#Imp for Interview

Time And Space Complexity

Good Code : less time and less memory

*Topic is very important as it helps to write optimised code

Order Complexity Analysis

→ Amt of Space or Time taken up by an algorithm / code as function of input size.

*not the actual time taken

Just Relationship or function

#Two Ways

① experimentally * ~~hardware dependent~~

```
// before code
long start = System.currentTimeMillis();
// After the end of code
long end = System.currentTimeMillis();
Time taken = end - start
```

② Asymptotic Notation or theoretically

*hardware independent

(depend on the size of input)

Big. O notation → Worst Case Scenario

Ω (Omega) → Best Case Scenario

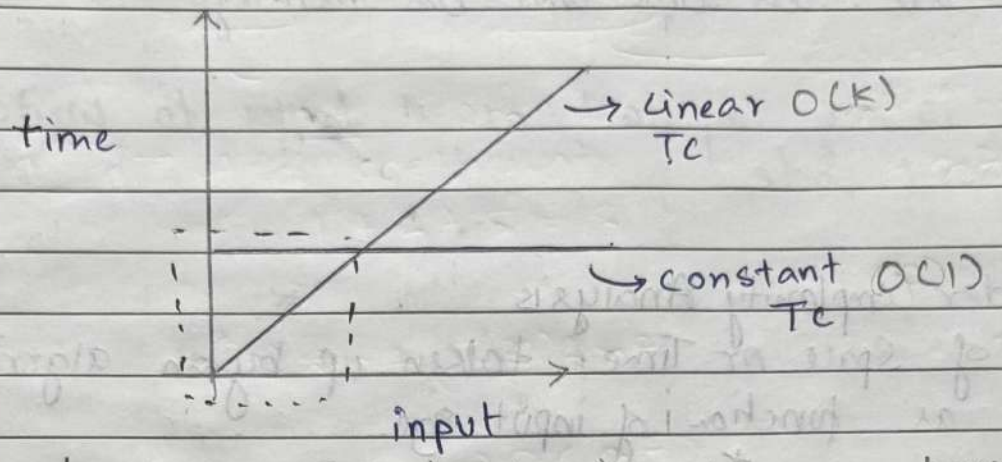
Θ (Theta) → Average.

Note: We always try to find worst case complexity

thus, we will use only

Big. O notation

There might be a possibility,



for a small input size we may have
constant TC > linear TC

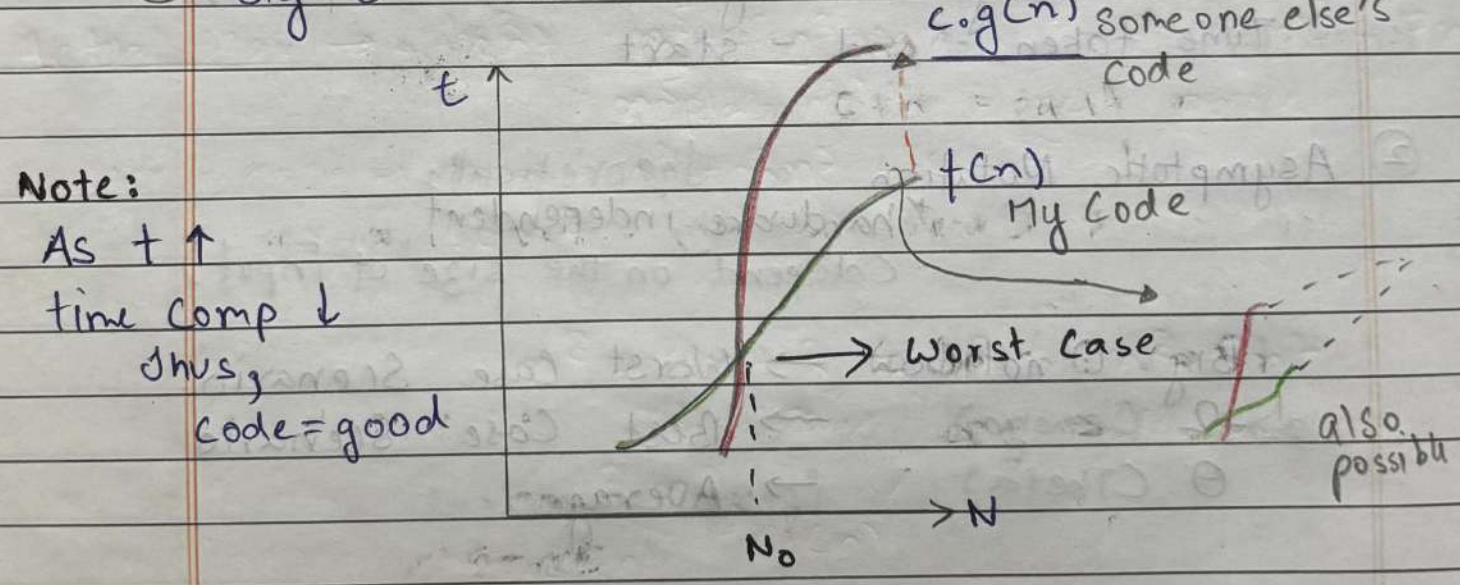
But, we always analyze for worst case

input size is very much

#Concept*

① Big O

aka upper bond



Note:
As t ↑
time comp ↓
thus,
code = good

Our functions time

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

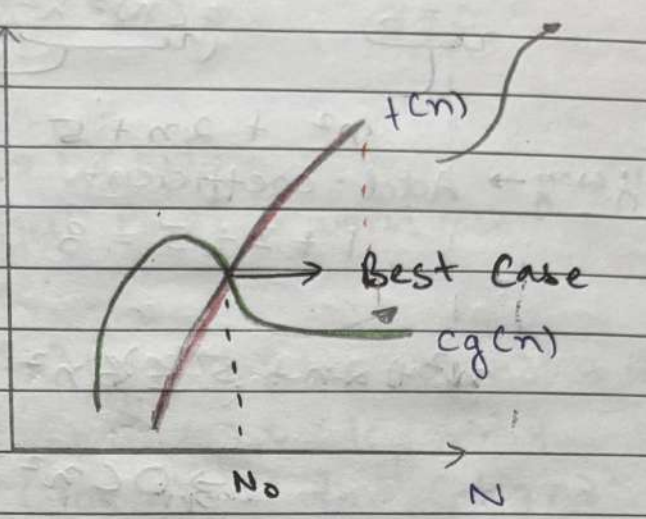
c.g(n) will always take more time then t(n)

$$n \geq n_0$$

Worst case time complexity

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

② Big Ω



Note:
As t ↑
time comp ↑
code → not good

$$F(N) \geq C.g(N) \qquad N \geq N_0$$

Best case time complexity

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

③ Θ (useless)

Examples → find Worst case scenario

- $f(n) = n+2$

↳ let $C = 3$ (don't know why $C = 3$) ∴ C = constant

$g(n) = n$

$F(n) \leq C.g(n)$

$n+2 \leq 3n$ ∴ $n \geq 1$

ignore coefficient

$3n \rightarrow n$

↳ $O(n)$

• $f(n) = n^2 + 2n + 5$

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

$c = 8$ $\quad g(n) = n^2$

Biggest n^{power} or value $= n^2$

$n^2 + 2n + 5$

easy way → Add coefficient

$1 + 2 + 5 = 8$

$n^2 + 2n + 5 \leq 8n^2 \quad \because n \geq 1$

↳ $O(n^2)$

• $f(n) = n^4 + 5n^3 + 6n^2 + 7n + 4$

Find biggest power → n^4

↳ $O(n^4)$

Actual Way → • $n^3 + 4n + 8$

replace with biggest power

worst case = $n^3 + 4n^3 + 8n^3$

$= 13n^3$

$n^3 + 4n + 8 \leq n^3 + 4n^3 + 8n^2$

will always be smaller

$13 n^3$

constant $g(n)$

↳ $O(n^3)$

$n > \log$
$32 > \log_2 b$ Date 32
$5 \ldots$ Page:

- $f(n) = n^7 + 5n^5 + 6\log(n) + 7n^3 + 19$
  $\rightarrow 1 + 5 + 6 + 7 + 19$
  $C \rightarrow 38$
  biggest P $\rightarrow N^7$
  $\rightarrow O(N^7)$

∴ In MCQ, just biggest Power = worst case

- $f(n) = n^5 + n^3 \log n + 9n^5 (\log n)^2 + 7n$
  $= n^5 (\log n)^2 + n^5 (\log n)^2 + 9n^5 (\log n)^2 + 7n^5 (\log n)^2$
  $= 18\, n^5 (\log n)^2$
  $O(18 n^5 (\log n)^2)$ ∴ $n \geq 1$

Interview

Syso("Hi"); Print → constant time O(1)
Mathematical Operation → constant time O(1)
Linear Search → O(n)

for(i=0; i<5; i++) P.T.O O(1)
for(i=0; i<n; i++) P.T.O O(n) n varries

Operations | Binary Search

$1 - N$
$1 - \frac{N}{2^1}$
$1 - \frac{N}{2^2}$
$\vdots$
$\frac{N}{2^K} = 1$ } K steps

$\frac{N}{2^K} = 1 \rightarrow 2^K = N$

$\boxed{K = \log_2 N}$

3 Rules → • worst case scenario
• avoid constants
• avoid lower values.

Date: _____
Page: _____

- for(int i = 1; i <= 5; i++){ }

worst scenario = O(5)

constant

⇓

O(1)

- for(int i = 0; i < n; i++){ }

Assume,

We are having constant work — k

i = 0 → k
i = 1 → k
i = 2 → k
⋮
i = n-1 → k

→ n × k

Worst case ⇒ O(n × k)

constant

↓

O(n)

- for(int i = 1; i < n; i++){
  for(int j = 1; j <= n; j++){ } }

Types : • Is outer loop depend upon inner loop?

No!

k operations

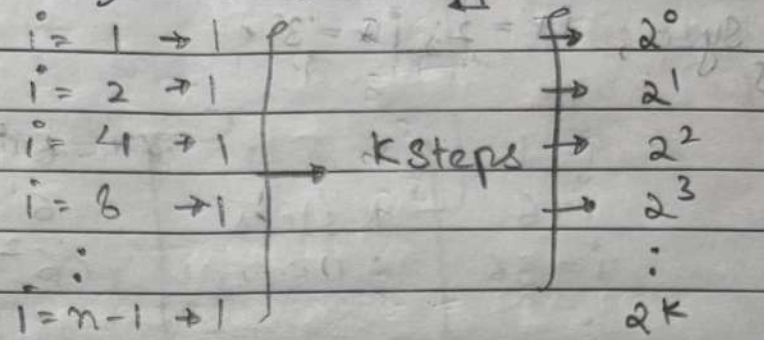
O(n)

O(n)

* Multiply outer loop complexity w. inner loop = O(n) * O(n)

= O(n²)

- for (int i = 1; i = i * 2; i < n) { }

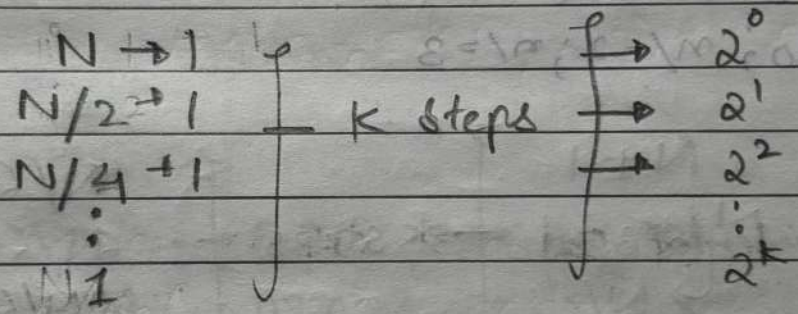


$\longrightarrow 2^k = n$

$k = \log_2 n$

$n > 0 \quad n /= 2$; i++

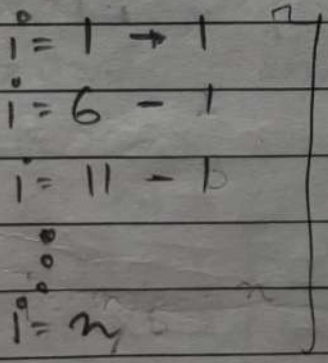
- for (i = 1; ;) { }



$\longrightarrow 2^k = n$

$k = \log_2 n$

- while (i <= n) {
  Syso; i+=2; i+=3;
  }



$\rightarrow O(n/5)$

$n/5$ steps $\rightarrow O(n)$ constant

- while (i<=n){
  Syso; i*=2; i*=3
  }

i = 1	k steps	6^0
i = 6		6^1
i = 36		6^2
⋮		⋮
N = i = 6^k		6^k

→ $6^k = N$

→ $k = \log_6 N$

- while (n>0){
  Syso; n/=2; n/=3
  }

N+1	k steps	$N/6^0$
N/2*3-1		$N/6^1$
↳ N/6		$N/6^2$
N/36-1		$N/6^3$
⋮		⋮
1		$N/6^k$

$\frac{N}{6^k} = 1$

$N = 6^k$

→ $k = \log_6 N$

Date:
Page:

- int k = 5
  while (i <= n) {
  Syso ; i += k
  }

i = 0 → 1
i = 5 → 1
i = 10 → 1
i = 15 → 1
⋮
i = n

i += k, k = 5

loop will run 5 times

$\frac{N}{5}$ or $\frac{N}{k}$

→ O(N/k)
constant
→ O(n)

let, k = 2

- while (i <= n) {
  Syso ; i *= k ;
  }

i = 1
i = k x 1
i = k x k
i = k^3
⋮
i = n

k times

k^0
k^1
k^2
k^3
⋮
k^k

→ $k^k = n$
→ $k = \log_k n$

- while (n > 0) {
  Syso ; n = n - 2 ; n = n - 3
  }

PTO same question

N → 1
N - 5 - 1
N - 10 - 1
⋮
1

N - (5 x k)
constant
↳ O(n)

- while (n > 0) {
  n = n - k
  }

same as i += k at top

N
N - 1
N - 2
⋮
1

N - (k)
constant
↳ O(n)

- for (i=1; i*i <= n; i++) {

i = 1	1
i = 2	4
i = 3	9
i = 4	16
⋮	⋮
i = N	

$i * i \leq n$

$i^2 \leq n$

loop Run till

$i \geq \sqrt{n}$

↳ $O(\sqrt{n})$

- for (i=1; i<=n; i++) {
  for (j=i+1; j<=n; j++) { }

#2nd way → total operations runs

i = 1	n − 1
i = 2	n − 2
i = 3	n − 3
⋮	⋮
i = n	0

Operations

$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 + 0$ — Arithmetic Progression

0 to n $\quad \frac{n(n+1)}{2}$

0 to n−1 $= \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$

↳ $O\left(\frac{n^2}{2} - \frac{n}{2}\right)$ — largest term; constant

↳ $O(n^2)$

- for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++) { ; } → k = constant work

  i = 0 to n-1 → n times
  j = 0 to i-1

i = 0	0 times		
i = 1	1×	j = 0 to 0	k = k
i = 2	2×	j = 0 to 1	k = 2k
i = 3	3×	j = 0 to 2	k = 3k
⋮	⋮	⋮	⋮
i = n-1	(n-1)	j = 0 to (n-1)	k = (n-1)k

→ k + 2k + 3k .. (n-1)k
→ k (1 + 2 + 3 .. (n-1)) (AP)

→ k (n(n-1)/2) = (kn² - kn)/2 = k/2 (n² - n) (largest; k/2 constant)

= O(n²)

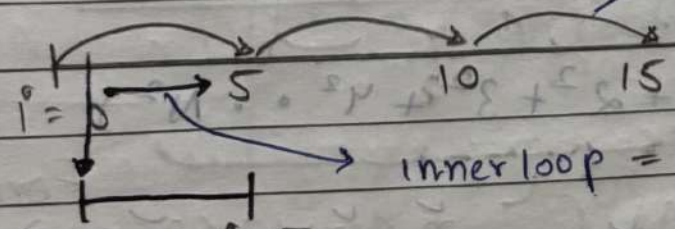
Assume k - constant when k < n

- for (i = 0; i < n; i = i + k) {
  for (j = i + 1; j <= k; j++) { ; } → p work

Lets consider,
n = 50 k = 5

n/k

// Outer loop



// inner loop

inner loop = k

= O(n/k × k × p) (k × p constant)

= O(n)

- ```
  for(i=1;i<=n;i++){
    for(j=1;j<=i;j++){
      for(k=1000;k<=1000;k++){
  ```

	j	k
i = 1	1 × 1000	
i = 2	2 × 1000	+ 1000(1+2+3...n)
i = 3	3 × 1000	1000.(N(N+1)/2)
:	:	
i = n	n × 1000	v. big no so we will count it; N² can ignore but big no.

→ O(1000.(N²))

- ```
  for(i=1;i<=n;i++){
    for(j=1;j<=i*i;j++){
      for(k=1;k<=n/2;k++){
  ```

# Operations (times loop will run)	j	k
i = 1	1 (1²)	n/2
i = 2	4 (2²)	4n/2
i = 3	9 (3²)	9n/2
:	:	:
n	n²	n² × n/2

Add all

$\frac{N}{2}(1+2^2+3^2+4^2 \cdots N^2)$

Formula:

$\frac{N}{2}\left[\frac{n(n+1)(2n+1)}{6}\right]$

↳ $O(N^4)$

- ```
  for (i = n/2; i <= n; i++){
    for (j = 1; j <= n/2; j++){
      for (k = 1; k <= n; k = k*2){
        // some O(1) expressions
      }
    }
  }
  ```

* Here every loop is independent of each other

$$\frac{n}{2} \times \frac{n}{2} \times (\log_2 n)$$

↳ $O(n^2 \log_2 n)$

- ```
  for (i = 1; i <= n; i++){
    for (j = 1; j <= n; j += i){
      // some O(1) expressions
    }
  }
  ```

$i = 1$	→	n
$i = 2$	→	$n/2$
$i = 3$	→	$n/3$
$i = 4$	→	$n/4$
⋮		⋮
$i = N$		n/n

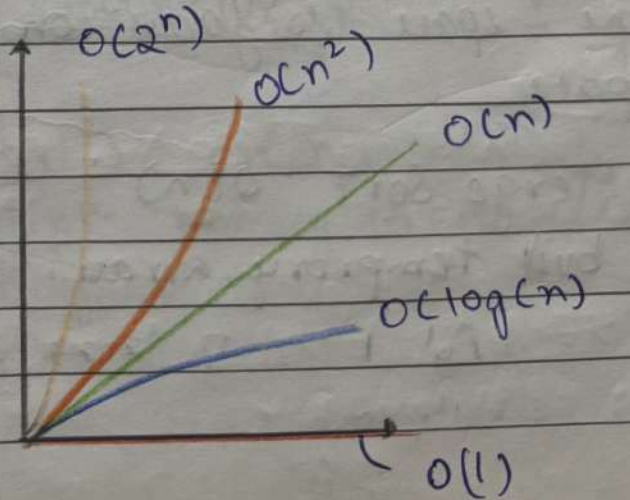
network

$$n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \ldots\right)$$ Harmonic Progression

aka $\log n$

↳ $n \log n$

Common Complexities



Recurssive Relation

Two type will be observed
- ↳ Linear → $f(n) = k + f(n-1)$
- ↳ D&C → merge sort

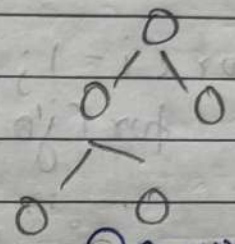
$f(n) = f(n/2) + f(n/2) + k_{merge}$

Time:

Method
1. Total work done = (no of calls * work in each call)
2. Recurrence equation

Space:
→ (max depth * memory in each cell)

Visualise in tree form



@RewrssioTree

↳ Space Complexity
- ↳ heap (Object)
- ↳ stack (C functions)

input space + auxiliary space
(temperory)

Interview : Not very Important unless asked

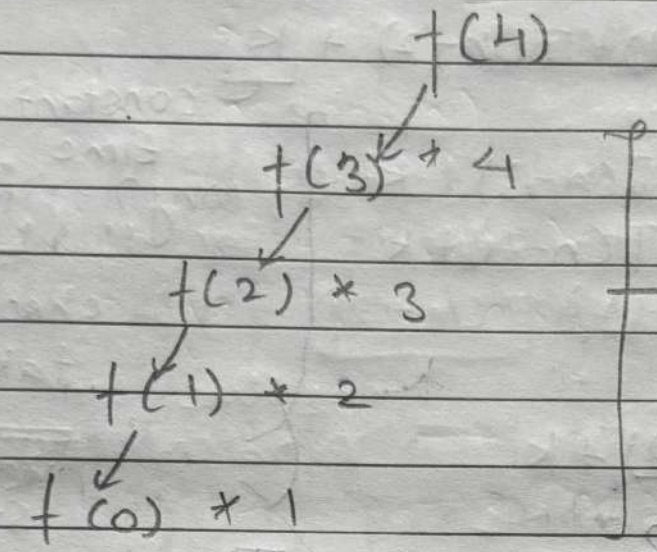
(. We won't be adding input space as by default every program will take some space so focus on auxilary space

↳ Merge sort O(n)
built temprory array

- factorial

```
public static int fact(n){
  if(n==0){ ] constant work
    return 1; ] (k1)
  return n * fact(n-1); } constant work (k2)
}
```

Method 1
→ no of calls * work
→ n * k
→ nk
↳ O(n)
TC.

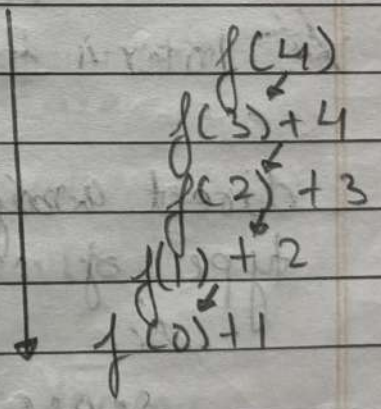


S.C : max depth * memory in each call
height of tree
↳ n * k — constant memory is being used
We haven't use any other data structure.
↳ O(n)

- Sum of n

```
static int sum(int n){
  if(n==0){ ] k1
    return 0; ]
  return n + sum(n-1); } k2
}
```

n calls
k



work : k
TC: O(nk) = O(n)
SC: Max depth * memory in each call
→ n * k
→ O(n)

Date:
Page:

Divide and Conquer

- Fibonacci

```
static int fib(n){
if(n==0 || n==1){
 return n;
}return f(n-1) + f(n-2);
}
```

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

Recursive equation / Relation

Time C for n fibonacci

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

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

$T(n-2) = T(n-3) + T(n-4) + k$

$T(2) = \underbrace{T(1)}_{k_1} + \underbrace{T(0)}_{k_2} + k$

$T(1)$ time, constant

k constant time

Why? if statement: checking will take time; return statement

∴ This time won't be big does not depend upon input size

$O(1)$

to find TC, we will use Master's Theorem.

↳ formula for solving recurrence relations of the form

direct way to get a solution. Work only for a type of recurrences:

$$T(n) = aT\left(\frac{n}{b}\right) + \underbrace{\Theta(n^k \log^p n)}_{f(n)}$$

- $a >= 1$; $b > 1$; $k >= 0$
  p: real number

@4t 5 minute Engineering

Date:
Page:

Cases

Case 1: if $a > b^k$
then
$$T(n) = \theta(n^{\log_b^a})$$

Case 2: if $a = b^k$

a) if $p > -1$ then $T(n) = \theta(n^{\log_b^a} \log^{p+1} n)$

b) if $p = -1$ then $T(n) = \theta(n^{\log_b^a} \log\log n)$

c) if $p < -1$ then $T(n) = \theta(n^{\log_b^a})$

Case 3: if $a \leq b^k$

a) if $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

b) if $p < 0$ then $T(n) = O(n^k)$

examples

① $T(n) = 4T(n/2) + n$
$a = 4; b = 2; k = 1; p = 0$
Satisfying all condition
Now lie in which case

case 1: $T(n) = \theta(n^{\log_2^4})$
$= \theta(n^2)$

② $T(n) = 2T(n/2) + n\log n.$
$a = 2; b = 2; k = 1; p = 1$
Satisfying

case 2: $2 = 2^1; p = 1$
category a $T(n) = \theta(n^{\log_2^2} \log^2 n)$
$= \theta(n \log^2 n)$

③ $T(n) = 2T(\frac{n}{2}) + n^2$

$a = 2 \quad b = 2 \quad k = 2 \quad p = 0$

Case 3: $a < b^k ; p = 0$

category 1

$T(n) = \theta(n^2 \log^0 n)$

$= \theta(n^2)$

More simplified & jugaad way,

Now, for fibonacci series,

As it's hard to solve it further for

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

as this will create problem

To solve that,

We will use just ~~append~~ upper bound.

$T(n-1) + T(n-2) + k \leq T(n-1) + T(n-1) + 1$

we will take its time complexity

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

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

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

$\vdots \quad \vdots \quad \vdots$

$T(1) = 1 \quad 2^{N-1}$

To cut these equation, we need to ~~use~~ multiply equate $T(n-1)$ with 2 to equate with $2T(n-1) + k$

∴ Geometric Progression $\frac{a*(1-r^n)}{1-r}$ $r<1$ $\frac{a(r^n-1)}{r-1}$ $r>1$

Date: Page: 1

And thus

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

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

→ $T(n) = 2T(n-1)+k$

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

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

$\times\ T(1) = 1\times 2^{N-1}$

$T(n) = k(1+2+2^2+2^3 \cdots 2^{N-1})$

G.P. Geometrical progression

$T(n) = \frac{a(r^n-1)}{r-1}$

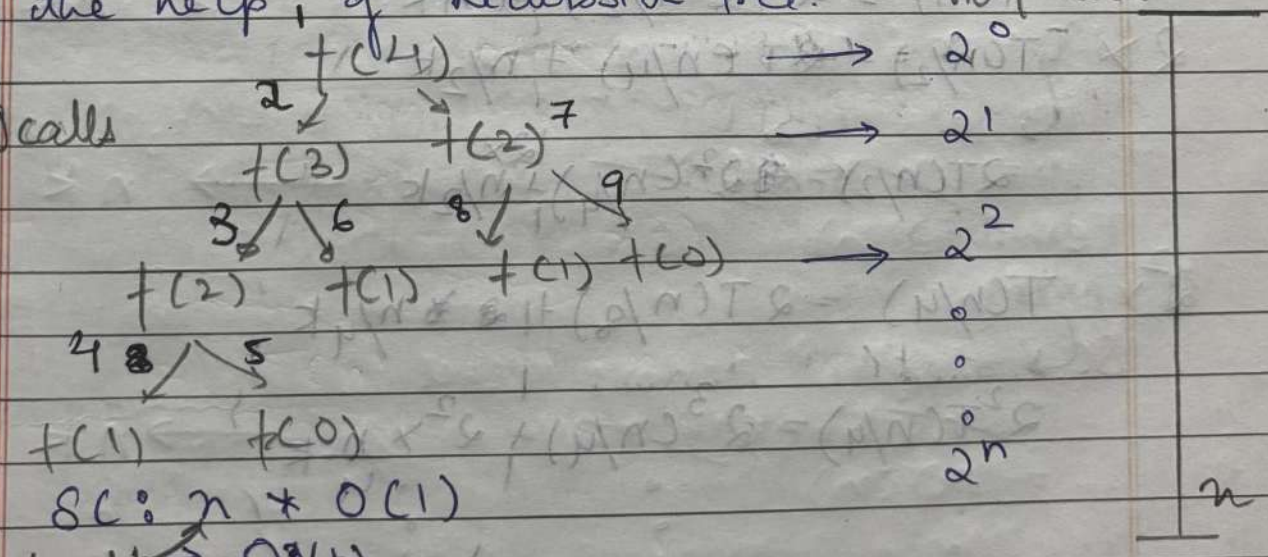
$= \frac{1(2^n-1)}{2-1} = 2^n$

⇒ $O(2^n)$

With the help of Recursion tree.

n=4

total = 9 calls



SC: $n * O(1)$

depth → $O(n)$

TC: $2^n * O(1)$

: $O(2^n)$

$2^0+2^1+2^2 \cdots 2^n$

1C

* if 2 recursive call → 2^n
  3 recursive call → 3^n
  dice problem → 6^n
* String → $2^n \times n$ (for concatination)
subsequence

• Merge Sort

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

```
void mergesort(
if(si >= ei){ k2
  return; }
mid = si+(ei-si)/2; k2
mergsort(arr,si,mid); T(n/2)
mergesort(arr,mid+1,ei);
merge(arr,si,mid,ei); O(n)
```

$T(n) = 2T(n/2) + nk \longrightarrow nk$

$2\times$ $T(n/2) = 2T(n/4) + n/2\,k$

$2T(n/2) = 2^2(n/4) + n/2\,k \longrightarrow nk$ (2×)

$2^2\times$ $T(n/4) = 2T(n/8) + n/4\,k$

$2^2T(n/4) = 2^3(n/4) + 2^2 \times \frac{n}{4}k \longrightarrow nk$

$T(1) = O(1) \longrightarrow nk$

$\longrightarrow n$

Similarly like fibonacci.
↳ $T(n) = O(1) + n(nk)$ next page

$n \rightarrow \frac{n}{2^0}$

$\downarrow$

$n/2 \rightarrow \frac{n}{2^1}$

$\downarrow$

$n/4 \rightarrow \frac{n}{2^2}$

$\downarrow$

$n/8 \rightarrow \frac{n}{23}$

$\downarrow$

$\vdots \quad \vdots$

$1 \rightarrow \frac{n}{2^n}$

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

$n = 2^n$

$\boxed{\log_2 n = n}$

$T(n) = \underbrace{O(1)}_{\text{constant}} + \underbrace{(\log n)(n k)}_{\text{majority term}}$ constant

$\log n \times n$

by default base is 2

$= n \log n$

- Quick sort

  best case $n \log n$

  Worst Case n^2 (sorted Array)

i. Power

↳ Function / Analysis

```
int power (a, n){
 if (n == 0){ }
   return 1
return a* power (a, n-1);
}
```

WD = no of calls * time in each call

= n * k(1)

TC = O(n)

SC = depth * memory in each

= n * O(1)

= O(n)

f(a, n) a^n

↓

a * f(a, n-1) a^{n-1}

↓

a * f(a, n-2) a^{n-2}

↓

a * f(a, n-3) a^{n-3}

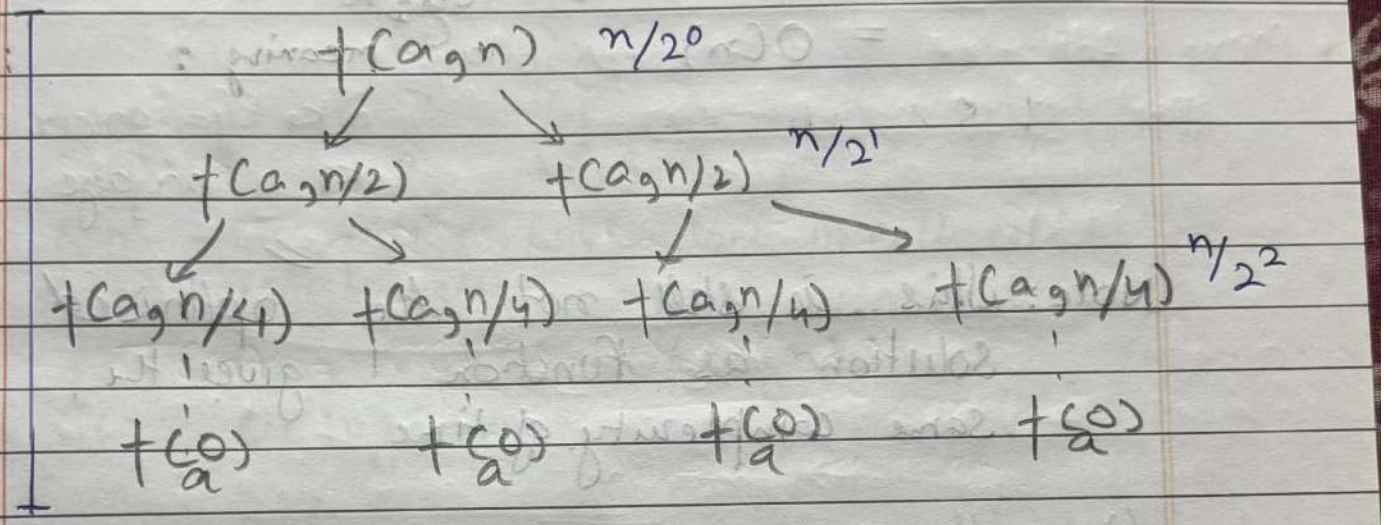
⋮

(1) f(a, 0) a^0

↳ Function 2 Analysis

```
if (n==0)2
    { return 1; }
int halfPowerSq = power2(a,n/2) * power2(a,n/2);
if (n%2 != 0)2 // a is odd
    { return a * halfPowerSq;
return halfPowerSq;
}
```

TC:



$\Rightarrow \frac{n}{2^n} = 1 \rightarrow n = \log_2 n$ depth

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

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

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

$2 \times (T(n/2) = 2 \times T(n/2) + k)$

$\vdots \quad \vdots \quad \vdots \quad \vdots$

$T(1) = 2.T(0) + k$

replace by a constant k_2

k	$k \times 2^0$
$2k$	$k \times 2^1$
$4k$	$k \times 2^2$
$8k$	$k \times 2^3$
$2^{\log n}k$	$k \times 2^{\log n}$

depth

$T(n) = 2k_2 + k(2^0 + 2^1 \cdots 2^{\log n})$

Geometric Progression

Date:
Page:

$$T(n) = \underbrace{2k_2}_{\text{constant } k_3} + k\left(\frac{2(2^{\log n} - 1)}{(2-1)}\right)$$

$$= k_3 + k(2^{\log n + 1} - 2)$$

$$= k_3 + k \cdot 2^{\log n + 1} - 2k$$

$$= \underbrace{k_3}_{\text{constant}} + k \cdot 2^{\log n + 1} - 2k$$

$$= O(2^{\log_2 n}) \quad \rightarrow \log_2 n$$

$$= O(n)$$

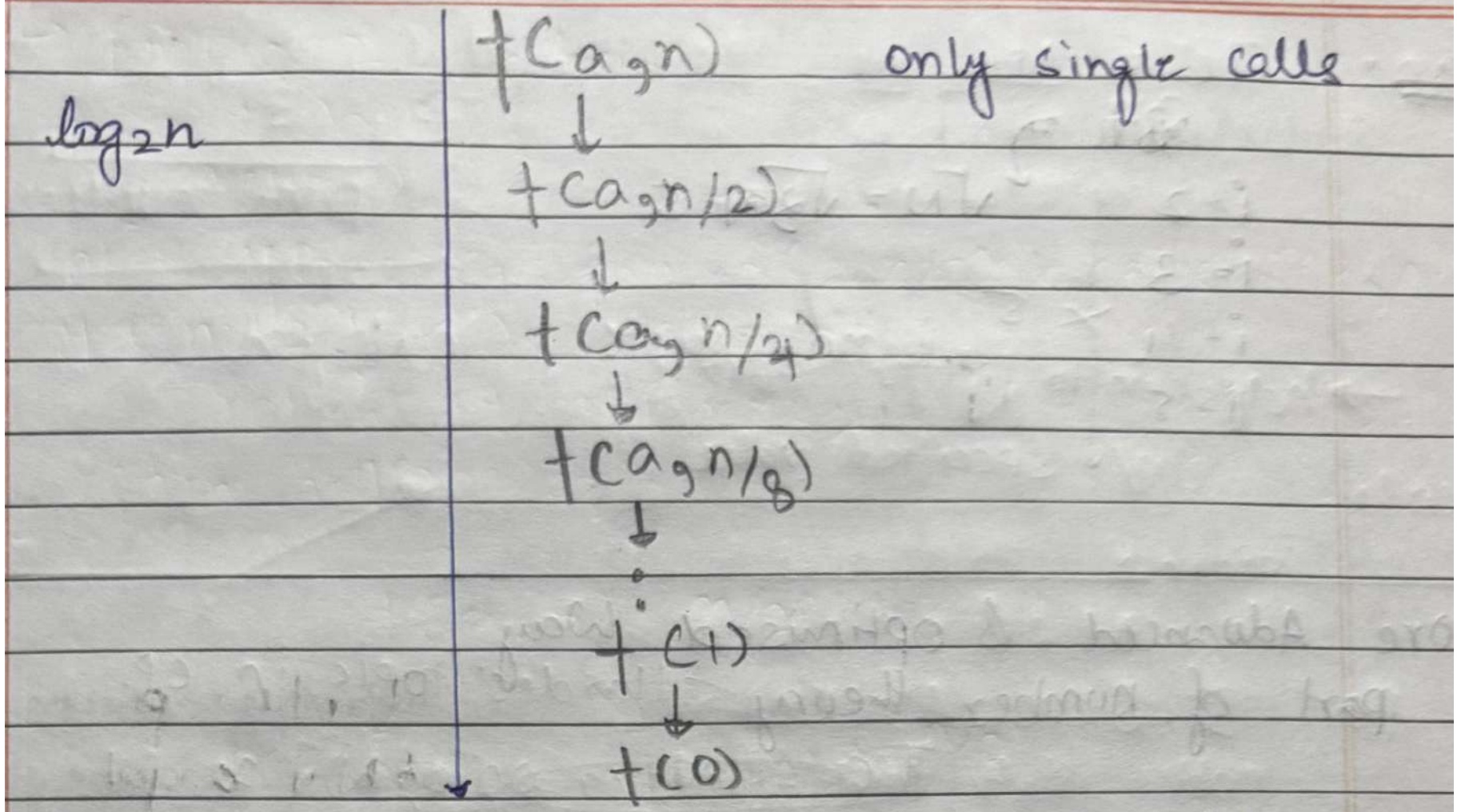
meaning :
2 ki kya power rakhu
ki n aajaye
=

Note: This is still not the optimised solution as function 1 gives the same complexity of time

We will store value

↳ function 3 Analysis

```
pubic power3 (int a, int n){
  if (n == 0){
    return 1;
  }
  int halfPower = power3(a, n/2);
  int halfPowerSq = halfPower * halfPower;
  if (n % 2 != 0){ // n is odd
    return a * halfPowerSq;
  }
  return halfPowerSq;
}
```



WD = total calls * work done in each call.
(total levels)

TC = $\log n \times K = O(\log n)$

SC = $O(\log n)$