ***Combine*** : Merge procedure on n-elements takes $\Theta(n)$ time, ***how ????***.

  So $C(n) = \Theta(n)$

# Analysis of Merge Sort

| Statement | Effort :- T(n) |
|---|---|

```
MergeSort(A, left, right)
{
    if (left < right)                          Θ(1)
    {
        mid = floor((left + right) / 2);       Θ(1)
        MergeSort(A, left, mid);               T(n/2)
        MergeSort(A, mid+1, right);            T(n/2)
        Merge(A, left, mid, right);            Θ(n)
    }
}
```

# Analysis of Merge Sort

```
MERGE( array A, int p, int q int r)
 1  int B[p..r]; int i ← k ← p; int j ← q + 1
 2  while (i ≤ q) and (j ≤ r)
 3  do if (A[i] ≤ A[j])
 4        then B[k++] ← A[i++]
 5        else B[k++] ← A[j++]
 6  while (i ≤ q)
 7  do B[k++] ← A[i++]
 8  while (j ≤ r)
 9  do B[k++] ← A[j++]
10  for i ← p to r
11  do A[i] ← B[i]
```

- We have D(n) + C(n) = Θ(1) + Θ(n). This sum is a linear function of n, i.e. Θ(n) . The worst case running time T(n) of merge sort:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(1) + \Theta(n) & n > 1 \end{cases}$$

**OR**

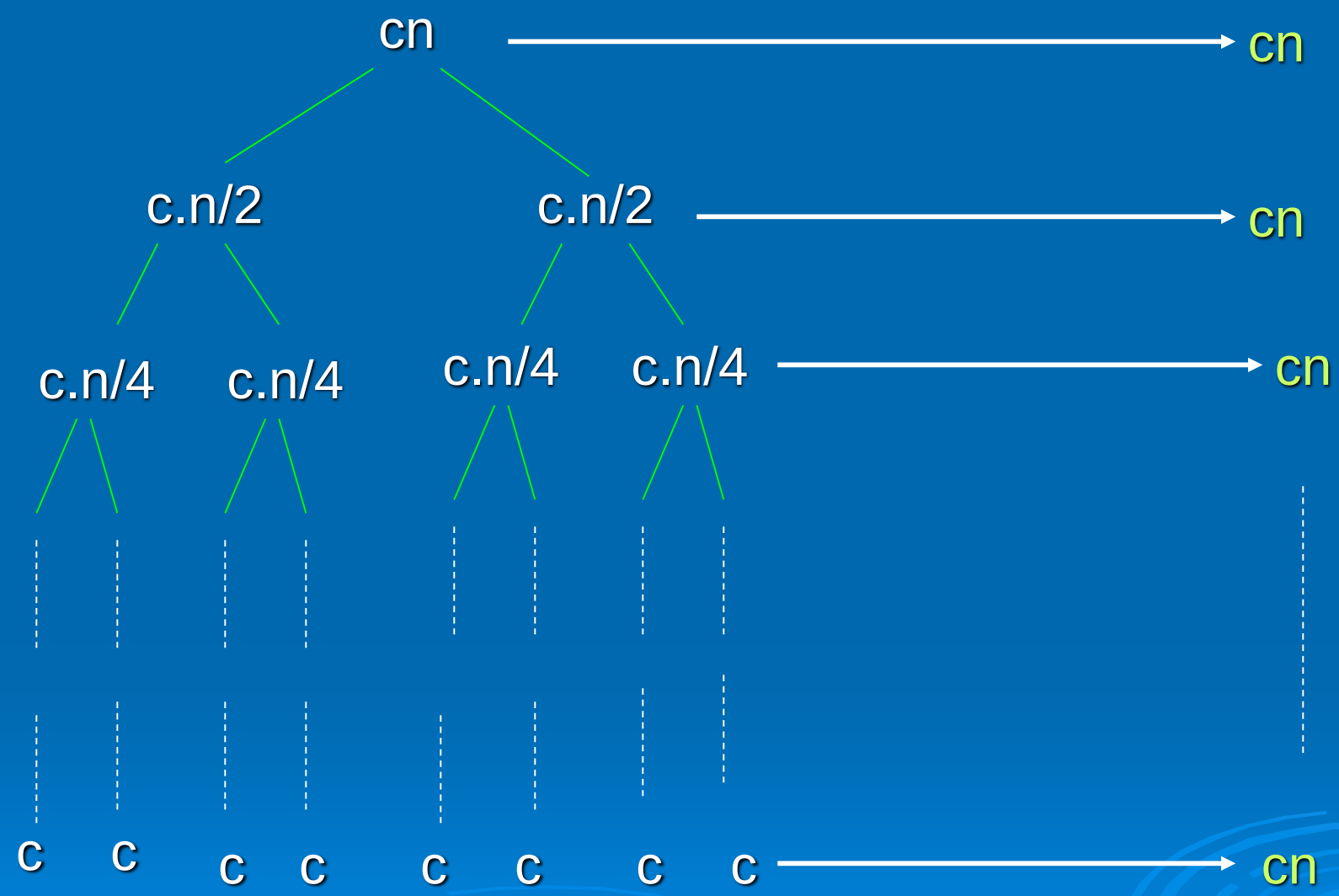
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

- We shall see in “Master Theorem” ( to be discussed later) that T(n) for Merge Sort is **Θ(n.lgn)** [ lgn means $\log_2 n$ ].
- Even we can predict and prove this time without using master theorem. Lets rewrite the above recurrence.

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- Where c represents the time required to solve the problem of size 1.

# Construction of recurrence tree for the recurrence T(n) = 2T(n/2) + cn



- Now we add the costs of each level:

  Top level : cn

  Next level from top: c.n/2 + c.n/2 = cn

  Next level : c.n/4 + c.n/4 + c.n/4 + c.n/4 = cn

- In general each level i (starting from 0) has $2^i$ nodes and each node contributing the cost $c(n/2^i)$.
- So that the total cost the ith level has the total cost $2^i c(n/2^i) = cn$
- At the bottom level, there are n nodes each contributing a cost of c, for total cost of cn.

- The total number of levels of above recursion tree are $\lg n + 1$.

- So to compute total cost represented by the recurrence, we simply add up all costs of all levels. There are $\lg n + 1$ levels and each is costing $cn$, for total cost of

$$cn(\lg n + 1) = cn\lg n + cn$$

ignoring lower order terms and the constant c we get the desired result of $\Theta(n.\lg n)$ .

- The worst case running time T(n) of merge sort could be described by the recurrence:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Whose solution was claimed to be T(n) = Θ (n.lg n)

- We will study three methods of solving recurrences I.e to obtain the asymptotic *O* or Θ bounds on the solution.

# Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

# Methods for Solving Recurrences

1. Substitution method
2. Recursion Tree (iteration) method
3. Master method

- The recurrence describing the worst case running time of Merge Sort is really :

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

the recurrence that arise from the running time of an algorithm generally have T(n) = Θ (1), for sufficiently small n. consequently we generally omit the statements of the boundary conditions of recurrences and assume that T(n) is constant for small n.

- When we state and solve recurrences, we often omit floors, ceilings and boundary conditions. We move ahead and later see whether or not they matter. They usually don't but it is important to know when they do so.

# Substitution Method

- The substitution method entails two steps
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works.
- This method is powerful but is obviously applied only in cases when it is easy to guess the form of the answer.
- The substitution method can be used to establish either upper or lower bounds on the recurrence.

- As an example, lets determine an upper bound on the recurrence

  $$T(n) = 2T(n/2) + n$$

- We ***guess*** that the solution is **$T(n) = O(n.\lg n)$**. It means that we have to prove that **$T(n) \leq cn.\lg n$** for an appropriate choice of the constant **$c > 0$**.

- Similarly from above we can say that

  **$T(n/2) \leq c.n/2.\lg n/2$**.

- We have

$$T(n) = 2T(n/2) + n$$
$$\leq 2.c.n/2.\ \lg n/2 + n$$
$$= c.n.\ \lg n/2 + n$$
$$= c.n\lg n - c.n\lg_2 2 + n$$
$$= c.n\lg n - c.n + n$$

ignoring small terms from above

$$\leq c.n\lg n$$

Where the last step holds as long as $c \geq 1$.

So from above it is proved that **$T(n) = O(n.\ \lg n\ )$**.

# Making a good guess

- Unfortunately there is no general way to have a good guess. Guessing a solution needs experience and occasionally creativity.
- We can use recursion trees for generating good guess, which we saw while analyzing Merge Sort.
- Another way to to make good guess is prove for loose upper and lower bounds on the recurrence and the reduce the range of uncertainty. We can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically correct solution

# Avoiding Pitfalls

- We can misinterpret the asymptotic notation if we say that our guess for **T(n) = 2T(n/2) + n** is **O(n)**. I.e. **T(n) ≤ cn** and then arguing that

$$T(n) = 2T(n/2) + n$$
$$\leq 2.c.n/2 + n$$
$$= cn + n \text{ or } n.(c + 1)$$
$$= O(n) \longleftarrow \textit{Wrong !!!}$$

*The error is that we haven't proved the exact form of our hypothesis I.e.* **T(n) ≤ cn**

# Recursion Tree Method

- As we found that it is sometime difficult to have a good guess in case of substitution method.
- Drawing out the recursion tree as we did in the analysis of Merge Sort is the straightforward way to have a good guess.
- In this method each node represents the cost of each sub problem somewhere in the set of recursive function invocations. We sum all the costs of sub problems at all levels of recursion to find the logical guess for our solution.

- This method is more useful for describing the running time of divide and conquer problems.

- This method is used to generate a very good guess which can be later verified by the substitution method. However you can use a recursion tree method as a direct proof of a solution to the recurrence.

# Master's Theorem

- The Master Method provides the bound for the recurrence of the form

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

  where $a >= 1$, $b > 1$ and $f(n)$ is a given function.

- The above recurrence describes the running time of an algorithm that divides the problem of size $n$ into $a$ sub problems, each of size $n/b$, where $a$ and $b$ are positive constants.

- Then a sub problems are solved recursively, each in time T(n/b).
- The cost of dividing problem and combining the results of the sub problems is described by the function f(n) i.e.

  f(n) = D(n) + C(n)
- This method requires the memorization of three cases as follows

# The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) \le cf(n) \text{ for large } n \end{cases} \quad \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

- You can go through Book (second edition) page no 73 for clear understanding of previous three cases.

# Master Method -- Examples

- Let $T(n) = 9T(n/3) + n$
  - a=9, b=3, f(n) = n
  - Now $n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$
  - So we need to subtract power from 2, so that it seems to be like f(n) which is = n
  - So let $\varepsilon$=1 and
  - f(n) = $O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon$=1, case 1 applies: i.e.

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

  - f(n) = $O(n^{2-1})$

  - Thus the solution is $T(n) = \Theta(n^2)$

# Example 2:

- Let $T(n) = T(2n/3) + 1$
  - a=1, b=3/2, f(n) = 1
  - Now $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
  - So we can apply case 2:
  - i.e if $f(n) = \Theta(n^{\log_{3/2} 1}) = \Theta(1)$,

  - Thus the solution is $T(n) = \Theta(\log_2 n)$

# Example 3

- Let $T(n) = 3T(n/4) + n.\log n$
  - $a=3, b=4, f(n) = n.\log n$
  - Now $n^{\log_b a} = n^{\log_4 3} = n^{0.8}$
  - So we need to add some factor with power (0.8), so that it seems to be like f(n) which is $= n.\log n$ , which corresponds to third case of our theorem.
  - So let $\varepsilon=0.2$ and we have $n^{0.8+0.2} = n^1$ ( near to f(n) )
  - Now we have to fulfill the second part of this case i.e.
  - $af(n/b) \leq c.f(n)$ for $c < 1$
  - $3f(n/4) \leq c.\ n.\ \log n$
  - $3.\ n/4.\ \log (n/4) \leq c.\ n.\ \log n$ let $c = ¾ < 1$
  - $3/4.\ n.\ \log (n/4) \leq 3/4.\ n.\ \log n$
  - Above inequality is proved so we have $T(n) = \Theta(n.\log n)$

# Assignment # 2

- Use the Master method to give tight asymptotic bounds for the following recurrences

  $T(n) = 4T(n/2) + n$

  $T(n) = 4T(n/2) + n^2$

  $T(n) = 4T(n/2) + n^3$

  $T(n) = 2T(n/4) + n^{1/2}$

***Due Next Week, same day, same time***