AAD Module 1

Algorithm Analysis and Design (APJ Abdul Kalam Technological University)

Algorithm

An algorithm is a finite set of instructions that accomplishes a particular task.

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computable problem.

Problem.

→ every algorithm must satisfy the following criteria

Criteria / characteristics

① input :- externally supplied

② output :- atleast one quantity is produced

③ definiteness : each instruction must be clear and unambiguous

④ finiteness : the algm will terminate after a finite no: of steps

⑤ effectiveness : instruction is basic enough to be carried out (must be feasible)

(algm breakdown chayyumbol each statements edukkumbol. it should be a basic instruction carryout chayyan pattunna)

~~Complexity of an algorithm~~
Algorithm

An algorithm is a finite step by step instruction for solving a particular problem.

Complexity of Algorithm (evaluation of algorithms)

It is the function which gives the running time or space in terms of input size.

Two type of complexity.

① space complexity

② Time complexity

Evaluating algorithm

Algorithm can be evaluated based on their performance

Performance evaluation can be divided into

1. Performance Analysis
2. Performance measurement

* Performance Analysis (Analyzing our program before it is executed in a system)

1) space complexity ✓

2) Time complexity ✓

Space complexity

→ the space complexity of an algorithm is the amount of memory it need to run to completion

⇒ space needed is sum of two components

1) fixed part
2) variable part

$S(P) = C + SP$

Fixed part : It is independent of input & output characteristics

it includes (eg)

⇒ instruction space (i.e space of the code)

⇒ space for constant

⇒ space for variable

Variable part (space for variable whose size is depend on particular problem instance)

⇒ space need by referenced variable

⇒ Recursion stack space etc.

$S(P) = C + SP$

C → fixed, SP → variable

P is any algm

c is fixed part space

SP is variable space

It is dependent on the characteristics of the I/P & O/P's

```
Algorithm
{
return
}
```

```
Algorith
{
s =
fo
s
}
int
float
double
char
algorith
{
i
```

int a[5]

③ Three variable fixed

1) Algorithm xyz(a, b, c)
{
return a+b*c+(a+b-c)/(a+b)+4.0;
}

a → 1 unit
b → 1 unit
c → 1 unit
here no variable part

c + sp

S(p) = 3+0 = 3 (constant)

Algorithm sum(a, n)
{
s = 0;
for i = 1 to n do
s = s + a[i]
return s;
}

Space for i, n, s ⇒ 1 unit each ⇒ 3 unit total

Space for a[i] ⇒ n, ↳ depends on values of n

Total S(p) = 3+n

1	2	3

0 1 2

Algorithm RSum(a, n)
{
if (n ≤ 0) then
return 0
else
return RSum(a, n-1) + a[n];
}

(a, 2)
n = 3
(a, 3)
(a, 2) + a[n]
a, 2
s, 2-1

RSum(a, n) ⇒ 1(a[n]) + 1[n] + 1(return) = 3 unit

RSum(a, n-1) ⇒ 1(a[n-1]) + 1[n] + 1(return) = 3 unit
current index — value of n — return address

RSum(a, n-n) ⇒ 1(a[n-n]) + 1[n] + 1(return) = 3

each call take 3 unit

Here
S(p) = 3(n+1) = 3n+3

recursion, using stack
3 variables are stored, generally
1) formal parameters 2) local variable ③ return address

[there is no local variable]

⇒ each call these are store
⇒ n+1 times executed (Total no: of recursive call → n+1)

a
5	6	7

0 1 2

n = 2

For each recursive call the amount stack is required is 3
Space for parameters a → 1 n → 1
no local variable
space for return address ⇒ 1

[each call 3 space]

Algorithm

④
```
Void main()
{
int x, y, z, sum
printf("enter three nos");
scanf("%d %d %d", &x, &y, &z);
sum = x+y+z;
printf("the sum = %d", sum);
}
```

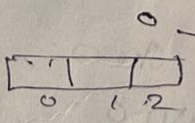
Space for variables x, y, z, & sum

So, Total space needed is 1+1+1+1 = 4.

S(p) = 4 units ⟹ 4×2 = 8 bytes

4×2 = 8 bytes but 4 units

⑤ eg. Algm sum(a, n)




```
{
  s = 0
  for (i=0 do n-1   (i=1; i<=n; i++)
      For (j=0 do m-1
          s = s + a[i][j]
  return s;
}
```

Space for n → 1
s → 1
i → 1
j → 1
m → 1
array a[i][j] → nm

Total S(p) = nm+5

n²+p

~~$f(n) = 3(n+1) \iff 3n+3$~~

1) write an algorithm to find Factorial of a no & find the space complexity

```
1.  Algorithm Fact(n)        => first call
2.  {
3.      if (n==0)
            return 1
4.
5.      else
            return n * Fact(n-1)
6.
7.  }
```

$sp) = 3*(n+1)$

$= 3n+3$

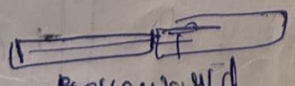
5! n = 2

First call
Fact(2) ⇒ 1
2nd call
(return) + (2) + Fact(1)
3rd call
(return + (1) + Fact(0)

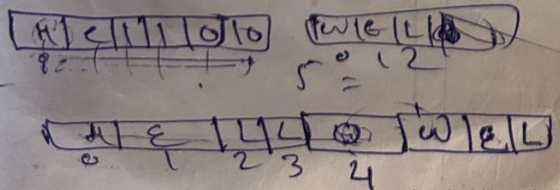
Formal parameter
n = 1
returnaddress = 1
⇒ no local variable

2 * fact(1)
1 * fact(0)
2*1

Hello
welcome
Hellowelcome



3x(5+1)
3x6 = 18+2 = 20



(15)

(16) 1. Algorithm madd (m, n, a, b, c)

2. {

3. for i = 1 to m do
{
for j = 1 do n do

4. {
c[i,j] = a[i,j] + b[i,j];

5. }

6. }

Space complexity = space for 5 parameters (a, b, c, m, n) & space for loc

variables: (i, j)

m ⇒ 1, n ⇒ 1, a[ ] = mn×1, b[ ] = mn×1, c[ ] = mn×1

i = 1, j = 1

S(p) = 3mn + 4

Time complexity (Balance)

Generally, time complexity of an algm Expressed using big O notation. The TC is normally estimated by counting the no: of elementary operations performed by the algm, then express the efficiency of algms using the growth function.

→ in the case of array first push starting addr

→ to keep starting address
① a ⇒ 1

→ n is the size of the arr
② n ⇒ 1

→ there is no local va

recursion

→ return address is a address ③ → 1

→ for each recursive the amount of stack
↳ 3 unit

TCP

Time complexity

* If the time complexity is lower, it means the algm is faster.

⇒ It the amount of computer time it needs to run to completion (Time complexity of an algm measures how much time an algm requires to run the prgm completely)

⇒ The Total time (TCP) taken by a progm P is the sum of its compile time & its run time (1 time only)

$T(P)$ = compile time + Execution time

⇒ Time complexity consider only Execution Time (tp) (run)

⇒ For calculating time complexity we use a method called Frequency count [counting the no: of steps]

① Step count | Frequency count

⇒ we can assign the count value for each line

⇒ algm heading ⇒ 0 step
⇒ comments ⇒ 0 "
⇒ For Braces ⇒ 0 "
⇒ Assignment, conditional statement = 1 step
⇒ For Expression ⇒ 1

⇒ For any looping statements ⇒ No: of times the loop is repeating (n+1) steps. [successful + unsuccessful]

⇒ Body of the loop ⇒ n steps.

eg:) Algorithm ~~abc(a,b,c)~~

1. algorithm abc(a,b,c) ——→ 0
2. { ——→ 0
3. return a+b*c(a+b-c)/(a+b)+4.0; → 1
}

——→ 0

1 Unit

⇒ Every simple statements in an algorithm takes one unit of time.

T(p) = T(1) / O(1)

```
/*----*/
{
a=1
n=5
n2=5
}
```

```
/* Addition of two */
For(i=0;i<n
```

O(1)

Algorithm 2

```
1. Algorithm Sum(a,n)
2. {
3.     s=0.0;
4.     For i=1 to n do
       {
5.     s=s+a[i];
       }
6.     return s;
7. }
```

Step count

```
1. Algorithm Sum(a,n)        ---> 0
2. {                         ---> 0
3.     s=0.0;                ---> 1
4.     For i=1 to n do       ---> n+1
       {
5.     s=s+a[i];             ---> n
       }
6.     return s;             ---> 1
7. }                         ---> 0
```

7	2	9	6	1
0	1	2	3	4

i=1

i=0	i<n	i++
i=0	0<5	true i++
i=1	1<5	" i++
i=2	2<5	" i++
i=3	3<5	" i++
i=4	4<5	" i++
i=5	5<5	false

$T(n) = 2n+3$

$TC = O(n)$

n ⇒ True va

nil ⇒ False

③ Algorithm RSum(a,n)

```
{
    If(n<=0) then return 0.0;      ⟹ 1+1 = 2
    else
        return RSum(a,n-1)+a[n];   ⇒ 2+T(n-1)
}
```

$$T(n) = \begin{cases} 2 & \text{if } n \le 0, \; n=0 \\ 2+T(n-1) & \text{if } n>0 \end{cases}$$

$n = n-1$

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

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

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

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

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

Repeat

Node:

$= 2n + T(0)$

$T(n) = 2n + 2$ $\quad O(n)$

Frequency count

```
Algm sum (a[], n, m)                    ---> 0
{                                       ---> 0
                                        ---> n+1
  for i = 1 to n do                     ---> n(m+1)   - Execution
  {                                          ↳ nm+n
    for j = 1 to m do                   ---> nm
      s = s + a[i][j]                   ---> nm
  }                                     ---> 1
  return s;
}
```

$n^2 + n$

n^2

$2n^2 + n$

$O(n^2)$

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

```
* Algorithm sum()          ---> 0
  {                        ---> 0
  // this is an example     ---> 0
  a = 5;                   ---> 1
  b = 10;                  ---> 1
  c = a+b;                 ---> 1
  }                        ---> 0
```

3 ⇒ constant

$TC = O(1)$

Note: If there is no iteration or recursion in the algorithm then the complexity is order of 1 i.e constant.

method to calculate complexity

Frequency count :- how many times is executed in a piece of code (the no. of operations / steps)

Rules

- Comments & Declaration (variable, Function, array declaration) → Step count = 0 (FC has FC = 0)
- Return & Assignment statement → Step count = 1
- Ignore low order exponent when higher order exponents are present.

~~eg: 5~~

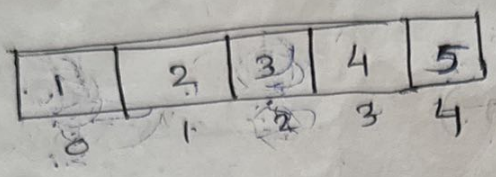
For Eg consider $5n^4 + 7n^3 + 10n^2 + n + 100$

($5n^4$ — Higher order exponent consider; $7n^3 + 10n^2 + n + 100$ — Low order exponents ignore)

Time complexity (TC) = $O(n^4)$ (Big O) / order of

$O(n^4)$

- Ignore constant multiplier $5n^4$ → avoid 5



Key = 3

6

3

i =

a[i] == key

a[i]

* Algorithm hello()
{
For(i = 0; i<n; i++) → 0 → 0 → n+1
{
print hello; → n
}
} → 0 → 0

2n+1 , TC = O(n)

* Algorithm Add(a,n)
{
For(i=0; i<n; i++) → n+1
{
For(j=0; j<n; j++) → n(n+1) = n²+n
{
s[i][j] = a[i][j]+b[i][j] → n²
}
}
}

2n²+2n+1

TC = O(n²)

⇒ In certain case we cannot find the exact value of frequency count. In this case we have 3 types of frequency count.

① Best case : It is the minimum no: of steps that can be executed for a given parameter. (Instance of size n)
⇒ Denoted by Ω
⇒ It's the minimum time taken by an algm to run to its completion

② Worst case : It is the maximum no: of steps that can be executed for a given parameter
⇒ Denoted by O

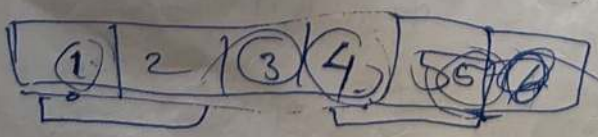
③ Average : It is the Average no: of steps that can be executed for a given parameter.
Denoted by Θ

eg: Linear search (It sequentially check each element of the list until a match is found)

Best case : Search data will be in the first location of the array.

Worst case: Search data does not exist in the array / Last Location

Average case: Search data is in the middle of the array. (n/2 elements)



PQ

Algorithm search (a, n, x)

```
{
    For i=0; i<n; i++          -> 1
    for i=0 do n do            -> 1
    {
        if a[i] == x then      -> 1
            return i;          -> 1
        else
            return -1; (does not exist)
    }
```

	B	W	A
	1	n+1	n/2
		n	n/2
	1	0	1
	1		
TCD	3	1, 0	0
		2n+2	n+

Time complexity = 3 | 2n+2 | n

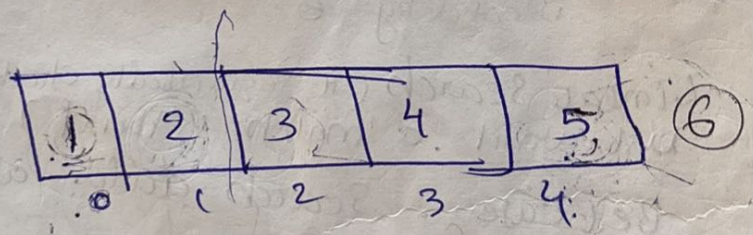
$\Omega(1)$ | $O(n)$ | $\Theta(n)$

$\frac{n}{2} + \frac{n}{2}$

$\frac{2n}{2}$

$O(1)$

$O(n) =$



$n = 5$

X

a[i] == x.

$\frac{n}{2} + \frac{n}{2} = \frac{2n}{2}$ n+1

$\frac{n}{2} + \frac{n}{2} =$

n+1+n+1

2n+2

B(1)

n+2+n

$\frac{2n}{2} + 1$

n+1

n

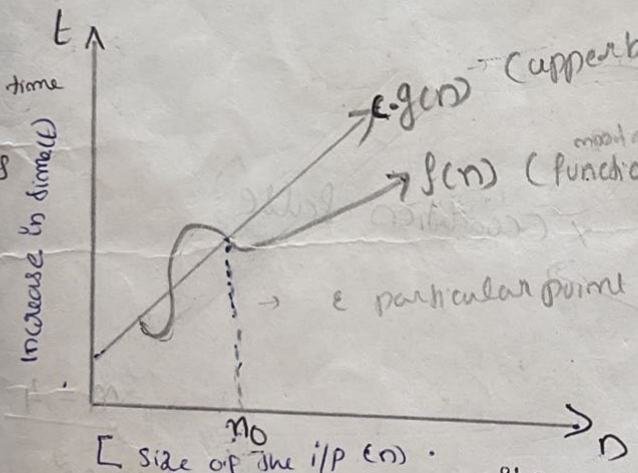
1

Asymptodic Notations For complexity

⇒ mathematical notations used to describe the running time of an algorithm
⇒ (mathematical way to representing time complexity)
⇒ Notations are defined in terms of functions.
Types: Big-oh Notation(O), Big omega, Big theta, Little oh, Little omega (ω)

① Big "oh" (O)

⇒ It is used to find the worst time complexity (maximum time for executing an algorithm)
⇒ It is used to calculate upperbound of the function f(n)

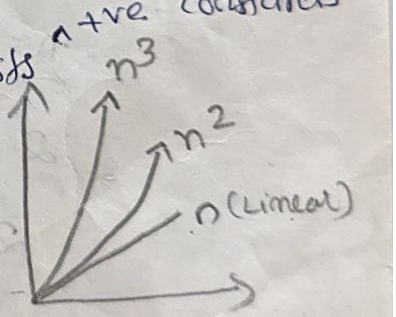


The function f(n) = O(g(n)), if there exists +ve constants c and n0

f(n) = O(g(n))

f(n) ≤ c.g(n) for all n >= n0

(c ≥ 0, n0 >= 1)



(here upperbound)

n² ∈ n

g(n) = n²

eg: consider this example f(n) = 2n² + n + 3

2n² + n + 3 ≤ c.g(n)

Derive the big o notation

$f(n) \leq c \cdot g(n)$ $\quad g(n) = n^2$

$2n^2 + n + 3 \leq c \cdot n^2$

here $c = 3$

(n^2 ன் coefficient ன் next value)

next task, To find no value

$f(n) \leq c \cdot g(n)$

If $n = 0$

LHS RHS

$2 \times 0^2 + 0 + 3 \leq 3 \cdot x\, 0^2$

$0 + 3 \leq 0$ ✗ (condition false)

$n = 1$

LHS RHS

$2 \times (1)^2 + 1 + 3 \leq 3 \times 1^2$

$2 + 4 \leq 3$

$6 \leq 3$ ✗ (condition false)

$n = 2$

$2 \times (2)^2 + 2 + 3 \leq 3 \times (2)^2$

$2 \times 4 + 6 \leq 3 \times 4$

$8 + 6 \leq 12$

$14 \leq 12$ ✗ (condition false)

$n = 4$

$2 \times (4)^2 + 4 + 3 \leq 3 \times 4^2$

$32 + 7 \leq 48$

$39 \leq 48$ (✓)

If $n = 3$

$2 \times (3)^2 + 3 + 3 \leq 3 \times 3^2$

$18 + 6 \leq 27$

$24 \leq 27$ ✓ (Condition True)

the above equation is True when $n \geq 3$

Where $n_0 = 3$

$n_0 = 3$

$f(n) = O(g(n))$

$f(n) = O(n^2)$ (It is the maximum value)
(least upper bound)

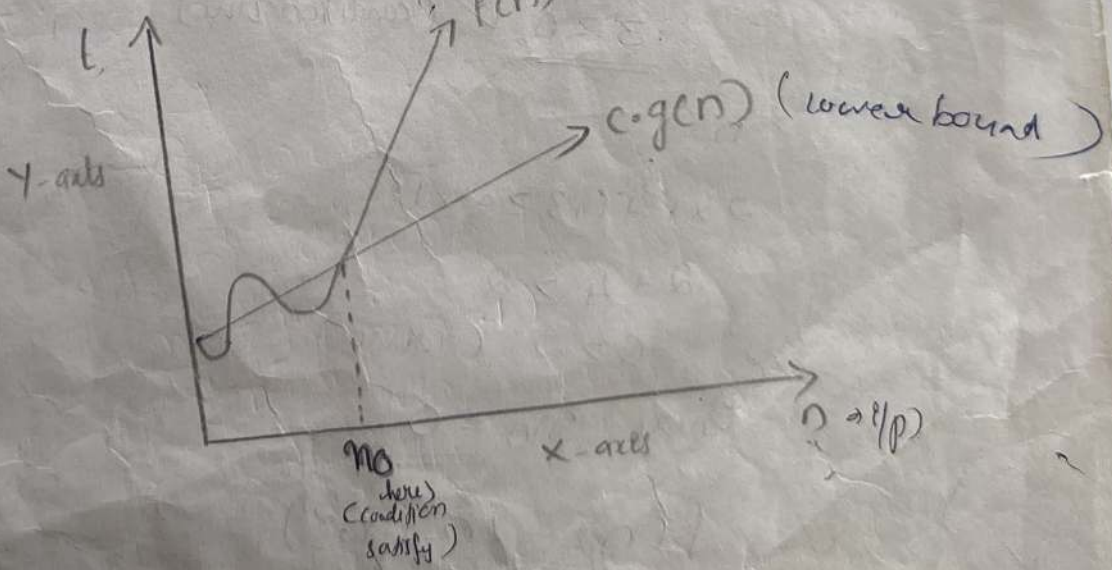
$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots$

↳ Best Time complexity = 1, then $\log n$, then $\sqrt{n}$ then n the

Big - Omega (Ω)

⇒ It is used to find the Best Time complexity (minimum value)

⇒ find greatest lower bound



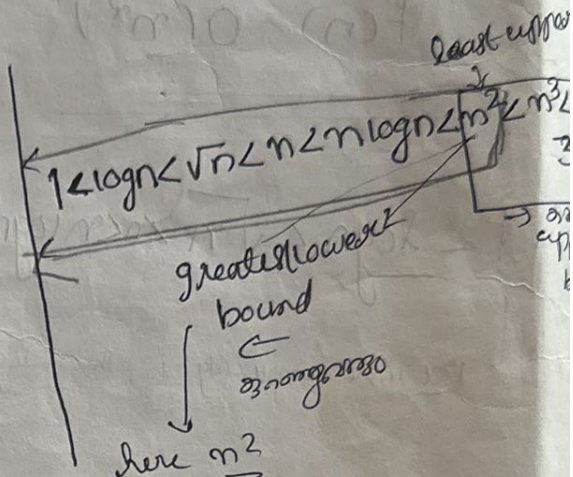
$f(n) = \Omega \cdot g(n)$ here c is a constant

$f(n) \geq c \cdot g(n)$

$c \geq 0$

$n \geq n_0$

$n_0 \geq 0$



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

$2n^2 + n + 3 \geq c \cdot n^2$

here $c = 2$ (coefficient of n^2)

$2 \times n^2 + n + 3 \geq 2 \cdot n^2$, $g(n) = n^2$

First case $n = 0$

$2 \times 0^2 + 0 + 3 \geq 2 \times 0^2$

$0 + 0 + 3 \geq 0$

$3 \geq 0$ ✓ (condition true)

$n = 1$

$2 \times 1^2 + 1 + 3 \geq 2 \times 1^2$

$2 + 4 \geq 2$

$6 \geq 2$ (True)

here $n_0 = 0$

$f(n) = \Omega(n^2)$

$2n^2 + n + 3 = \Omega(n^2$

Asymptotic Notations

⇒ mathematical Notations used to describe the running time of an algorithm (mathematical way to represent time complexity.

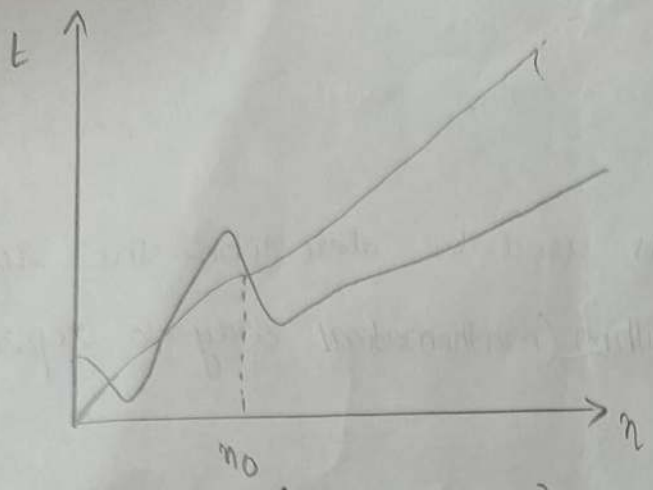
⇒ Notations are defined in terms of functions.

Types

- Big-oh Notations (O)
- Big omega notation (Ω)
- Big theta notation (θ)
- Little oh (o)
- Little omega (ω)

Big-oh Notation (O)

The function $f(n) = O(g(n))$, if there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n >= n_0 (c>0, n_0>=1)$



$f(n) = O(g(n))$

$\Rightarrow g(n)$ is an asymptotic upper bound for $f(n)$

- we try to find what is the worst case or upper bound of the function $f(n)$
- Big oh notation gives the maximum time an algorithm can take
- It describe the worst case of an algorithm time complexity
- the x-axis of the above graph showing the size of the I/P (n) and Y axis showing the increase in time(t)
- If we bound the time complexity or f(n) of a given algorithm in terms of I/P n with another function called c.g(n) for all the I/P size greater than no. then we can say that the f(n) is upper bound

with the function g(n) with the multiple of some constant called c.

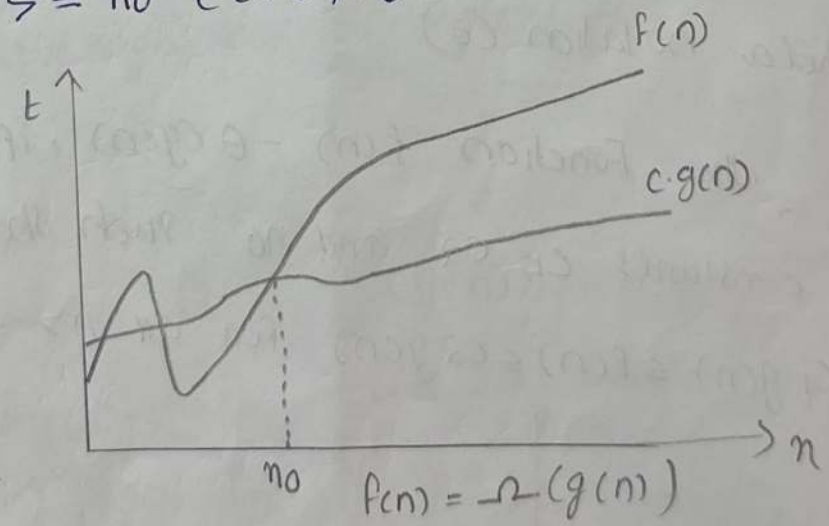
or

$f(n) <= c \cdot g(n)$ for all input $n >= n_0$, and $c > 0$ and $n >= 1$

then we can say that $f(n) = O(g(n))$, which means f(n) is smaller than $c \cdot g(n)$.

2. Big omega notation (Ω)

the function $f(n) = \Omega(g(n))$, if there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n >= n_0$ ($c > 0, n_0 >= 1$)



$f(n) = \Omega(g(n))$

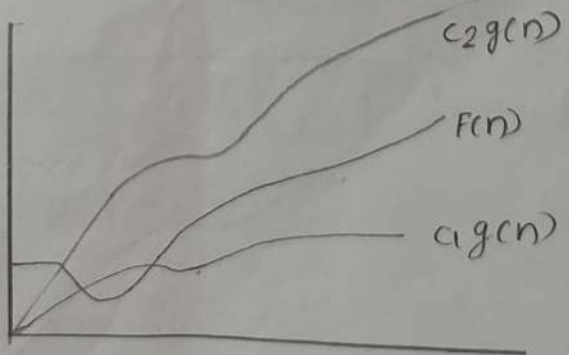
→ g(n) is an asymptotic lower bound for f(n)

- $F(n) = \Omega(g(n))$, IF there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq F(n)$ for all $n \geq n_0$
- the above graph shows the time complexity analyses of any algorithm in terms of Big-Omega. Suppose, for a given problem you have a algorithm. To find out the time complexity in terms of lower bound, we have define a function called f(n) and so bound it with lower bound $c \cdot g(n)$
- Big omega notation always indicate min time required for an algorithm for all input value
- It describe the best case time complexity

3 Big Theta notation (Θ)

The Function $F(n) = \Theta(g(n))$, if there exist +ve constants c_1, c_2 and n_0 such that

$c_1 g(n) \leq F(n) \leq c_2 g(n)$ for all $n >= n_0$ ($c_1, c_2 > 0, n_0 >= 1$)



$F(n) = \Theta(g(n))$

g(n) is an asymptotically tight for f(n)

For a given problem, we have a functions $c_1 g(n)$ and $c_2 g(n)$ that is upper bound as well as lower bound the function $f(n)$. Then we can say that the time complexity of the given problem is $F(n) = \Theta g(n)$.

- We can say that, the function $F(n)$ is upper bound and lower bound. Hence, $c_1 g(n) <= F(n) <= c_2 \cdot g(n)$. i.e

$$f(n) = \Theta g(n)$$

- theta notation encloses the function from above & below
- $f(n)$ is bounded by $g(n)$ both in the lower & upper.
- Represent average case complexity

4. Little oh(o)
   - the function $f(n) = o(g(n))$ iff for any +ve constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < g(n)$ for all $n \geq n_0$
   - It is asymptotically loose upper bound

$$\left(\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \right)$$

$g(n)$ becomes arbitrarily large relative to $f(n)$ as n approaches infinity

5. Little omega (ω)

- The function $f(n) = \omega(g(n))$ iff for any +ve constant $c > 0$, there exist a constant $n_0 > 0$ such that $f(n) > c \cdot g(n) \geq 0$ for all $n \geq n_0$

- It is asymptotically loose lower bound.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity

Properties of Asymptotic Notation

- Reflexivity
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$
  - $f(n) = \Theta(f(n))$

$a \geq b$
$b \geq a$

- Symmetry
  - $f(n) = \Theta(g(n))$ if any only if $g(n) = \Theta(f(n))$

- Transpose symmetry
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
  - $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Transitivity

- $F(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $F(n) = O(h(n))$
- $F(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$
- $F(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- $F(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$
- $F(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

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

lower bound

upper bound

upper bound

tight bound

$c = 10$ ആയാൽ

[Recursive Time complexity Represent using Recurrence Relation]

Recurrence Equations

A Recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.

- There are several methods for solving recurrence relations.
  - Iteration method | substitution
  - Recursion tree method
  - master's method

Fc	Step
2	n<=0
1	
1	

1) Algorithm Rsum(a,n)
{
if(n<=0) — 1
return 0 — 1
else
return a[n]+Rsum(a,n-1)
}

Iteration method

2) Algorithm Fib(n)
{
if(n==0 || n==1)
return n;
else return fib(n-1)+fib(n-2);
}

Recurrence relation

$$T(n) = \begin{cases} 2 & ; n<=0 \\ 2+T(n-1) & ; n>0 \end{cases}$$

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

⇒ T(n) Time required to solve a problem of size n

Recurrence relation

$$T(n) = \begin{cases} 3 & ; \text{if } n==0 \text{ or } n==1 \\ 3+T(n-1)+T(n-2) & ; \text{if } n>1 \end{cases}$$

0 1 1 2 3

③ Algorithm mergesort(a, 0, n) (low, high)

{

mergesort(a, 0, n/2)

mergesort(a, 0, n/2 + 1, n)

merge(a, 0, n) (low, high)

}

Algorithm merge(a, 0,

{

for(i=0 to n)

merge the sub arrays

}

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

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

$\frac{n}{2}$

$\frac{n}{2}+1(n)$

$\frac{2n}{2}+n$

$\frac{2n}{2}$

① Solution of Recurrence Relation through Iterative method: [It means to expand the recurrence & express it as a summation of terms of n & initial condition]

1) $T(n) = \begin{cases} 2 & \text{if } n=0 \\ 2+T(n-1) & \text{when } n>0 \end{cases}$

$T(n) = 2 + T(n-1)$ — (i)

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

$= 2 + [2 + T(n-2)]$

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

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

$= 2 + T(n-2)$

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

$= 2 + T(n-3)$

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

$= 2\times 3 + T(n-3)$

$= 2+2+2+[2+T(n-4)]$

$= 2\times 4 + T(n-4)$

$\vdots$

$= 2\times i + T(n-i)$

when $i = n$

$T(n) = 2\times n + T(0)$

$= 2n + 2$

$= O(n)$

n is the size of i/p. Same function Repeatedly call ചെയ്യുന്നു. നമ്മുടെ i/p size n ആണെങ്കിൽ പിന്നെ call ചെയ്യുമ്പോൾ i/p size n-1

* $T(n) = 1 + T(n-1)$ ①

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

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

$\vdots$

K^{th} term $= K + T(n-K)$

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

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

$= O(n)$

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

$= 1 + T(n-2)$

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

$= 1 + T(n-3)$

$\vdots$

Assume $n-K=1$

$K = n-1$

$$* \; T(n) = \begin{cases} 2T(n/2) + 2 & \text{if } n > 2 \\ 1 & \text{if } n = 2 \end{cases}$$

$$T(n) = 2 + 2T(n/2) \quad — ①$$

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

$$= 2 + 2[2 + 2T(n/2^2)]$$

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

$T(n/4) =$
$2 + 2T(n/8)$
$2 + 2T(n/2^3)$

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

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

$$\vdots$$

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

k^{th} term

2 പൊതുവായി എടുക്കാം

out 2

Assume that $n/2^k = 2$ (if $n=2$ → 1) → $2^k = n/2$

$$T(n) = 2[1 + 2 + 2^2 + \dots 2^{k-1}] + 2^k \cdot T(n/2^k)$$

GP (geometric progression)

$$= 2[2^k - 1] + 2^k T(2) = 2 \times 2^k - 2 + 2^k$$

$2^k = n/2$

$$= 3 \times 2^k - 2 = 3 \times (n/2) - 2$$

(ഇതിലെ ഉള്ള variable n ആണ് എടുക്കേണ്ടത്)

$$= O(n)$$

$$= 2 \times \frac{n}{2} - 2 + \frac{n}{2}$$

$$= n - 2 + n/2$$

Geometric progression

$a, ar, ar^2, ar^3, ar^4, \ldots ar^n$

a = first term r = common ratio

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

Sum of first n term $S_n = a + ar + ar^2 + ar^3 + \ldots ar^{n-1}$

Sum of n terms (sum)

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

Infinite Geometric progression $a, ar, ar^2, ar^3 \ldots ar^{n-1} \ldots$

the sum of infinite geometric series

$$S_\infty = \frac{a}{1-r} \quad (|r| < 1)$$

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

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

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

$$= 2 \times 2^k - 2 + 2^k \times 1$$

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

$$= 4^k - 2 + 2^k \qquad -2 \quad -1$$

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

$$4^k - 1 = 2^k(2^k - 1)$$

$n = 2k$

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

$\phi \times n/2 - 2 + n/2$

$n - 2 + \frac{n}{2}$

$T(1) = 1$

① $T(n) = 2T(n/2) + n$ ——— (1)

② $T(n) = n + 2T(n/2)$

$= n + 2\left[(n/2) + 2T\left(\frac{n}{2^2}\right)\right]$

$= n + 2\frac{n}{2} + 2^2 T\left(\frac{n}{2^2}\right)$

$= n + n + 2^2 T\left(\frac{n}{2^2}\right)$

$= 2n + 2^2 T\left(n/2^2\right)$

$T(n/2) = \frac{n}{2}$ $T(n)$

$= \frac{n}{2} + 2T$

$=$

$\frac{n}{2 \times 2}$

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

$2^k = n$

k^{th} term $= kn + 2^k T\left(n/2^k\right)$ ——— (*)

Assume that $n/2^k = 1$ $\Rightarrow$ $2^k = n \Rightarrow k = \log_2(n)$

(Take log both side)

$(\log k = \log n)$

$k = \log_2 n$

$T(n) = n\log_2(n) + nT(1)$

$= n\log_2(n) + n$

larger term

$2^k = n$

$\log 2^k = \log n$

$k \log 2 = \log n$

$k = \frac{\log n}{\log 2} = \log_2 n$

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

$= \log_2 n$

So $O(n\log_2(n))$

$T(n) = 3T\left(\frac{n}{4}\right) + n$

$T(n/4) = 3T\left(\frac{n}{16}\right) + \frac{n}{4}$

$T(n/16) = 3T\left(\frac{n}{64}\right) + \frac{n}{16}$

~~$= 3[3T$~~

(A) $T(n) = n + 3T(n/4)$

$= n + 3\left[3T\left(\frac{n}{16}\right) + \frac{n}{4}\right]$

$16, 4, 64, 2$

$= n + 9T[n/16] + 3n/4$

$= n + 9T[n/16] + 3n/4$

$= n + 9\left[3T(n/64) + n/16\right] + 3n/4$

$= n + \frac{3n}{4} + 27T(n/64) + \frac{9n}{16}$

$= n + \frac{3n}{4} + \frac{9n}{16} + 27T(n/64)$

$= n\left(1 + 3/4 + 9/16\right) + 27T(n/64)$

rearrange

$= 27T(n/64) + n\left(9/16 + 3/4 + 1\right)$

$= 3^3 T(n/4^3) + n\left[(3/4)^2 + 3/4 + 1\right]$

$\vdots$

k^{th} term $= 3^k T\left[n/4^k\right] + n\left[1 + 3/4 + (3/4)^2 + \cdots (3/4)^{k-1}\right]$

Assume $n/4^k = 1 \rightarrow 4^k = n \rightarrow k = \log_4(n)$

$a\log_c(b) = b\log_c(a)$

sum of infinite series $a + ar + ar^2 + ar^3 + \cdots \frac{a}{1-r}$ if $|r| < 1$

Let $n = 4^k$

Take log, $\log n = \log 4^k$

$k = \log_4 n$

$T(1) = 1$

Assume $n/4^k = 1$

$4^k = n$

$k = \log_4(n)$

$T(n) =$

$T(n) = 3^{\log_4 n \to k} T(1) + n\left(1 + 3/4 + (3/4)^2 + \cdots (3/4)^{k-1}\right)$

$= 3^{\log_4 n} T(1) + n\left(\frac{1}{1-3/4}\right)$

$= n^{\log_4 3} \cdot T(n) + 4n$

$= n^{\log_4 3} + 4n$

$= n + 4n = O(n) + O(n^{\log_4 3})$

Answer $= O(n)$

Biggest n (power)

$n^{\log_4 3}$

$\log_4 3 = x$

$4^x = 3$

$x \log 4 = \log 3$

$x = \frac{\log 3}{\log 4} = 0.7924$

GP =

$a = 1$

$r = 3/4$

$k = \log_4(n)$

$3^{\log_4 n} = n^{\log_4 3}$

$a^{\log(b)} \; c = b^{\log(a)}$ swap a & b

$n^{\log_4 3} = n^{\log_4 4}$

$\frac{n}{1-3/4}$

$1 - \frac{3}{4}$

$\frac{4-3}{4} = \frac{1}{4}$

$n(1/4) = \frac{1}{1/4} = 4$

$\log_{10} 100 = 2$

$10^2 = 100$

log

$\log_4^k = \log n$

$k =$

Recursion Tree

It is a pictorial representation of iteration method which is in the form of a tree where at each level, nodes are expanded
It is used to solve recurrence relations of divide and conquer Types of algorithms like merge sort

- Divides means the problem is divided into small parts
- conquer means joins the small parts into large.
- in recursion Tree, each node represent the cost of single subproblem.
- It will not work for ~~for~~ all recurrence relations.

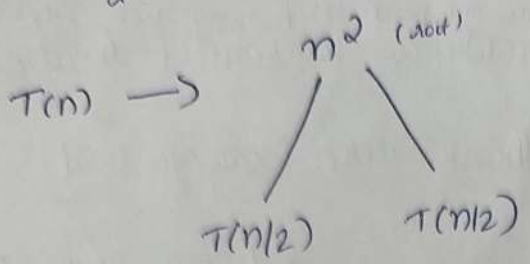
steps

- Draw a Recurrence tree
- calculate time taken by every level of tree
- finally we sum the work done at all levels

How to draw Recurrence tree?

1. Start From the given recurrence relation.
2. second term in recurrence relation become our root node.
3. keep drawing till we find a pattern among levels.
4. the pattern is typically a A.P or G.P

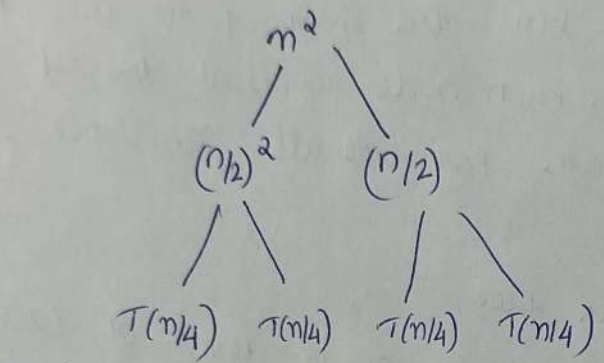
eg: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

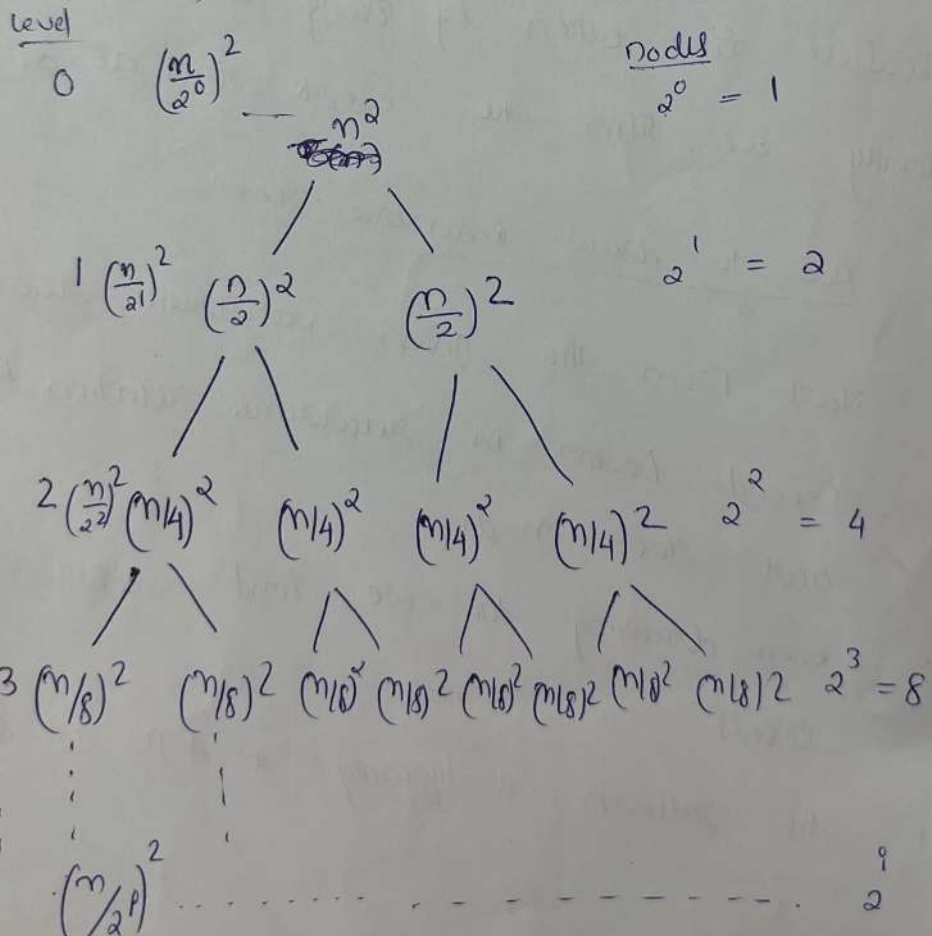


$T(n/2) = 2T(n/4) + \frac{n}{2}$

$T(n/4) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$

$\frac{n}{2^i}$





Sum of work done at each level.

$$n^2 + 2\left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{4}\right)^2 + 8\left(\frac{n}{8}\right)^2 + \cdots + \cdots$$

to find upper bound lets assume this series will go to infinite

$$n^2 + 2 \cdot \frac{n^2}{4} + 4 \frac{n^2}{16} + 8 \frac{n^2}{64} + \cdots$$

$$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots$$

$$n^2\left(1 + \underbrace{1/2 + 1/4 + 1/8 + \cdots}_{\text{GP}}\right)$$

$$\left(\text{GP} = \frac{a}{1-r}\right)$$

$$a = 1, \quad r = 1/2$$

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

$$\frac{1}{1} \quad \frac{1}{1-\frac{1}{2}} \quad \frac{2-1}{2} \quad \frac{1/2}{1/2}$$

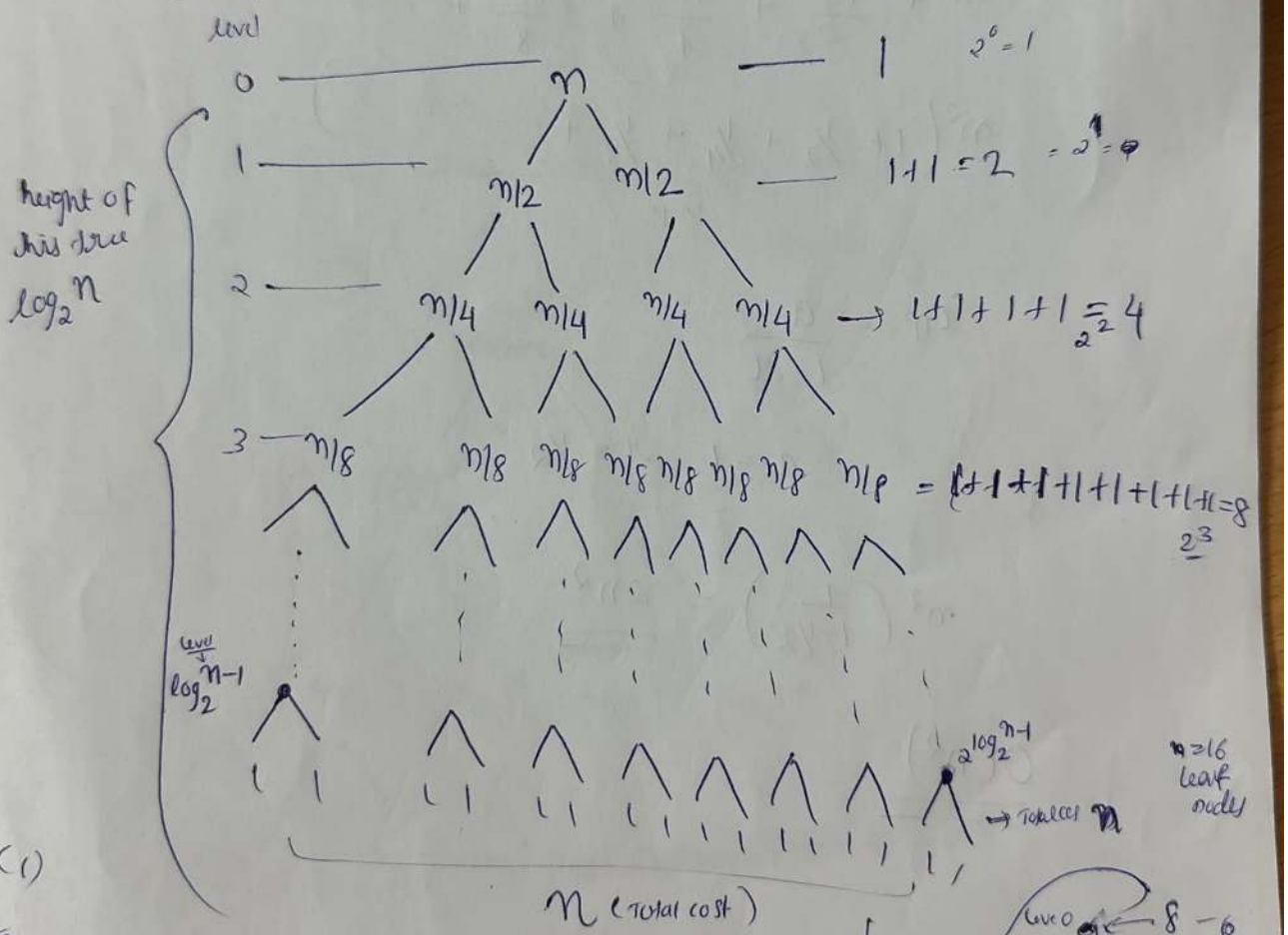
$$O(n^2)$$

* $T(n) = 2T(n/2)+1$, $T(1) = 1$ using Recursion Tree

[the problem of size n get divide into two subproblem of $\frac{n}{2}$ each, & cost per root is n.

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

⇒ problem of size n, divide into 2 subproblem of size n/2 each, & merging the soln, cost is 1 unit

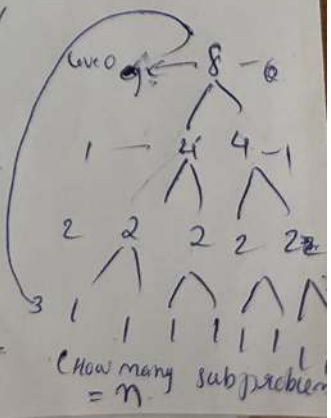


Total cost

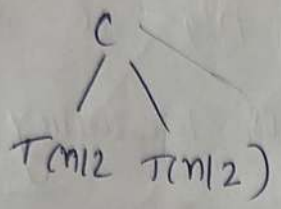
$$= \underbrace{2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log_2 n-1}}_{\log_2 n-1+1} + n$$

$$= 2^{\log_2 n} - 1 + n$$

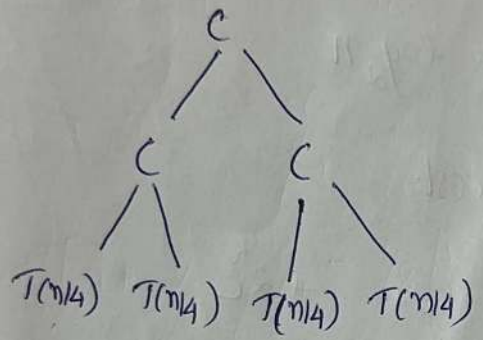
$$= n - 1 + n = 2n - 1 = O(n)$$



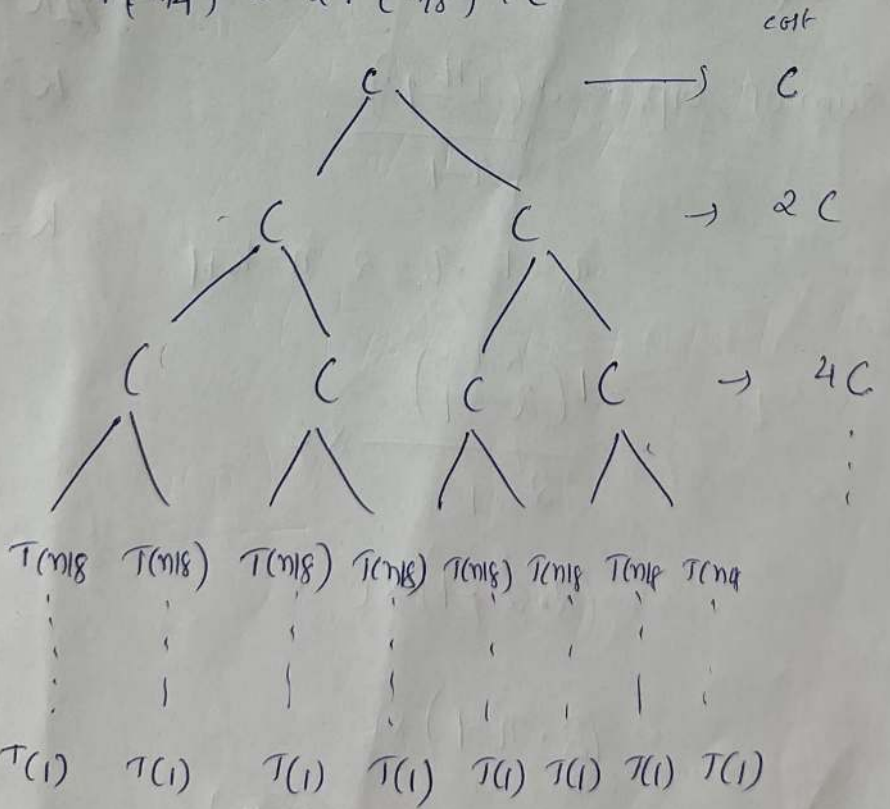
$T(n) = 2T(n/2) + C_{root}$



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



$T(n/4) = 2T(n/8) + C$



Now we have to find out the cost at each level and total no: of levels. the cost at level 0

c, 1st level 2c, 2nd level 3c, & 3rd 4c, ... and it goes on. At last level, it ends up at T(1) i.e n/2 at level 1, n/4 at level 2, n/8 at level 3. The denominator is increasing as the power 2. So at last level,

to get T(1), $n/2^k = 1$

$\Rightarrow k = \log_2 n$

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

$n = 2^k$

Take log both

$\log n = \log_2 k$

$\log_2 k = \log n$

$k \Rightarrow \frac{\log n}{\log 2}$

$k = \log_2 n$

Summing up the cost

$c + 2c + 4c + 8c + \cdots + 2^k c$

$c(1 + 2 + 4 + 8 + \cdots 2^k)$

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

$a = 1, r = 2, n = k+1$

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

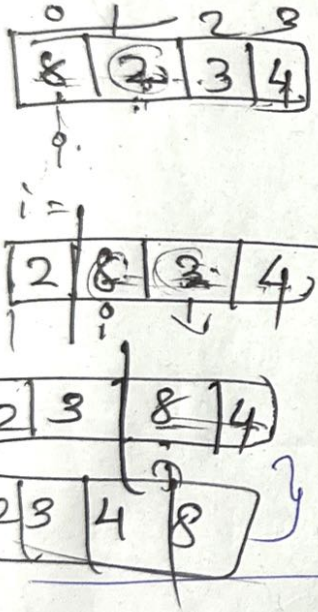
$= (2 \cdot 2^k - 1) \times c$

$= (2 \cdot n - 1) \times c$

$= 2n - 1 \times c = \underline{O(n)}$

Analysis of selection sort:-

```
For(i=0 ; i<n-1; i++) ⇒ n-1
{
  int min_index = i;  ⟶ n-1
  for (j = i+1; j<n; j++)
  {
    if (A[j] < A[min_index]) ⇒ n²
    {                          // comparison statement
      min_index = j;
    }
  }
  swap(A[i], A[min_index]); → n-1
}
```



⇒

i	j	no. of iteration
0	1–5	5 times
1	2–5	4 times
2	3–5	3 "
3	4–5	2
4	5–5	1 "
5		

n = 6

sum of first n natural no
0+1+2+3 - - - n - - -

$$\sum_{i=0}^{n-1} \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2}$$

$n + n-1 + n^2$

$= n + n-1 + n-1 + n^2$

$= 3n - 2 + n^2$

Total time unit $\} n-1 + n-1 + n^2$

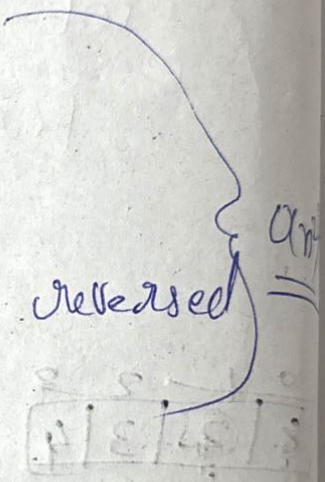
consider only n^2

$O(n^2)$

Best case : sorted array

Worst case : Reversed array

Average case : Neither sorted nor reversed



Best case (Ascending order)

10 20 30 40 50 60

min = 10

$n-1$
$n-2$
$n-3$
$n-4$
⋮
0

$0+1+2+3+\cdots+n-1$

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

⇒ here no swaping, ~~one~~ swap ⇒ $O(1)$

worst case (Descending order)

60 50 40 30 20 10

min = 60, swap 60 & 10 ⇒ $n-1$ comparison for 1 swap

10] 50 40 30 20 60

max = 50, there min = 20

swap 50 & 20 n-2 comparisons

also 00 : n-1 + n-2 + n-3 + --- 0

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

swap ⇒ O(n), compar = $O(n^2)$

Average case = $O(n^2)$ comparison

performance = O(n) swap.

Average case = $O(n^2)$