Unit: 1

Foundation Of Algorithm Analysis

⊙ **Algorithm** is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce value(s) as output.

The five key properties of an algorithm are:

① Input: An algorithm should have well-defined inputs. They are data given to the algorithm for processing. (≥ one)

② Output: An algorithm must produce at least one output which is the result of the algorithm's computations or processing.

③ Definiteness: Every step of the algorithm must be precisely defined i.e. instructions should be clear and unambiguous.

④ Finiteness: An algorithm must always terminate after a finite number of steps.

⑤ Effectiveness: The operations of the algorithm should be feasible and should not require infinite computation.

⑥ Correctness

⊙ **RAM Model**

Random Access Machine model is a theoritical model used in computer science to analyze and measure the efficiency of algorithms. In this model we count:

i) Basic operation (+, -, *) as 1 step

ii) Memory reference (read & write) as 1 step

iii) loops, function calls are not basic operation. Hence not counted.

⊙ Time and Space complexity

Time complexity measures the amount of time an algorithm takes to complete as a function of the size of its input.

Space complexity measures the amount of memory an algorithm requires to run as a function of the input size. The time complexity of an algorithm is commonly expressed using asymptotic notations.

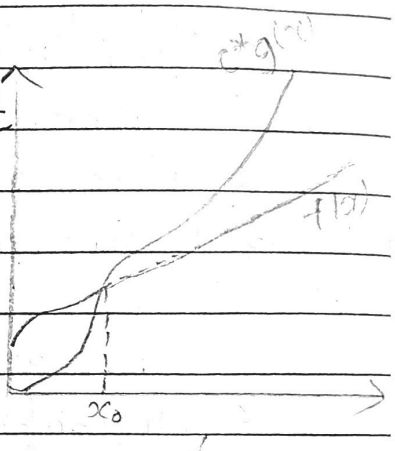
⊙ Big O - O(n)

It describes the upper bound or the worst case scenario of an algorithm's growth rate. Here,

$f(x) = O\,g(x)$

iff $f(x) \leq c * g(x)$ $\forall\ x \geq x_0$

where, x_0 & c are constant.



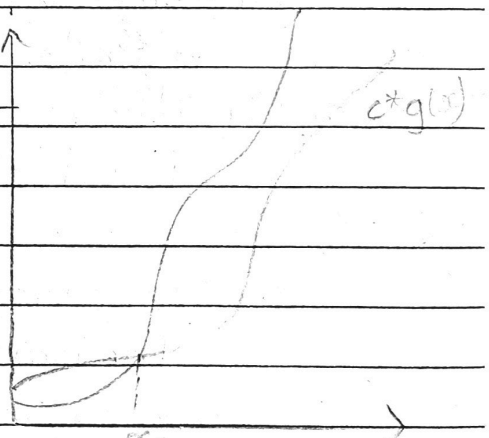
⊙ ~~Big Theta - Θ(n)~~ Big Omega - Ω(n)

It describes the lower bound or the best case scenario of an algorithm's growth rate. Here,

$f(x) = \Omega\, g(x)$ iff

$f(x) \geq c * g(x)$ $\forall\ x \geq x_0$

where x_0 & c are constant.



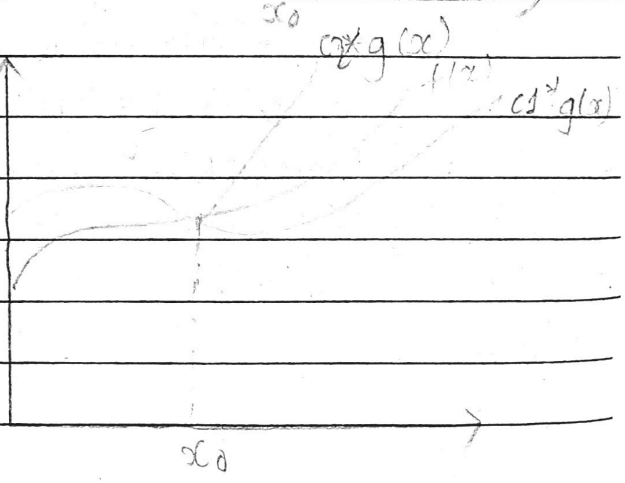
⊙ Big Theta - Θ(n)

It describes the tight bound.

Here, $f(x) = \Theta\, g(x)$ iff

$c1 * g(x) \leq f(x) \leq c2 * g(x)$ $\forall x$ $x >= x_0$

where, $c1$, $c2$ & x_0 are constant.



Solving Recurrences:

The process of finding solution of given recurrance relation in terms of big oh notation is called solving recurrences. Some of the methods are:-

1. Iteration Method
2. Recursion Tree Method
3. Substitution Method
4. Master Method

① **Iteration method:** Here we expand the given relation until the boundary is not met.

Example: $T(n) = 2T(n/2) + 1$ when $n > 1$

$T(n) = 1$

Solution: $T(n) = 2T(n/2) + 1$

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

$= 2^2 T(n/2^2) + 2 + 1$

$= 2^2\{2T(n/2^3) + 1\} + 2 + 1$

$= 2^3 T(n/2^3) + 2^2 + 2 + 1$

$\vdots$

$= 2^k T(n/2^k) + 2^{k-1} + \cdots + 4 + 2 + 1$

For simplicity assume: $n/2^k = 1 \Rightarrow n = 2^k$

$\Rightarrow \log n = k \log 2$

$\Rightarrow k = \log n$ (since $\log 2 = 1$)

Now, $T(n) = 2^k T(n/2^k) + 2^{k-1} + \cdots + 4 + 2 + 1$

$= 2^k T(1) + 2^{k-1} + \cdots + 4 + 2 + 1$

$= (2^{k+1} - 1)/(2-1)$ $\quad \{S = a(r^n - 1)/(r-1)\}$

$= 2^{k+1} - 1$

$= 2 \cdot 2^k - 1$

$= 2n - 1$

$\therefore T(n) = O(n)$

① Recursion Tree : It is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

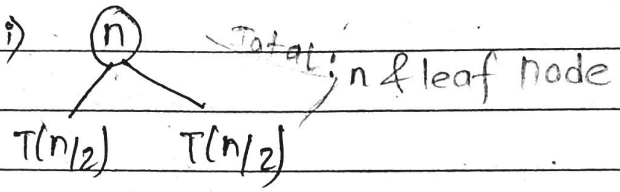
Example:

① For merge sort / best case of quick sort

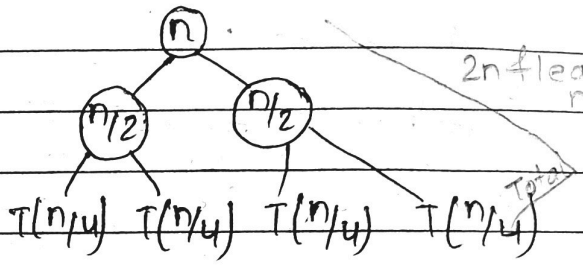
$T(n) = 2T(n/2) + n$ where $n > 1$

$= 1$ when $n = 1$

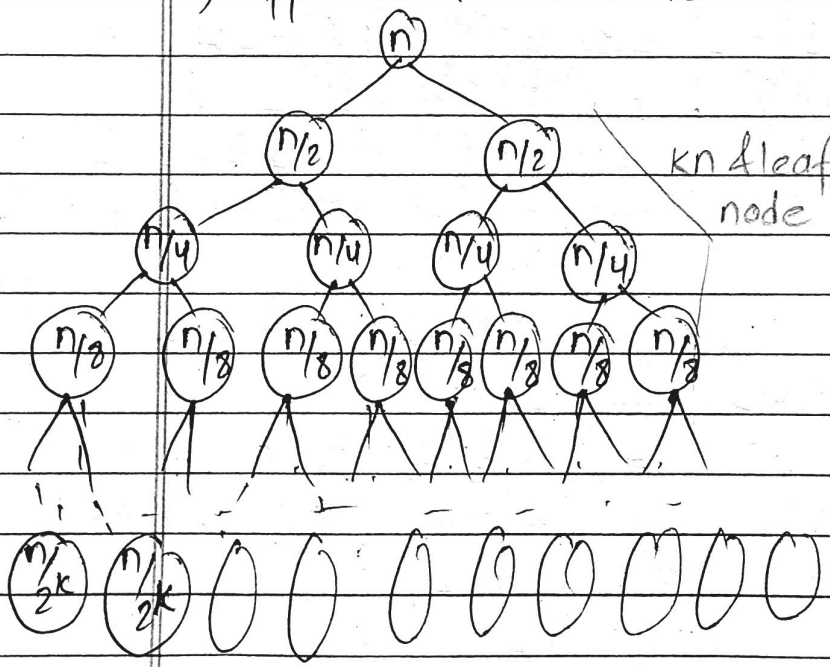
i)



ii)



iii) Suppose at k^{th} iteration:

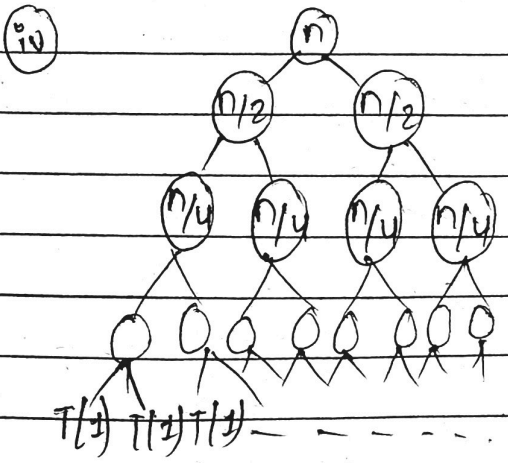


for simplicity,

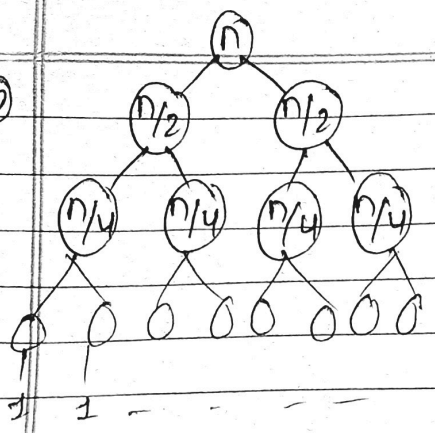
$\frac{n}{2^k} = 1 \Rightarrow 2^k = n$

$\Rightarrow k = \log_2 n$

iv)



(v)



Here, Total steps:

$T(n) = n + n + n + \dots n$

$= (k+1)\, n$

$= (\log_2 n + 1)\, n$

$= n \log_2 n + n$

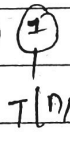
$\leq 2 * n \log_2 n$

$\therefore T(n) = O(n \log_2 n)$

(2) Binary Search

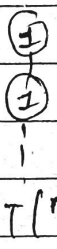
$T(n) = T(n/2) + 1$ when $n > 1$

$= 1$ when $n = 1$

(i) 

(ii) 

(iii) 

(iv) 

(v) 

(vi) 

⊙ for simplicity,

$\frac{n}{2^k} = 1$

$\Rightarrow 2^k = n$

$\Rightarrow k = \log_2 n$

Here, $T(n) = 1 + 1 + 1 + 1 + \dots + 1$

$= (k+1)$

$= \log_2 n + 1$

$\leq 2 \log_2 n$

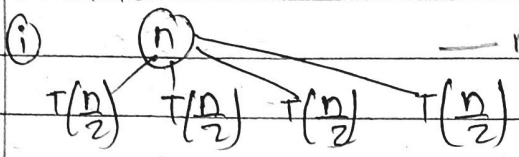
$\therefore T(n) = O(\log_2 n)$

③ $T(n) = 4T(n/2) + n$ when $n > 1$

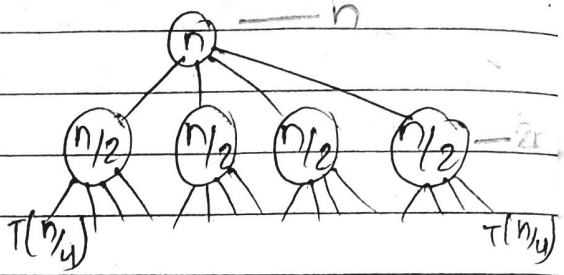
$= 1$ when $n = 1$

⇒ Solution:

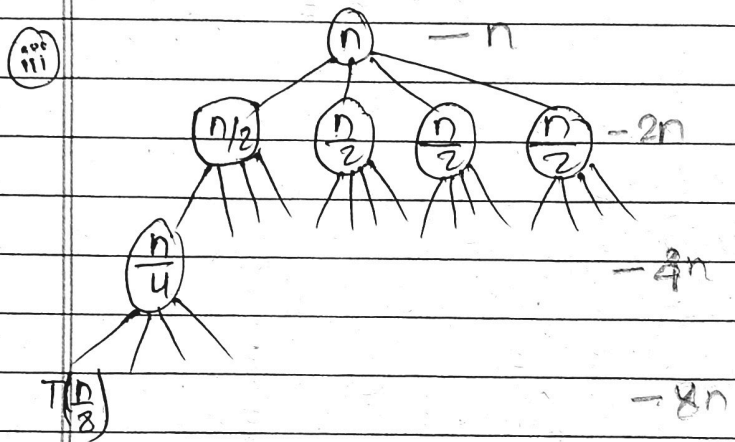
(i)



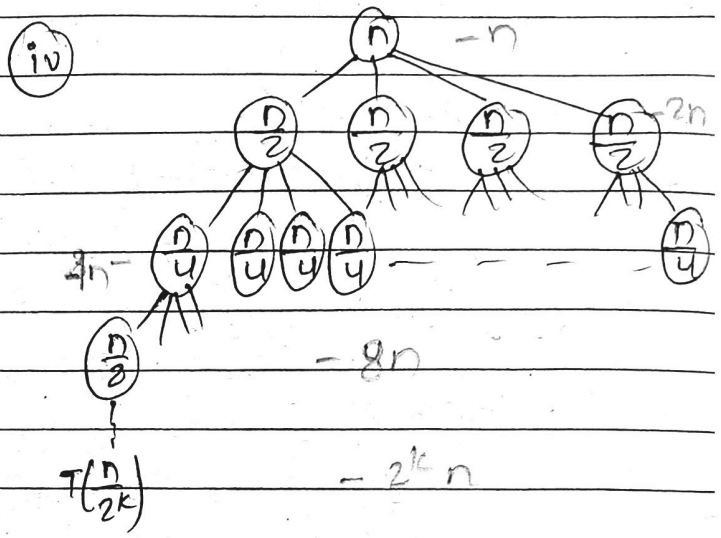
(ii)



(iii)



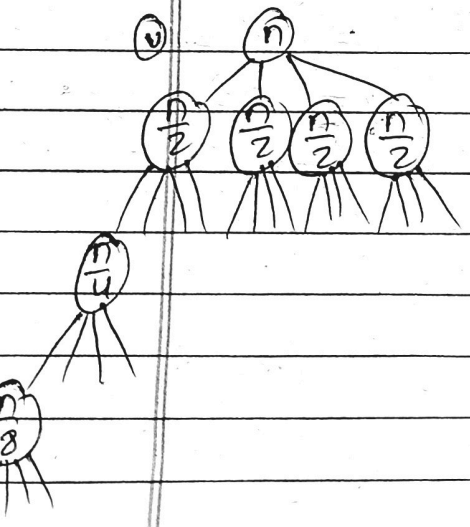
(iv)



⇒ for simplicity assume that $\frac{n}{2^k} = 1$

$\Rightarrow k = \log_2 n$.

(v)



Total steps or Total cost

$= n + 2n + 4n + 8n + 2^k n - - - + 2^k n$

$= n(1 + 2 + 2^2 + 2^3 + - - + 2^k)$

Using geometric:

$S_n = \frac{a(r^n - 1)}{r - 1}$; $r > 1$ Or,

$S_n = \frac{a(1 - r^n)}{1 - r}$; $r < 1$

So, $T(n) = n\left\{1 \cdot \frac{(2^{k+1}-1)}{2-1}\right\}$

$= (2^{k+1}-1) \times n$

$= n(2 \cdot 2^k - 1)$

$= n(2n-1)$

$= 2n^2 - n$

$\leq 2n^2$

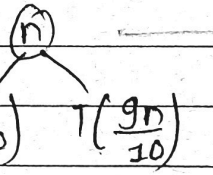
$\therefore T(n) = O(n^2)$

④ Case between worst and best (Quick Sort) | 1:9 portion

$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$ when $n > 1$

$= 1$ when $n = 1$

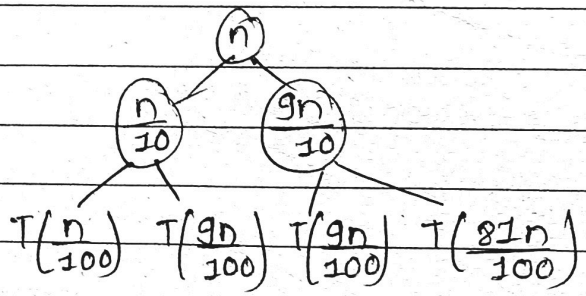
⇒ (i)



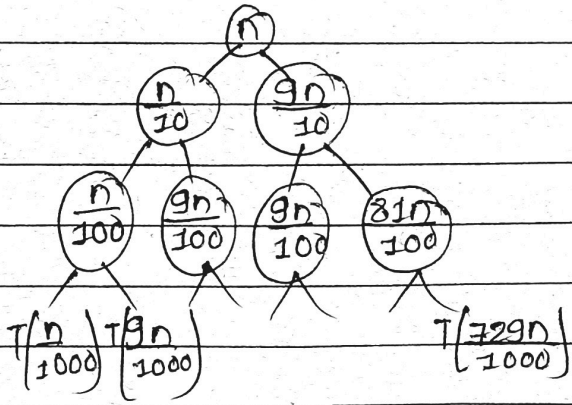
⟶ Represent no. of steps

⟶ Represent input size wherever T occurs.

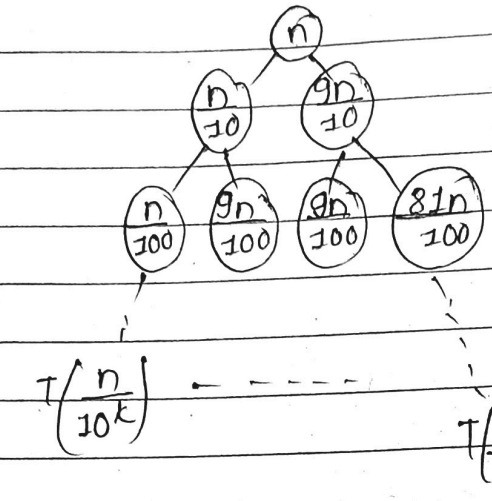
(or ii)



(iii)



(iv) At k^{th} iteration:



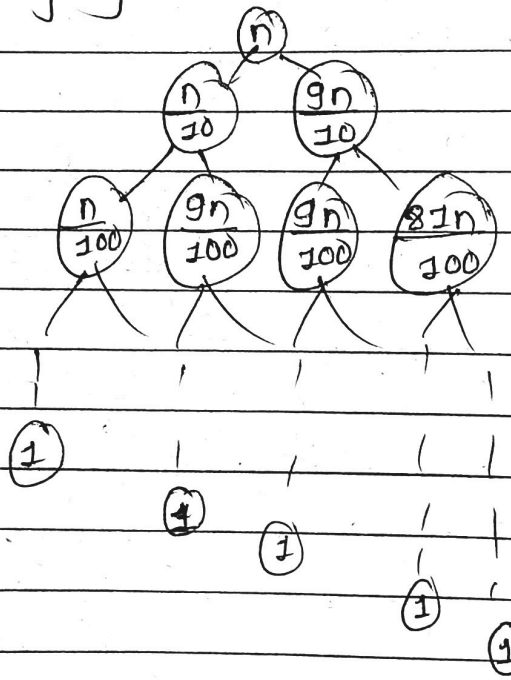
since the portion $\left(\frac{n}{10}\right)$ will reach to 1 earlier as compared to $\left\lfloor\frac{9n}{10}\right\rfloor$. Hence, they are not written in same level. And $T\left(\frac{9^k n}{10^k}\right)$ will complete at the last.

for simplicity:

$$\frac{9^k}{10^k} n = 1 \Rightarrow \left(\frac{10}{9}\right)^k = n$$

$$\Rightarrow k = \log_{(10/9)} n$$

Modifying k^{th} iteration:



Total

n

n

n

$\leq n$

$T(n) \leq n+n+n+n+ \text{---} +n$

$\leq (k+1)n$

$\leq \left(\log_{10/9} n + 1\right) n$

or, $T(n) \leq 2 \times n \log_{\frac{10}{9}} n$

$\therefore T(n) = O\left(n \log_{\frac{10}{9}} n\right)$

⊙ Substitution Method: This method is described using two steps:

(i) Guess the form of the solution.

(ii) Use induction to show that the guess is valid.

Examples:

(1) $T(n) = 4T(n/2) + n$ when $n \geq 1$

$= 1$ when $n = 1$

Guess $T(n) = O(n^3)$

Inductive goal $T(n) \leq cn^3$

Basic case: When $n = 1$

$T(1) = c \cdot 1^3$

$1 \leq c \quad \forall c \geq 1$

This is triviary.

Inductive case: Assume that this is true for all k.

i.e. $T(k) \leq c \cdot k^3 \quad \forall k < n$

$T\left(\frac{n}{2}\right) = c \cdot \left(\frac{n}{2}\right)^3$ is also true

Now, $T(n) = 4T\left(\frac{n}{2}\right) + n$

$\leq 4c\left(\frac{n}{2}\right)^3 + n$

$= \frac{1}{2} cn^3 + n \quad = cn^3 - \left(\frac{1}{2} cn^3 - n\right)$

$\therefore T(n) \leq cn^3$ proved #

Let us take another guess as $T(n) = O(n^2)$

Inductive goal $T(n) \leq cn^2$

Basis case: When $n=1$, $T(1) \leq c \cdot 1^2$

or, $1 \leq c \;\forall\; c \geq 1$

It is triviary.

Inductive case: $T(k) = ck^2 \;\forall\; k < n$

$T(\frac{n}{2}) \leq c \cdot (\frac{n}{2})^2$ is also true.

Now, $T(n) = 4T(\frac{n}{2}) + n$

$\leq 4\{c \cdot (\frac{n}{2})^2\} + n$

$= cn^2 + n \not\leq cn^2$ {Impossible}

Again, Take $T(n) \leq cn^2 - dn$ as inductive goal.

⊙Basis case: When $n=1$

$T(1) \leq c \cdot 1^2 - d \cdot 1$

$1 \leq c-d \;\forall\; c-d \geq 1$

It is triviary.

⊙Inductive case:

Assume that: $T(k) = ck^2 - dk \;\forall\; k < n$

$T(n/2) \leq c(\frac{n}{2})^2 - d(\frac{n}{2})$ is also true

$T(n) = 4T(\frac{n}{2}) + n$

$\leq 4\left(c \cdot (\frac{n}{2})^2 - d(\frac{n}{2})\right) + n$

$= cn^2 - 2dn + n \quad = cn^2 - dn - (dn - n)$

$T(n) \leq cn^2 - dn$

proved #

② $T(n) = T(n-1) + T(n-2) + 1$ when $n > 2$

$= 1$ when $n = 1$ or $n = 2$

Guess $T(n) = O(2^n)$

Inductive goal: $T(n) \leq c \cdot 2^n$

⇒ Basis case: When $n = 1$:

$T(1) \leq c \cdot 2^1$

$1 \leq 2c \quad \forall \; c > 1$

When $n = 2$

$T(2) \leq c \cdot 2^2$

$1 \leq 4c \quad \forall \; c \geq 1$

Inductive case:

Assume that: $T(k) \leq c \cdot 2^k \; \forall \; k < n$

$T(n-1) \leq c \cdot 2^{n-1}$

$T(n-2) \leq c \cdot 2^{n-2}$ are also true.

Now, $T(n) = T(n-1) + T(n-2) + 1$

$\leq c \cdot 2^{n-1} + c \cdot 2^{n-2} + 1$

$= \frac{c \cdot 2^n}{2} + \frac{c \cdot 2^n}{4} + 1$

$= \frac{3}{4} c 2^n + 1$

$= c \cdot 2^n - \left(\frac{1}{4} c \cdot 2^n - 1 \right)$

$\therefore T(n \leq c \cdot 2^n$

Hence, $T(n) = O(2^n)$

proved #

⊙ Substitution method is powerful but it is only applicable to instances where the solutions can be guessed.

⊙ Master Method:

Can only be applicable if and only if it exist in the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

✱ case I: Exists if $aT\left(\frac{n}{b}\right) > f(n)$ ~~or, $f(n) = O(n^{\log_b a})$~~

or, $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$.

$\therefore T(n) = \Theta\left(n^{\log_b a}\right)$

✱ case II: if $f(n) = \Theta\left(n^{\log_b a}\right)$

$\therefore T(n) = \Theta\left(f(n) \log n\right)$

✱ case III: if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$.

$\therefore T(n) = \Theta(f(n))$

Q $T(n) = 7T(n/2) + n^2$

$a = 7$

$b = 2$

$f(n) = n^2$

Now, $n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

$\therefore f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$

$= O\left(n^{2.8 - \epsilon}\right) \quad \forall\ \epsilon \leq 0.8$

Hence, $T(n) = \Theta\left(n^{\log_b a}\right)$

$= \Theta\left(n^{2.8}\right)$

⊙ $f(n) \neq \Theta\left(n^{\log_b a}\right)$

or, $f(n) \neq \Theta(n^{2.8})$

case II fails & so on..

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

$a = 4$

$b = 2$

$f(n) = \frac{n^2}{\log n}$

Here, $n^{\log_b a} = n^2$

case I: $f(n) \neq O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$

$\neq O(n^{2-\epsilon})$

Since, $n^{2-\epsilon} \; ? \; n^2/\log n$

$\frac{n^2}{n^\epsilon} \; ? \; \frac{n^2}{\log n}$

n^ϵ is asymptotically greater than $\log n$.

Therefore, $\frac{n^2}{n^\epsilon}$ is smaller than $\frac{n^2}{\log n}$

$\therefore$ case I is failed.

case II: $f(n) \neq \Theta(n^{\log_b a})$

$\frac{n^2}{\log n} \neq \Theta(n^2)$

$\therefore$ case II is failed.

case III: $f(n) \neq \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$

$\neq \Omega(n^{2+\epsilon})$

since, $n^{2+\epsilon} \; ? \; \frac{n^2}{\log n}$

$n^\epsilon \cdot n^2 \; ? \; \frac{1}{\log n} \cdot n^2$

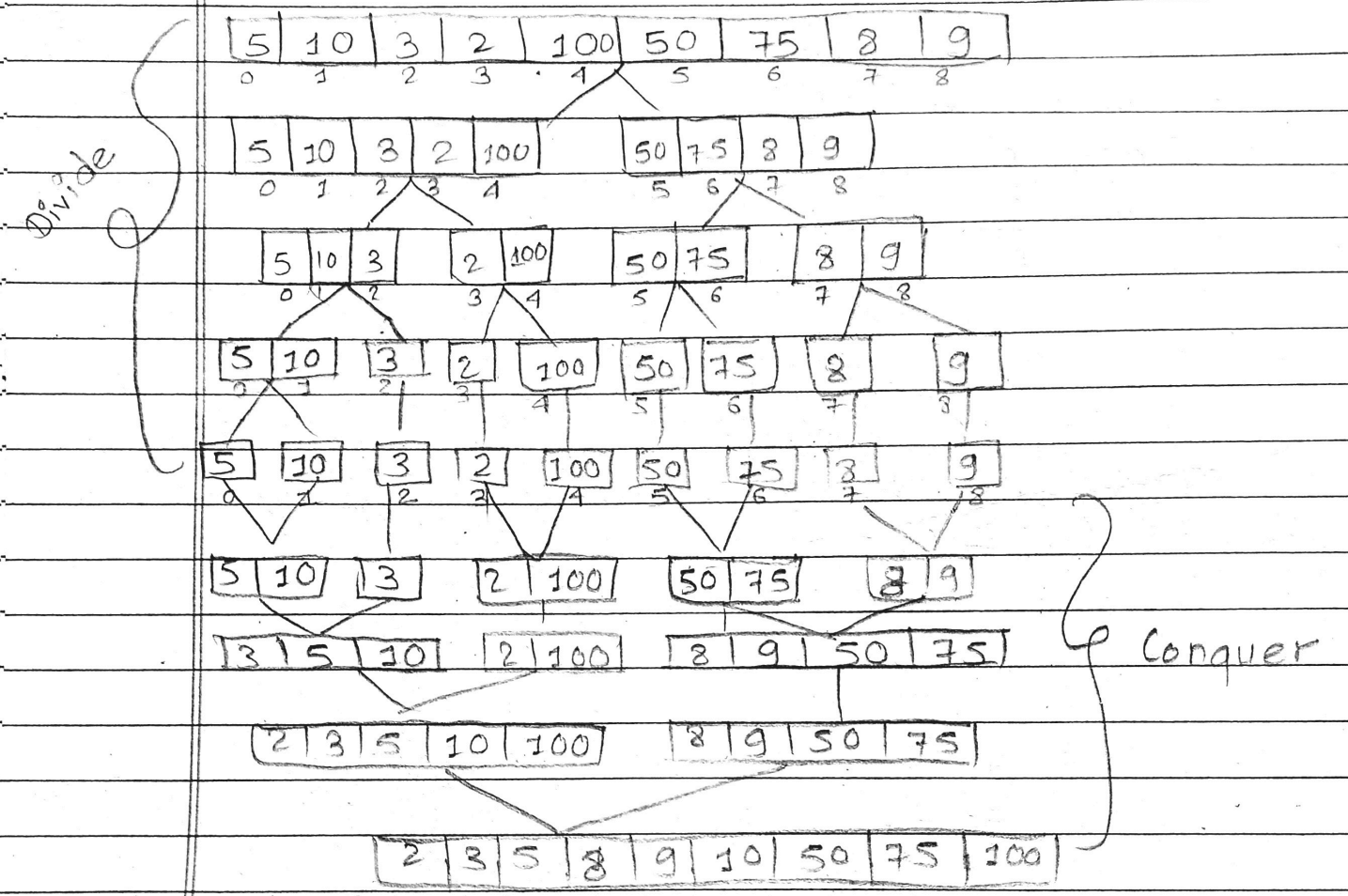
n^ϵ is asymptotically greater than $\frac{1}{\log n}$

$\therefore n^{2+\epsilon}$ is greater than $\frac{n^2}{\log n}$

$\therefore$ case III is failed

Since, all of the case are failed, we cannot solve it using master method.

⊛ Merge Sort




```
Merge_Sort (A, l, r)
    if (l ≤ r) {
                m = ⌊(l+r)/2⌋

    MergeSort (A, l, m);
    MergeSort (A, m+1, r);
```

Merge (A, l, m+1, r);
}

⊛ $T(n) = 2T\left(\frac{n}{2}\right) + n$ when $n > 1$

$= 1$ when $n = 1$

$\therefore T(n) = O(n \log n)$

```
Merge (A, l, m, r) {
    x = l
    y = m
    k = l
while (x < m && y <= r)
    { if (A[x] < A[y])
        { B[k] = A[x];
            k++;
            x++;
        }
    else {
            B[k] = A[y]
            k++;
            y++;
        }
    }
```

```
while (x < m)
  { B[k] = A[x]
    k++;
    x++;
  }
```

```
while (y <= r)
  { B[k] = A[y]
    k++;
    y++;
  }
```

```
for (i=l; i<=r ; i++)
        A[i] = B[i]
}
```

⊛ Quick Sort

```
Quicksort (A, l, r) {
        if (l<r) {
                p= partition ( A, l, r);
        Quicksort (A, l, p-1);
        Quicksort (A, p+1, r);
        }
}

partition (A, l, r) {
                x=l;
                y=r;
                pivot= A[l];
        while (x<y)
            { while (A[x] <= pivot)
                    x++;
              while (A[y] > pivot)
                    y--;
              if (x<y)
              swap (A[x], A[y])
            }
        A[l]= A[y];
        A[y]= pivot;
        return y;
}
```

To eliminate the worst case of Quick sort, we use Randomized Quick Sort.

```
RandQuicksort (A, l, r){
        if(l<r){
        p = Randpartition (A, l, r);
    Read RandQuicksort (A, l, p-1);
        RandQuicksort (A, p+1, r);
}}

Randpartition (A, l, r){
        k = random (l, r);
        swap(A[l], A[k]);
        return partition (A, l, r);
}

partition (A, l, r){
        x = l;
        y = r;
        pivot = A[l];
    while (x<y)
      { while (A[x] ≤ pivot)
            x++;
        while (A[y] > pivot)
            y--;
        if (x < y)
            swap (A[x], A[y])
        }
    A[l] = A[y];
    A[y] = pivot;
    return y;
}
```