Date:

Abdul Bari Lecture Notes

* Algorithms:

Step-by-step procedure for solving a problem, often 'Computational Problems'.

- Difference between 'program' & 'algo':

Algorithm	Program
i) Design Phase	i) Implementation phase
ii) Requires Domain Knowledge	ii) Programming Knowledge
iii) Written in any language (mathematical notations & Pseudocode recommended)	iii) Programming Languages (C, C++, Java, Python)
iv) Not dependent on H/W or OS	iv) Dependent
v) We analyse algorithms	v) We test programs

• Priori Analysis	&	• Posteriori Testing
i) Analysis of algo (Time, Space, ...)		i) Testing of programs (Bytes, seconds, ...)
ii) Language Independent		ii) Dependent
iii) H/W Independent		iii) Dependent
iv) Not actual time but the time function		iv) Actual figures of time & storage required.

Date:_____

- Characteristics of Algorithms:

i) Input can be 0 or more
ii) Output at least 1 thing (serve a purpose)
iii) Definiteness, every statement should be solvable and do-able, it should not be vague or ambiticus
iv) Finiteness, algo should terminate
v) Effectiveness, should complete task efficiently, should not make unnecessary steps.

- Pseudocode Form of writing algorithms:

i) Algorithm swap(a, b) {
    temp = a;
    a = b;
    b = temp;
}

OR

ii) Algo Swap(a, b) begin
    temp ← a;
    a ← b;
    b ← temp;
end

Syntax does not matter, should be understandable

- Analyzing algorithms: (Criteria)

1. Time - Getting time function } Very
2. Space - Getting space function } Imp.
3. Data Transfer
4. Power Consumption
5. CPU Registers (Device drivers, embedded)

Date:

• Eg. of algo analysis: (Small Overview)

```
Algorithm Swap(a, b)
{
    temp = a;  ——— 1
    a = b;     ——— 1
    b = temp;  ——— 1
}
```

We assume that simple statements take constant '1' time

$\therefore f(n) = 1 + 1 + 1 = 3$

$f(n) \rightarrow O(1)$

No matter the no. of operations

eg: Even $x = a * 5 + 6 * b$ .... (1)

However, we can also do in-depth analysis and not assume such statements constant, analysing all the way upto machine code

Real World eg: i) Plan to go to friend's house

ii) Plan to successfully land on Mars.

➢ Similarly for space, each variable used is assumed to use constant space

$\therefore$ a — 1

b — 1

temp — 1 , $S(n) = 1 + 1 + 1 = 3$

$S(n) \rightarrow O(1)$

★ • Frequency Count Method: (IMP)

Used to find Time & Space Complexity of Algorithms by analyzing the accurate Pseudocodes

Date:_____

Eg:

i) Algo Sum(A, n) {
    s = 0; ____ 1
    for (i = 0; i < n; i++) { ____ (n + 1)
        s = s + A[i]; ____ (n)
    }
    return s; ____ 1
}

$\therefore f(n) = n+1+n+1+1 = 2n+3$
$\rightarrow O(n)$

$\therefore S(n)$: s .. 1, & A ... n
i .. 1, n ... 1
Auxiliary Space
Complete Space
$\therefore S(n) = 1+1+n+1 = n+3$
$\rightarrow O(n)$

Note: We ignore constant & less dominant terms to find complexity bounds.

ii) Algo Multiply (A, B, n) { ..where A & B are Matrices
    for (i=0; i<n; i++) { ____ (n+1)
        for (j=0; j<n; j++) { ____ (n(n+1))
            c[i, j] = 0; ____ n^2
            for (k=0; k<n; k++) ____ $n^2(n+1)$
            {
                c[i, j] = c[i, j] + A[i, k] * B[k, i]; ____ n^3
            }
        }
    }
}

$\therefore f(n) = (n+1)+n^2+n+n^2+n^3+n^2+n^3$
$= 2n^3+3n^2+2n+1$
$\rightarrow O(n^3)$

$\therefore S(n)$: A .. n^2, i .. 1, C ... n^2
B .. n^2, j .. 1
n .. 1, k .. 1
$\therefore S(n) = 3n^2+4$
$\rightarrow O(n^2)$

Date:_____

- Analysis of different iterative algorithms using 'Frequency Count Method' :- Understanding Approach

[Only the important & unique ones are mentioned here. for more, watch video]

①
```
for (i=0; i<n; i++){
    for (j=0; j<i; j++){
        stmnt;
    }
}
```

i	j	no. of times
0	0 ✗	0
1	0 ✓	1
	1 ✗	
2	0 ✓	2
	1 ✓	
	2 ✗	
⋮		⋮
n		n

∴ Total No. of times = $0+1+2+\dots+n$
$= n(n+1)/2$
∴ $f(n) = (n^2+n)/2$
$\rightarrow O(n^2)$

②
```
p = 0;
for (i=1; p<=n; i++){
    p = p+i;
}
```

i	p
1	0
2	1 = 1+0
3	3 = 1+2
4	6 = 1+2+3
5	10 = 1+2+3+4
⋮	⋮
k	$\sum(k-1)$ $\approx k\frac{(k+1)}{2}$

∴ Break point: $p > n$
Assume break at 'k'
∴ $\frac{k(k+1)}{2} > n$
∴ $\frac{k^2+k}{2} > n$
Remove const. & less dominant,
∴ $k^2 > n$
Equalize & take root, ∴ $k = \sqrt{n}$
∴ $f(n) \Rightarrow O(\sqrt{n})$

Date:

③ `for(i=n; i>=1; i=1/2){`
`stmnt;`
`}`

i
n
n/2
$n/2^2$
⋮
$n/2^k$

Break at: $i \leq 1$

$\therefore n/2^k \leq 1$

$\therefore n \leq 2^k$

$\therefore \log n \leq k \log 2$

$\therefore \log n \leq k$

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

④ `for(i=0; i*i<n; i++){`
`stmnt;`
`}`

i	i*i
0	0
1	1
2	2^2
3	3^2
⋮	⋮
k	k^2

Break at: $i*i >= n$

$k^2 >= n$

$k >= \sqrt{n}$

$\therefore O(k) = O(\sqrt{n})$

⑤ • Using While Loop:

`a=1;`
`while(a<b){`
`stmnt;`
`a = a*2`
`}`

a
1
2
2^2
2^3
⋮
2^k

Break: $a >= b$

$2^k >= b$

$\therefore k >= \log_2 b$

$\therefore O(k) = O(\log b)$

Considering 'n' as common notation we get $O(\log n)$

Date:

⑥
```
i = 1;
k = 1;
while (k < n) {
    stmnt;
    k = k + i;
    i++;
}
```

i	k
1	1
2	1+1=2
3	1+1+2=4
4	1+1+2+3=7
⋮	⋮
m	$\frac{m(m-1)}{2}+1$

Break at: $k >= n$

$\frac{m(m-1)}{2} + 1 >= n$

$m^2 >= n$, $m >= \sqrt{n}$

$\therefore O(m) = O(\sqrt{n})$

⑦
```
while (m != n) {
    if (m > n)
        m = m - n;
    else
        n = n - m;
}
```

$\therefore$ Best: $O(1)$

Worst / Avg: $O(n)$

Best Case:

$m = 2$, $n = 2$ ... 1 time

Avg Case / Worst Case:

$m = 16$, $n = 2$

16	2
14	2
12	2
10	2
8	2
6	2
4	2
2	2

7-times $\approx m/2$

↓

General notation

$O(n/2)$

↓

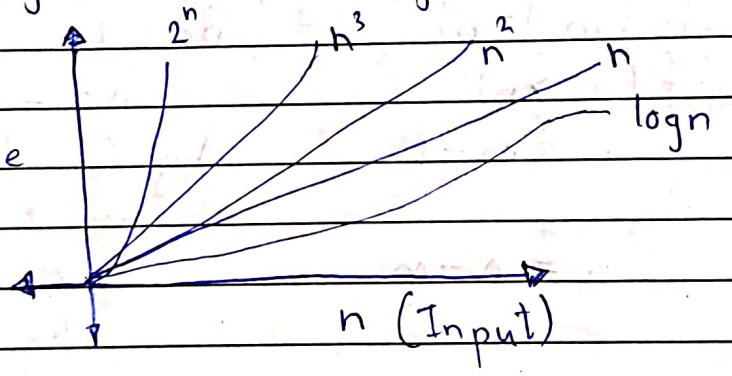
$O(n)$

Date:_____

- Types of Time functions:
  - $O(1)$ ___ constant (eg: $O(1000)$, $O(50)$ ...)
  - $O(\log n)$ ___ Logarithmic, $O(\sqrt{n})$ ___ root
  - $O(n)$ ___ Linear (eg: $O(2n+3)$, $O(145n+24)$...)
  - $O(n^2)$ ___ Quadratic
  - $O(n^3)$ ___ Cubic
  - $O(2^n)$, $O(3^n)$, $O(n^n)$ ___ Exponential

- Comparison of Time functions: (Always true for larger values)

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$



* Asymptotic Notation: (from Mathematics)

i) Big-Oh, (O) gives upper bound

$f(n) = O(g(n))$ if $\exists$ +ve constant 'c' and 'n_0' such that

$$f(n) \leq c * g(n) \quad \forall\ n \geq n_0$$

eg: $f(n) = 2n+3$, Lets take c as 10, n_0 as 1

$\therefore 2n+3 \leq 10n$, $n \geq 1$

Date:

Here, $f(n) = 2n + 3$, $c = 10$, $g(n) = n$, $n_0 = 1$
We can also take,
$c = 1000$ or 100 or 7 till, $c \cdot g(n) \geq f(n)$

To take closest (tight) c:
$2n + 3 \leq 2n + 3n$
$2n + 3 \leq 5n$ for $n \geq 1$
$\therefore f(n) = O(n)$ → Most useful

We can also write:
$2n + 3 \leq 5n^2$ ... True
$\therefore f(n) = O(n^2)$

Here $f(n)$ belongs to class $O(n)$ thus all functions that grow faster ($O(n^2), O(n^3) \ldots O(n^n)$) are suited to be the 'upper-bound'. [$f(n) = O(2^n), f(n) = O(n^3)$]
All functions with lesser growth ($O(1), O(\log n), O(\sqrt{n})$) are thus 'lower-bound'. [$f(n) \neq O(\log n), f(n) \neq O(1)$]
And 'n' i.e $O(n)$ itself is the 'Average-bound'.

ii) Big-Omega, (Ω) gives lower bound

$f(n) = c \cdot g(n)$ if $\exists$ +ve constant 'c' and 'no' such that
$f(n) \geq c * g(n) \quad \forall \quad n \geq 1$
e.g: $f(n) = 2n + 3$
$2n + 3 \geq 1n \quad \forall \quad n \geq 1$
$f(n) \quad \downarrow c \quad g(n)$

Similar to upper-bound properties we can write, here,
Most useful → $f(n) = \Omega(n)$, $f(n) = \Omega(\log n)$, $f(n) = \Omega(1)$
$f(n) \neq \Omega(n^2)$, $f(n) \neq \Omega(2^n)$

Date:

iii) Big-Theta, (Θ), most accurate, gives avg. bound

$f(n) = \Theta(g(n))$ if $\exists$ +ve constant 'c_1' and 'n_0', 'c_2' such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

Here, multiple classes cannot hold true!

eg: $f(n) = 2n+3$

$$\underset{c_1}{1} \times \underset{g(n)}{n} \leq \underset{f(n)}{2n+3} \leq \underset{c_2}{5} \times \underset{(g(n))}{n}$$

$\therefore f(n) = \Theta(n)$ ... Exact function

*Important: These terms are not related to best or worst case!

More Examples:

i) $f(n) = n^2 \log n + n$

$1 \times n^2 \log n \leq n^2 \log n + n \leq 10\, n^2 \log n$

$\therefore f(n) = O(n^2 \log n)$, $f(n) = \Theta(n^2 \log n)$, $f(n) = \Omega(n^2 \log n)$

ii) $f(n) = n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

$= 1 \times 2 \times 3 \times \ldots \times n$

$1 \times 1 \times 1 \ldots 1 \leq 1 \times 2 \times 3 \times \ldots n \leq n \times n \times n \times \ldots \times n$

$1 \leq n! \leq n^n$

Here, we cannot have same thing on both sides.

$\therefore \Theta$ not possible, $f(n) = O(n^n)$, $f(n) = \Omega(1)$

Date:_____

Thus, as Θ gives tight bound that is closest to the actual function, it is preferred over 'O' & 'Ω'. If not possible, we then consider 'O' & 'Ω'.

iii) $f(n) = \log n!$

$\log(1 \times 1 \times \ldots \times 1) \leq \log(1 \times 2 \times 3 \ldots \times n) \leq \log(n \times n \times \ldots \times n)$

$0 \approx 1 \leq \log n! < \log n^n \approx n \log n$

$\therefore f(n) = O(n \log n)$, $f(n) = \Omega(1)$

- Properties of Asymptotic Notations:

i) If $f(n)$ is $O(g(n))$ then $a * f(n)$ is $O(g(n))$

eg: $f(n) = 2n^2 + 5 \rightarrow O(n^2)$

$7 \cdot f(n) = 14n^2 + 35 \rightarrow O(n^2)$

Also applies for $\Omega(g(n))$ & $\Theta(g(n))$ //

ii) If $f(n)$ is given then, $f(n)$ is $O(f(n))$

eg: $f(n) = n^2 \Rightarrow O(n^2)$

iii) If $f(n)$ is $O(g(n))$ & $g(n)$ is $O(h(n))$, $f(n) = O(h(n))$

eg: $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$

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

$g(n) = O(h(n)) = O(n^3)$

then $f(n) = O(n^3)$

Date:_____

iv) If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$
eg: $f(n) = \Theta(g(n))$, $f(n) = n^2$ $\therefore g(n) = n^2$
Thus, $f(n) = \Theta(n^2)$
$g(n) = \Theta(n^2)$
!! Only true for 'Θ' notation.

v) If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$
!! True for 'O' & 'Ω'.
eg: $f(n) = n$, $g(n) = n^2$
then n has $O(n^2)$ &
n^2 has $\Omega(n)$

vi) If $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$
then, $f(n) = \Theta(g(n))$

☆: Uncommon log property: $a^{\log_c b} = b^{\log_c a}$

* Comparison of Functions:
1. Input values and check
2. Use log and check coeff. of bigger terms

Q. $f(n) = n^2 \log n$, $g(n) = n(\log n)^{10}$

Now, $\log[f(n)] = \log(n^2 \log n) = \log n^2 + \log(\log n)$
$= 2\log n + \log(\log n)$ ...(i)

$\log[g(n)] = \log(n(\log n)^{10}) = \log n + \log(\log n)^{10}$
$= \log n + 10\log(\log n)$ ...(ii)

Compare '$\log n$' in (i) & (ii)
$\therefore 2\log n > \log n$ (Asymptotically equal)
$\therefore f(n) > g(n)$

Date:

Q. $f(n) = 3n^{\sqrt{n}}$ $g(n) = 2^{\sqrt{n}\log_2 n}$

Now, $g(n) = 2^{\sqrt{n}\log_2 n} = 2^{\log_2 n^{\sqrt{n}}} = n^{\sqrt{n}}$

As, $3n^{\sqrt{n}} > n^{\sqrt{n}}$ (Asymptotically equal)

$\therefore f(n) > g(n)$

✱ Best, Worst & Average Case Analysis:

1. Linear Search

Best Case: Target at first index → $O(1)$

Worst Case: Last Index or not present → $O(n)$

Average Case: $\frac{\text{All possible case times}}{\text{No. of cases}} = \frac{1+2+3+\dots+n}{n}$

↓

Not possible for every algo.

$= \frac{n(n+1)}{2} \Big/ n = \frac{n+1}{2}$

→ $O(n)$

✱ Again, Cases are not related to notations!
Notations are for function bounds.
Cases are type of analysis done on algo

∴ For Linear Search:

Best Case: $B(n) = 1 = O(1) = \Omega(1) = \Theta(1)$

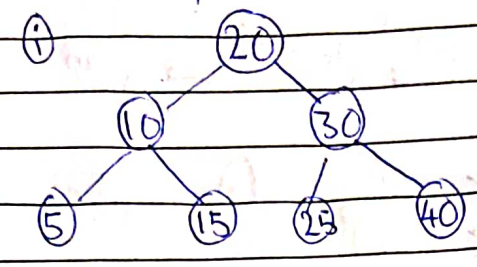
Worst Case: $W(n) = n = O(n) = \Omega(n) = \Theta(n)$

Avg. Case: $A(n) = \frac{n+1}{2} = O(n) = \Omega(n) = \Theta(n)$

Thus, we can look for upper 'O', lower 'Ω' or avg 'Θ' bound for ANY function irrespective of CASE type.

Date:_____

2. Binary Search Tree:



Here, Searching is done inside the tree, similar to simple Binary Search.
n = no. of nodes = 7
Height of tree ≈ $\log n \approx \lceil \log 7 \rceil = 3$
↳ $I_n(i)$

Now, Best Case: Search ends at root node : $B(n) = 1$
Worst Case: Search for leaf node : $W(n)$ = Height of tree $= \log n$
Avg. Case: (Not discussed)

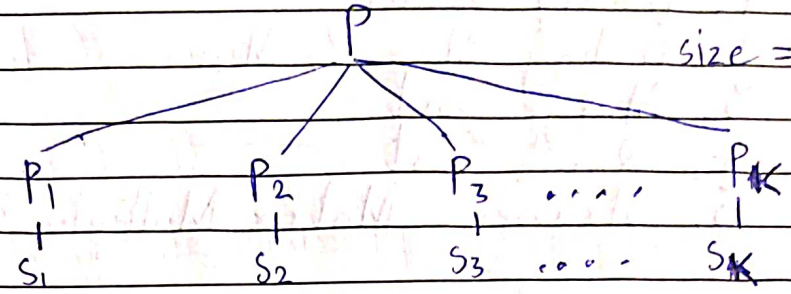
* Divide and Conquer Strategy:

A strategy is applied to solve a computational problem based on the type of problem and suitability.
Various other strategies:
i) Dynamic Programming
ii) Backtracking
iii) Branch & Bound
iv) Greedy Technique

There is no fixed formula to apply strategies but there are guidelines to follow.
Now, we discuss the 'Divide and Conquer' Strategy.

Date:_____

Given a problem 'P' of size 'n', given that the problem is large,



we divide problem into sub-problems, solve it recursively (individually) to get solutions and then combine these solutions to obtain solution for the large problem.
If a sub-problem is also large we apply divide & conquer on it too to get sub-sub-problems and the process continues till a small easy problem is obtained.
The 'sub-problem' however, should be the same type as of that of the original 'large problem'. Thus, these problems are solvable 'recursively'.

- General Algo:

```
DAC (P) {
    if (small(P)) {
        S(P);
    }
    else {
        divide P into P1, P2, P3, ..., Pk
        Apply DAC (P1), DAC (P2), ...  //Recursion
        Combine (DAC(P1), DAC(P2), ...)
    }
}
```

Date:

Examples of problems solvable by Divide & Conquer approach:

1. Binary Search
2. Finding Max & Min
3. Merge Sort
4. Quick Sort
5. Strassen's Matrix Multiplication

- Recursion:

Divide & Conquer is an 'algorithmic paradigm'.

Recursion is a 'programming paradigm' that is used to solve problems using the 'divide & conquer' strategy.

The 'divide & conquer' approach results in a 'recursive' algorithm which is then implemented using recursion.

Thus, in recursion a function calls itself by passing a 'decreasing' or 'dividing' or any other such 'changing' parameter. This continues until the 'base case' that is smallest solvable sub-problem is reached.

The solutions of each call are combined in the preceding calls until eventually the required solution is returned by the first call.

- Recurrence Relations:

Represents recursion logic for the purpose of understanding and analyzing complexity of recursive algorithms.

We will now see types of recurrence relations and the different methods to solve those.

Date:_____

Methods to solve Recurrence Relations:

i) Substitution / Iterative / Plug & Chug Method
ii) Recursive Tree Method
iii) Master's Theorem
iv) Akra Bazzi

Types of Recurrence Relations:

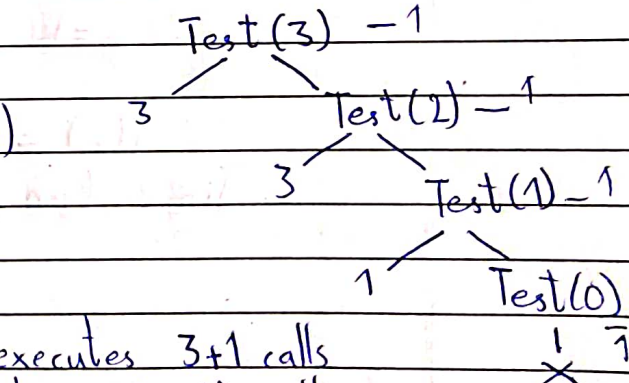
Commonly Solved:
i) Linear Recurrence Relations
ii) Divisive Recurrence Relations

Other Types:
Nonlinear, Homogenous, Non-homogenous, First-order, Higher-order, Full history.

• Examples of Tracing & Solving Recurrence Relations:

①
```
void Test(int n){ ... T(n)
  if(n>0){
    printf("%d",n); ... 1
    Test(n-1); ... T(n-1)
  }
}
```

* Recursive Tree:



∴ From Recursive Tree,
we see Test(3) executes 3+1 calls
∴ For Test(n) we have (n+1) calls.
∴ O(n) is the Time Complexity

Date:_____

* Tracing Recurrence Relation from algo and solving:

∴ $T(n) = T(n-1) + 1$ ... from algo

∴ Relation: $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$

* Substitution / Iterative / Plug & Chug method:

We know for $n>0$,

$$T(n) = T(n-1) + 1 \quad \ldots (i)$$

We also know, parameter is decrementing by 1.

∴ $T(n-1) = T(n-2) + 1$ ... (ii)

And ∴ $T(n-2) = T(n-3) + 1$ ... (iii)

Substitute (iii) in (ii) ∴ $T(n-1) = T(n-3) + 2$ ... (iv)

Substitute (iv) in (i) ∴ $T(n) = T(n-3) + 3$

Now we see the trend, if we continue 'k' times

⋮

∴ $T(n) = T(n-k) + k$

Assume '$n-k=0$' } ∴ $T(n) = T(n-n) + n$

∴ $n=k$ $= T(0) + n$

$= 1 + n$

∴ $T(n) = n+1 \rightarrow O(n)$

∴ Time Complexity is $O(n)$.

Date:_____

(ii)
```
void Test(int n) {          ___ T(n)
    if (n>0) {              ____ 1
        for(i=0; i<n; i++) {  n+1
            printf("%d",n); ___ n
        }
        Test(n-1);          ___ T(n-1)
    }
}
```

∴ Tracing:

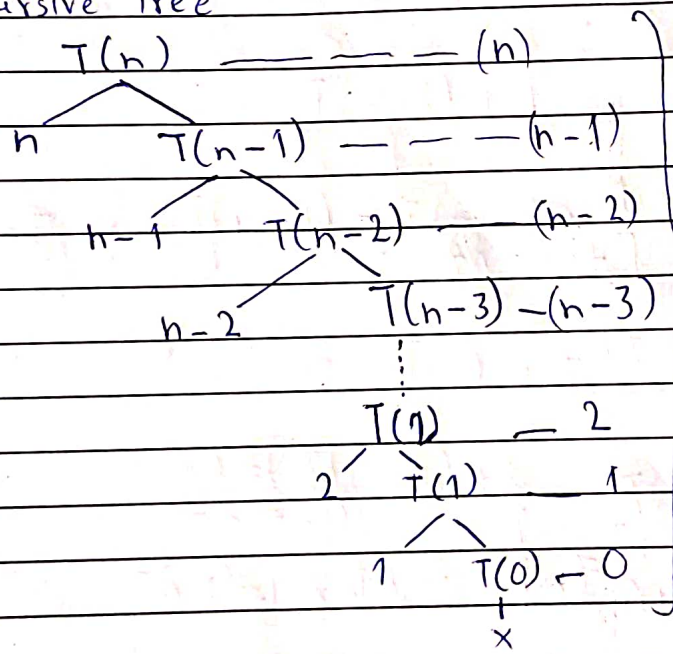
$T(n) = T(n-1) + 2n + 2$

Using Asymp. notation,

$T(n) = T(n-1) + n$

∴ Relation: $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+n & n>0 \end{cases}$

∴ Recursive Tree



$0+1+2+\ldots+(n-2)+(n-1)+(n) = \frac{n(n+1)}{2} = O(n^2)$ //

∴ Using Subst. on Relation,

$T(n) = T(n-1) + n$ ....(i)

$T(n-1) = T(n-2) + n-1$ ...(ii)

$T(n-2) = T(n-3) + n-2$ ...(iii)

Subst in (ii) ∴ $T(n-1) = T(n-3) + n-2+n-1 = T(n-3)+2n-3$

Subst in (i) ∴ $T(n) = T(n-3)+2n-3+n = T(n-3)+3n-3$

∴ Trend is: $T(n) = T(n-k)+kn-k$ } Possibly Inaccurate

Another way: → (Preferred)

$T(n-1) = T(n-3) + (n-2) + (n-1)$

$T(n) = T(n-3) + (n-2) + (n-1) + n$

∴ Assume $n=k$ and it runs 'k' times

$T(n) = T(n-k) + (n-(k-1)) + \ldots + (n-1) + n$

$\therefore T(n) = T(n-n) + 1 + 2 + \ldots + n-1 + n$

$= 1 + 1 + 2 + \ldots + n$

$= 1 + \frac{n(n+1)}{2} = O(n^2)$

(iii)
```
void Test(int n) {          --- T(n)
    if(n>0) {               --- 1
        for(i=1; i<n; i=i*2) {  --- (log n)+1
            printf("%d",i);     --- log n
        }
        Test(n-1);          --- T(n-1);
    }
}
```

$T(n) = T(n-1) + 2\log n + 2$

$\therefore T(n) = T(n-1) + \log n$

Now, $T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + \log n & n>0 \end{cases}$

$\therefore T(n) = T(n-1) + \log n \ldots$ (i)

$T(n-1) = T(n-2) + \log(n-1) \ldots$ (ii)

$T(n-2) = T(n-3) + \log(n-2) \ldots$ (iii)

Subst. in (ii) $\therefore T(n-1) = T(n-3) + \log(n-2) + \log(n-1)$

Subst. in (i) $\therefore T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$

∴ Trend if it runs 'k' times:

$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \ldots + \log(n-1) + \log n$

Date:

Assume 'n-k = 0' ∴ n = k.

∴ $T(n) = T(n-n) + \log(n-(n-1)) + \log(n-(n-2)) + \ldots + \log(n-1) + \log n$

$= 1 + \log 1 + \log 2 + \ldots + \log n$

$= 1 + \log(1*2*3*\ldots*(n-1)*n)$

$= 1 + \log(n*(n-1)*\ldots*2*1) = 1 + \log(n!)$

☆: We don't have a tight bound (Θ) for 'n!' However, we know that upper-bound (O) that is $O(n^n)$ will hold true for 'n!'.

As we consider

∴ $T(n) = 1 + \log(n^n) = 1 + n\log n$

$= O(n\log n)$

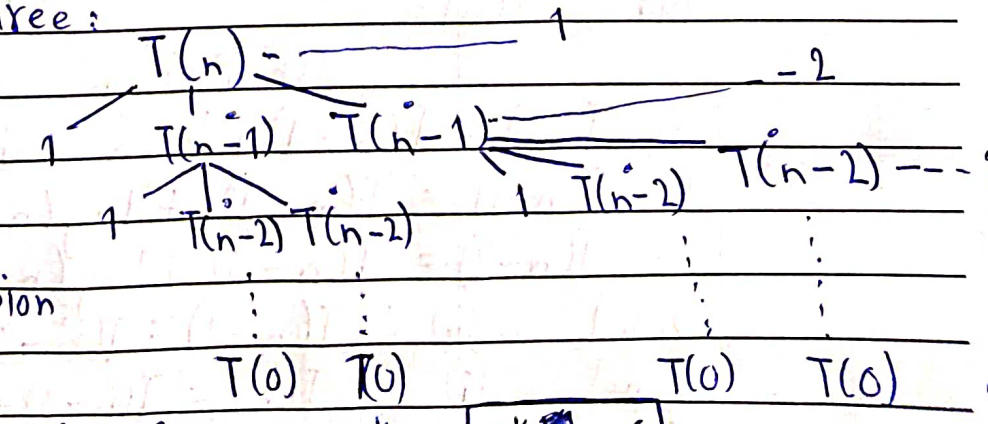
(iv)
```
Algo Test(int n) {                    ___ T(n)
    if (n>0) {                        ___ 1
        printf("%d",n);               ___ 1
        Test(n-1);                    ___ T(n-1)
        Test(n-1);                    ___ T(n-1)
    }
}
```

$T(n) = 2T(n-1) + 2$ → Asymp.

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

∴ Relation: $T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1)+1 & n > 0 \end{cases}$

∴ Recursive Tree:



Geometric Progression Series

∴ $1 + 2 + 2^2 + 2^3 + \ldots + 2^k = \boxed{2^{k+1} - 1}$

Date:

- Basics of Geometric Progression: (Summation)

If,

$$a + ar + ar^2 + ar^3 + \ldots + ar^k = \frac{a(r^{k+1} - 1)}{r - 1}$$

↓ Sum of 'k' terms in GP.

∴ Eg: $1 + (1)2 + (1)2^2 + (1)2^3 + \ldots + (1)2^k$

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

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

Now,

Assume 'n − k = 0' ∴ n = k

∴ $T(n) = 2^{n+1} - 1$

$= 2 \cdot 2^n - 1$

$= O(2^n)$ → Exponential

∴ Subst. method

We know, $T(n) = 2T(n-1) + 1$ ... (i)

$T(n-1) = 2T(n-2) + 1$ ... (ii)

$T(n-2) = 2T(n-3) + 1$ ... (iii)

Subst. in (ii) ∴ $T(n-1) = 2[2T(n-3) + 1] + 1$

Subst. in (i) ∴ $T(n) = 2[2[2T(n-3) + 1] + 1] + 1$

$= 2[2^2 T(n-3) + 2 + 1] + 1$

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

Let the loop run 'k' times

∴ $T(n) = 2^k T(n-k) + 2^{k-1} + \ldots + 2^1 + 2^0$

Assume, 'n − k = 0'

∴ $T(n) = 2^n + 2^{n-1} + \ldots + 2^1 + 2^0$

$= 1 + 2 + 2^2 + \ldots + 2^n$

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

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

∴ Time Complexity is $O(2^n)$.

Date:

- Master theorem for Decreasing (Linear) Functions:

From previous examples, we know patterns:

$T(n) = T(n-1) + 1$	$O(n)$	Multiplied by 'n'
$T(n) = T(n-1) + n$	$O(n^2)$	
$T(n) = T(n-2) + \log n$	$O(n \log n)$	
$T(n) = 2T(n-1) + 1$	$O(2^n)$	Multiplied by exponents.
$T(n) = 3T(n-1) + 1$	$O(3^n)$	
$T(n) = 2T(n-1) + n$	$O(2^n \cdot n)$	

∴ Generalizing:

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

where, $a, b > 0$ and $f(n) = O(n^k)$ where $k \geq 0$

If $a = 1$, we get complexity: $O(n^{k+1})$ or $O(n * f(n))$

If $a > 1$, we get complexity: $O(n^k \cdot a^{n/b})$ or $O(f(n) * a^{n/b})$

eg: $T(n) = 2T(n-3) + \log n \longrightarrow O(\log n * 2^{n/3})$

& $T(n) = nT(n-1) + c \longrightarrow O(n^n)$

If $a < 1$, we get complexity: $O(n^k)$ or $O(f(n))$

- Brief info of parameters & functions in recursion:

For a parameter 'n'

If call in recurrence relation is:

$(n-1) \rightarrow$ Decreasing (Linear) func.

$(n/2) \rightarrow$ Dividing function

$(\sqrt{n}) \rightarrow$ Root function (Non-linear)

Date:

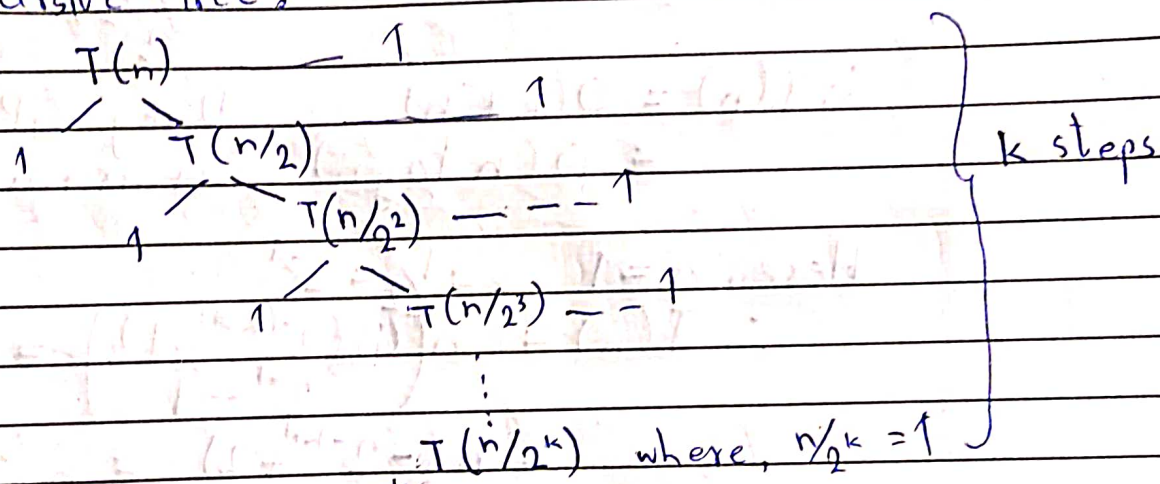
- More Examples (Dividing) Functions:

(v) Algo Test (int n) {
    if (n > 1) {
        printf("%d", n);
        Test(n/2);
    }
}

T(n) = T(n/2) + 2
≈ T(n/2) + 1

∴ Relation: $T_n = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$

∴ Recursive Tree:



∵ $n/2^k = 1$, $n = 2^k$

∴ $\log n = k \log 2$ ∴ $k = \log_2 n$

∴ $T(n) = O(\log n)$

∴ Subst. method:

$T(n) = T(n/2) + 1$ ... (i)

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

$T(n/4) = T(n/2^3) + 1$ ... (iii)

Subst. in (ii), $T(n/2) = T(n/2^3) + 1 + 1$

Subst. in (i), $T(n) = T(n/2^3) + 1 + 1 + 1 = T(n/2^3) + 3$

∴ Trend for 'k' times: $T(n) = T(n/2^k) + k$

Assume '$n - 2^k = 0$', $n = 2^k$, and $k = \log_2 n$ ∴ $T(n) = T(1) + \log n$

$= 1 + \log n = O(\log n)$

Date:_____

- More Basics of Geometric Progression:

If we have a sequence,

$$\underbrace{a + ar + ar^2 + ar^3 + \dots + ar^{n-1}}_{n\text{-terms}}$$

Then: a = first term, r = common constant $= \frac{t_{n+1}}{t_n}$

$t_n = n^{th}$ term $= ar^{n-1}$

S_n = Sum of 'n' terms $= \frac{a(r^n-1)}{r-1} = \frac{a(1-r^n)}{1-r}$

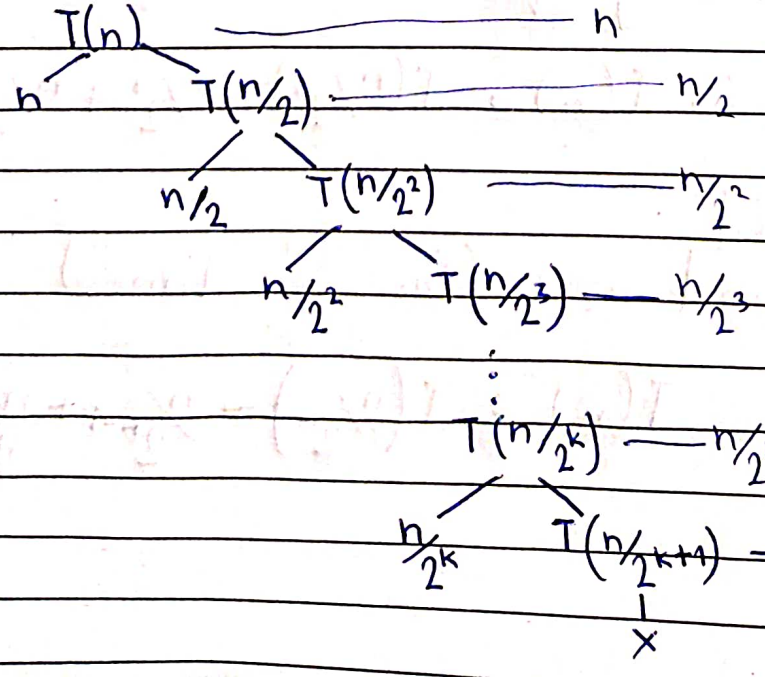
$= na$ ↘ when $r=1$

$\frac{a(r^n-1)}{r-1}$ ↓ useful when $r>1$

$\frac{a(1-r^n)}{1-r}$ ↓ useful when $r<1$

(vi) $$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

Recursive Tree:

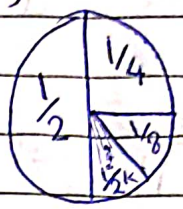


Date:

$$\therefore T(n) = 1 + n + \frac{n}{2^1} + \frac{n}{2^2} + \dots + \frac{n}{2^k}$$

$$= 1 + n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^k}\right)$$

$$= 1 + n\left(1 + \sum_{i=1}^{k} \frac{1}{2^i}\right)$$



$\Rightarrow 1$

$$= 1 + n(1+1)$$

$$= 2n + 1$$

$$= O(n)$$

Subst. Method:

$$T(n) = T(n/2) + n \quad \text{(i)}$$

$$T(n/2) = T(n/2^2) + n/2 \quad \text{(ii)}$$

$$T(n/4) = T(n/2^3) + n/2^2 \quad \text{(iii)}$$

Substituting eq (iii) in (ii),

$$\therefore T(n/2) = T(n/2^3) + n/2^2 + n/2 \quad \text{(iv)}$$

Substituting eq (iv) in (i),

$$\therefore T(n) = T(n/2^3) + n/2^2 + n/2 + n \dots$$

$\therefore$ Trend found as : (for 'k' times)

$$T(n) = T\left(n/2^k\right) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \dots + \frac{n}{2} + n$$

Date:

Assume $n/2^k = 1$, $\therefore T(n) = T(1) + \left(2 + 4 + 8 + \ldots + \frac{n}{2} + n\right)$

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

$= 2n$

$= O(n)$

$\therefore T(n) = T(1) + n\left(1 + \frac{1}{2} + \ldots + \frac{1}{2^{k-2}} + \frac{1}{2^{k-1}}\right)$

Approx ≈ 1

$= T(1) + n(1+1)$

$= 1 + 2n$

$= O(n)$

(vii)
```
void Test (int n) {
    if (n > 1) {
        for (int i=0; i<n; i++) {
            stmnt;          ———— n
        }
        Test(n/2);          ———— n/2
        Test(n/2);          ———— n/2
    }
}
```

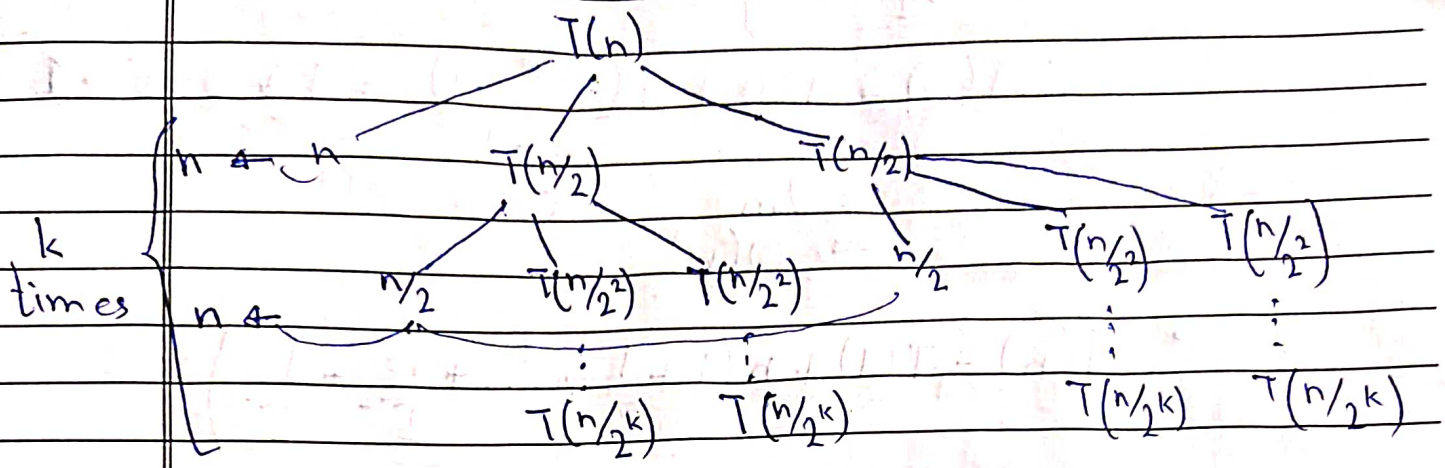
$n + T(n/2) + T(n/2)$

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

$\therefore$ Relation: $T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$

Date:_____

Recursive Tree:



$\therefore T(n) = kn$ and assuming $n/2^k = 1$, $k = \log_2 n$

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

Subst. Method:

$T(n) = 2T(n/2) + n \quad \ldots (i)$

$T(n/2) = 2T(n/2^2) + n/2 \quad \ldots (ii)$

$T(n/4) = 2T(n/2^3) + n/2^2 \quad \ldots (iii)$

Subst. in (ii), $T(n/2) = 2[2T(n/2^3) + n/2^2] + n/2$

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

Subst. in (i), $T(n) = 2[2^2 T(n/2^3) + n] + n$

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

$\therefore$ Trend for 'k' steps: $T(n) = 2^k T(n/2^k) + kn$

Assume, $\frac{n}{2^k} = 1$, $\therefore k = \log_2 n$

$\therefore T(n) = n\,T(1) + n \log n$

$= n + n \log n$

$= O(n \log n)$

Date:

- Master's Theorem for Dividing Functions:

General Form: $T(n) = aT(n/b) + f(n)$

where,

$a \geq 1$ and $f(n) = O(n^k \log^p n)$

$b > 1$

The two important elements: i) $\log_b a$, ii) k

Case 1: $\log_b a > k$, then $O(n^{\log_b a})$

Case 2: $\log_b a = k$, then

(i) If $p > -1$ then, $O(n^k \log^{p+1} n)$

(ii) If $p = -1$ then, $O(n^k \log(\log n))$

(iii) If $p < -1$ then, $O(n^k)$

Case 3: $\log_b a < k$ then,

(i) If $p \geq 0$ then, $O(n^k \log^p n)$

(ii) If $p < 0$ then $O(n^k)$

Eg:

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

Here, $a = 2$, $b = 2$, $f(n) = O(1) = n^0 \log^0 n$

$\therefore k = 0$, $\log_b a = \log_2 2 = 1$

$\therefore \log_b a > k$ ... Case 1,

Then, Time Complexity is $O(n^1)$

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

Here, $a = 4$, $b = 2$, $f(n) = O(n^1)$, $k = 1$, $\log_b a = 2$

$\therefore \log_b a > k$ ... Case 1

Then, Time Complexity is $O(n^2)$

Date:

(iii) $T(n) = 8T(n/2) + n$

$a = 8, b = 2, \log_b a = 3, f(n) = O(n), k = 1$

$\because \log_b a > k$ ... Case 1, $T(n) = O(n^3)$

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

$a = 9, b = 3, \log_b a = 2, f(n) = O(1), k = 0$

$\because \log_b a > k$ ... Case 1, $T(n) = O(n^2)$

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

$\log_b a = 1, k = 1$

$\because \log_b a = k$ ... Case 2

Here, $f(n) = O(n^1 \log^0 n), p = 0 > -1$

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

(vi) $T(n) = 8T(n/2) + n^3$

$\log_b a = 3, k = 3, p = 0 > -1$

$\therefore \log_b a = k$ ... Case 2, $T(n) = O(n^3 \log n)$

(vii) $T(n) = 2T(n/2) + n/\log n$

$\log_b a = 1, k = 1, p = -1 = -1$

$\therefore \log_b a = k$ ... Case 2, $T(n) = O(n \log(\log n))$

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

$\log_b a = 1, k = 1, p = -2 < -1$

$\therefore \log_b a = k$ ... Case 2, $T(n) = O(n)$

(ix) $T(n) = T(n/2) + n^2, \log_b a = 0, k = 2, p = 0$

$\therefore \log_b a < k$ ... Case 3, $T(n) = O(n^2)$

Date:

- More Examples (Root) Functions:

(viii) T(n) ——— void Test (int n) {
if (n > 2) {
1 ——— stmnt;
T(√n) ——— Test(√n);
}
}

∴ Relation: $T(n) = \begin{cases} 1 & n = 2 \\ T(\sqrt{n}) & n > 2 \end{cases}$

$T(n) = T(\sqrt{n}) + 1 = T(n^{1/2}) + 1$ ... (i)

$T(\sqrt{n}) = T(n^{1/2^2}) + 1$ ... (ii)

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

Subst. in (ii), $T(\sqrt{n}) = T(n^{1/2^3}) + 1 + 1$

Subst. in (i), $T(n) = T(n^{1/2^3}) + 1 + 1 + 1$

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

∴ Trend for 'k' times:

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

Assume, $n = 2^m$, $m = \log_2 n$

~~$\frac{k}{2}\log n = \log 2$, $k = \frac{\log 2}{\log n}$~~

∴ $T(n) = T(2^{m/2^k}) + k$

Assume, $2^{m/2^k} = 2$

∴ $\frac{m}{2^k}\log 2 = \log 2$ ∴ $m = 2^k$

∴ $k \log 2 = \log m$, $k = \log_2 m$

∴ $T(n) = T(2) + \log_2 m$

$= 1 + \log(\log n)$

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