2023

Java

DSA
PART-0

CYBER TRON

JAVA

(THIRD PART)

T.I.M. HAMIM

2. Time and Space Complexity

Time complexity is a measure of how the runtime of an algorithm or a program grows as the input size increases. It provides an estimate of the amount of time an algorithm takes to run as a function of the size of the input data. Time complexity is usually expressed using Big O notation, which describes an upper bound on the growth rate of an algorithm's runtime relative to the input size.

Time complexity != Time taken

The mathematical function is known as time complexity which tells us how the time is going to grow as the input grows.

What do we consider when thinking about complexity :

- Always look for worst-case complexity.
- Always look at complexity for large/infinite data.
- Even though the value of actual time is different they are all going linearly.
- We don't need to care about actual time.
- We have to ignore all constants.
- Always ignore less dominating terms.

Asymptotic notation provides a simplified way to express how the runtime of an algorithm grows as the input size increases. The three main notations are Big O, Big Omega, and Big Theta. Here's a brief explanation of each:

1. **Big O Notation (O):** This notation gives an upper bound on the growth rate of an algorithm's runtime. It provides the worst-case scenario for the algorithm's performance. For example, if an algorithm has a time complexity of O(n), it means the algorithm's runtime grows linearly with the input size.
2. **Big Omega Notation (Ω):** This notation provides a lower bound on the growth rate of an algorithm's runtime. It gives the best-case scenario for the algorithm's performance. For example, if an algorithm has a time complexity of Ω(n^2), it means the algorithm's runtime is at least quadratic in relation to the input size.
3. **Big Theta Notation (Θ):** This notation provides both an upper and a lower bound on the growth rate of an algorithm's runtime. It gives a tight range for the algorithm's performance. For example, if an algorithm has a time complexity of Θ(n), it means the algorithm's runtime grows linearly with the input size, and both the best and worst cases have the same growth rate.

Big O notation:

- Wordly definition :
  O (n^3) ---> Upper bound

- Maths :
  f(n) = O(f(n))

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

=>

O(n^3) = O(6n^3 + 3n + 5)
g(n) f(n)

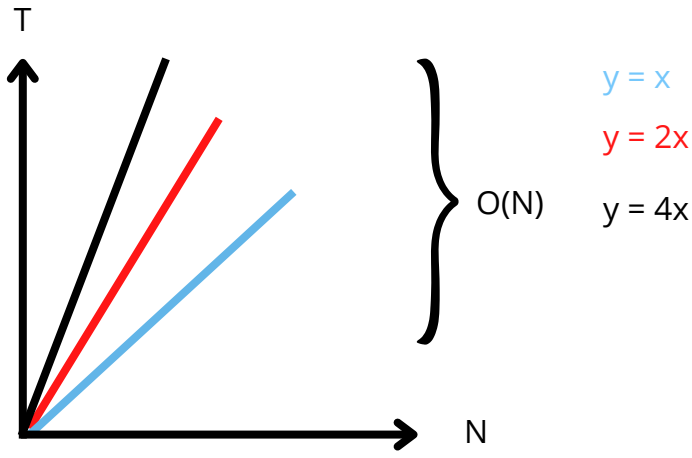
$$= \lim_{n \to \infty} \frac{6n^3 + 3n + 5}{n^3}$$

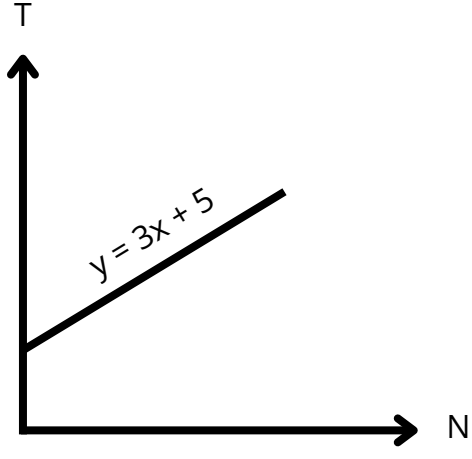
$$= \lim_{n \to \infty} 6 + \frac{3}{n^2} + \frac{5}{n^3} = 6 + 3/\infty + 5/\infty$$

$$= 6 + 0 + 0$$

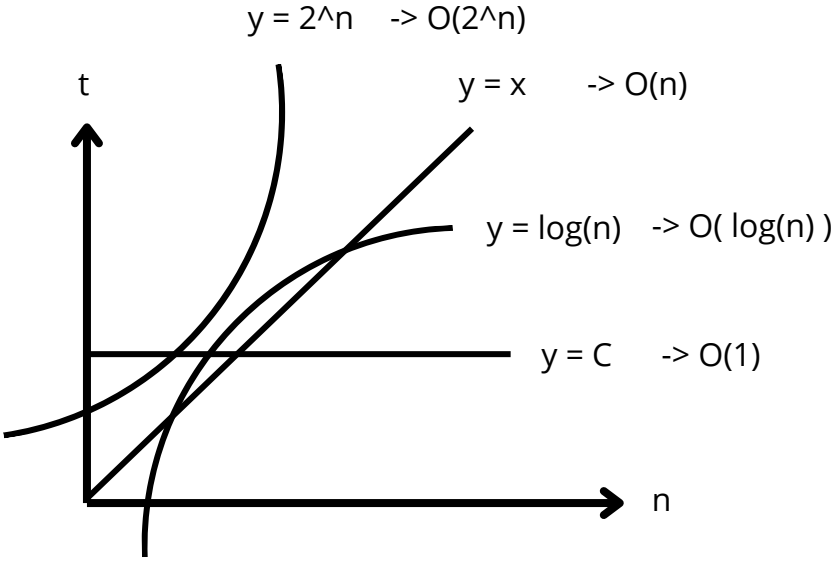
$$= 6 < \infty$$

Example :
O(3n^3 + 4n^2 + 5n + 6)
= O(n^3 + n^2 + n)
= O(n^3)





Here,
T = Time
N = Number of statements



Here,
O(1) < O(log(n)) < O(n) < O(2^n)

Big Omega notation:

Opposite of Big oh.

- Wordly definition :
  Ω(n^3) ---> Lower bound

- Maths :

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

Big Theta notation:

Combining the upper bound and lower bound.

- Wordly definition :
  Θ(n^2) ---> Both upper bound and lower bound is = n^2

- Maths :

$$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

In Big O notation, the time complexity of an algorithm is typically represented as a function of "n," where "n" is the size of the input. Here are some common types of time complexities and their explanations:

- **O(1) - Constant Time:** The algorithm's runtime remains constant regardless of the input size. Example: accessing an element in an array by index.

- **O(log n) - Logarithmic Time:** The runtime grows logarithmically as the input size increases. Algorithms with this complexity often divide the input in half with each step. Example: binary search.

- **O(n) - Linear Time:** The runtime grows linearly with the input size. If the input doubles, the runtime also roughly doubles. Example: iterating through an array.

- **O(n log n) - Linearithmic Time:** Common in efficient sorting algorithms like mergesort and heapsort. The runtime grows slightly more than linearly with the input size.

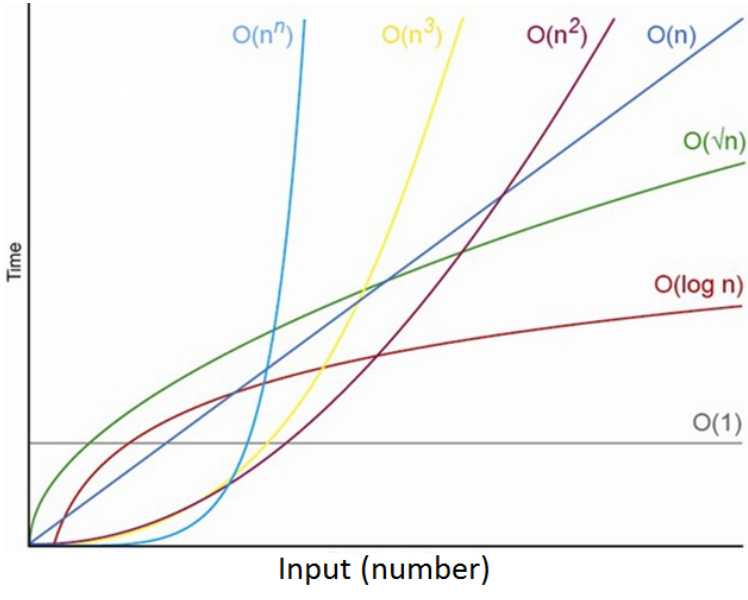
- **O(n^2) - Quadratic Time:** The runtime is proportional to the square of the input size. Nested loops are a common cause of this complexity. Example: bubble sort.

- **O(n^k) - Polynomial Time:** Similar to quadratic time, but with a higher exponent "k." Example: matrix multiplication using the naïve approach.

- **O(2^n) - Exponential Time:** The runtime grows exponentially with the input size. These algorithms often explore all possible combinations. Example: solving the traveling salesman problem with brute force.

- **O(n!) - Factorial Time:** The slowest growing time complexity, where the runtime increases rapidly with the input size. Example: brute force algorithms for permutation problems.

Rate of Growth



Time Complexity

(1)

```
for(i=0; i<n; i++)
{
        statement; ⟶ n
}
```

Time complexity : O(n)

(2)

```
for (i=n;  i>0; i--)
{
        statement; ⟶ n
}
```

Time complexity : O(n)

(3)

```
for (i=1; i<n ; i = i+2)
{
        statement ; ⟶ n/2
}
```

Time complexity : $O\left(\frac{n}{2}\right)$

$= O(n)$

(4)

```
for (i=1, i<n;  i = i+20)
{
        statement ; ⟶ n/20
}
```

Time complexity : $O\left(\frac{n}{20}\right)$

$= O(n)$

(5)

```
for (i = 0; i<n; i++)  ⟶ n+1
{
    for (j = 0; j<n; j++)  ⟶ n x (n+1)
    {
        statement;  ⟶ n x n
                    = n^2
    }
}
```

Time Complexity : $O(n^2)$

(6)

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

i	j	no of time
0	0 ×	0
1	0	1
	1 ×	
2	0	2
	1	
	2 ×	
3	0	3
	1	
	2	
	3 ×	
⋮		⋮
n		n

Total time of execute:

$$1 + 2 + 3 + \ldots\ldots + n = \frac{n(n+1)}{2}$$

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

∴ Time Complexity: $O(n^2)$

(7)

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

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3
4	1+2+3+4
⋮	⋮
k	1+2+3+4+....+k

We, Assume that,

$$p > n$$

$$\because p = \frac{k(k+1)}{2}$$

$$\therefore \frac{k(k+1)}{2} > n$$

$$\Rightarrow k^2 > n$$

$$\Rightarrow k > \sqrt{n}$$

$\therefore$ Time complexity : $O(\sqrt{n})$

(8)

```
for(i=1; i<n; i=i*2)
{   statement;
}
```

i
1
$1 \times 2 = 2$
$2 \times 2 = 2^2$
$2^2 \times 2 = 2^3$
⋮
2^k

Assume $i >= n$

$\because i = 2^k$

$\therefore 2^k >= n$

$\Rightarrow 2^k = n$

$\therefore k = \log_2 n$

$\therefore$ Time complexity : $O(\log_2 n)$

(9)

```
for(i=1; i<n; i=i*2)
{
    statement;
}
```

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

$2^k = n$

$\therefore k = \log_2 n$

$\therefore$ Time Complexity: $O(\log_2 n)$

$n = 8$

i
1
2
4
8 X

3

$\log_2 8 = 3$

$\log_2 2^3 = 3 \log_2 2$

$= 3 \times 1$

$= 3$

$n = 10$

i
1
2
4
8
16 X

4

$\log_2 10 = 3.2$

```
for(i=1; i<n; i++)
{
    statement;
}
```

$i = 1 + 1 + 1 + \ldots + 1 = n$

$k = n$

$\therefore$ Time Complexity:

$O(n)$

(10)

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

i
n
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
⋮
$\frac{n}{2^k}$

Assume $i < 1$

$\therefore \frac{n}{2^k} < 1$

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

$\Rightarrow 2^k = n$

$\Rightarrow k = \log_2 n$

$\therefore$ Time Complexity :

$O(\log_2 n)$

(11)

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

- Assume $i * i >= n$

$$i^2 = n$$

$$i = \sqrt{n}$$

∴ Time complexity:

$O(\sqrt{n})$

(12)

```
for (i=0; i<n; i++) {
{
    statement;  ------> n
}
for (j=0; j<n; j++)
{
    statement;  ------> n
}
```

Time complexity :

$$O(n) + O(n)$$

$$= O(2n)$$

$$= 2O(n)$$

$$= O(n)$$

(13)

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

$p = \log_2 n$

$\log_2 p$

$\log_2 (\log_2 n)$

∴ Time complexity :

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

(14)

```
for (i=0; i<n; i++) ——————> n
{
    for (j=1; j<n; j= j*2) ——> n x logn
    {
        statement; ——————> n x logn
    }
}
```

∴ Time Complexity :

$$O(n) + O(n * \log n) + O(n \log n)$$

$$= O(2n \log n + n)$$

$$= O(2n \log n)$$

$$= 2\,O(n \log n)$$

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

1. for (i=0; i<n; i++) ⟶ $O(n)$
2. for (i=0; i<n; i=i+2) ⟶ $\frac{n}{2} \rightarrow O(n)$
3. for (i=n; i>1; i--) ⟶ $O(n)$
4. for (i=1, i<n; i=i*2) ⟶ $O(\log_2 n)$
5. for (i=1; i<n; i=i*3) ⟶ $O(\log_3 n)$
6. for (i=n; i>1; i=i/2) ⟶ $O(\log_2 n)$

(15)

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

i	k
1	1
2	1+1 = 2
3	2+2
4	2+2+3
5	2+2+3+4
⋮	⋮
m	1+2+2+3+4+...+m

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

Assume,

$k \geq n$

$\Rightarrow \frac{m(m+1)}{2} \geq n$

$\Rightarrow m^2 \geq n$

$\Rightarrow m = \sqrt{n}$

Time complexity :

~~$O(m^2)$~~ $O(\sqrt{n})$

16) A() {

```
i = 1, s = 1;
while (s <= n) {
  i++;
  s = s + i;
  printf("Section: 61-S");
}
}
```

i	s
1	1
2	1+2 = 3
3	1+2+3
4	1+2+3+4
⋮	⋮
m	1+2+3+4+...+m
	$= \frac{m(m+1)}{2}$

Assume,

$$s > n$$

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

$$\Rightarrow m^2 = n$$

$$\Rightarrow m = \sqrt{n}$$

∴ Time complexity :

$$O(\sqrt{n})$$

(17)

```
int i, j, k=0;
for (i=n/2; i<=n; i++){
    for (j=2; j<=n; j=j*2){
        k= k + n/2 ;
    }
}
```

∴ Time Complexity: $O\left(\frac{n}{2} * \log n\right)$

$= O(n \log n)$

(18)

```
int count = 0;
for (int i = N; i > 0; i /= 2){
    for(int j = 0; j < i ; j++){
        count ++;
    }
}
```

Time complexity: $O(\log n * 2n)$

$= O(n \log n)$

(19)

```
A(){
    int i, j, k, n;
    for(i=1; i<n; i++){
        for (j=1; i<=i^2; j++){
            for (k=1; k<=n; k=k*2){
                printf("Hamim\n");
    }}}}
```

Time complexity: $O(n * n^2 * \log n)$

$= O(n^3 * \log n)$

(20)

```
for(k=1, i=1; k<n; i++)
{
    Statement;
    k = k+i;
}
```

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

Assume :

$k \geq n$

$\Rightarrow \frac{m(m+1)}{2} \geq n$

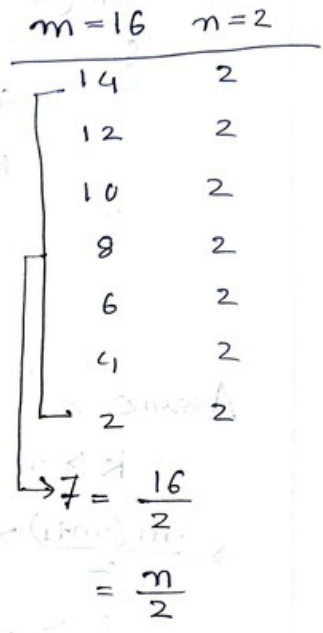
$\Rightarrow m^2 \geq n$

$\Rightarrow m = \sqrt{n}$

The complexity : $O(\sqrt{n})$

(21)

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



∴ Time Complexity : $O(n)$

(22)

```
AlgorithmTest()
{
    if (n<5) {
        printf(" %d", n);  ——→ 1
    }
    else {
        for(i=0; i<n; i++) {
            printf(" %d", i);  ——→ n
        }
    }
}
```

Time Complexity :

Best case = O(1)

Worst Case = O(n)

(23)

```
AlgorithmTest() {
    if (n>5) {
        for(i=0; i<n; i++) {
            printf("%d ", i);  ——→ n
        }
    }
}
```

Time complexity : O(n)

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

Asymptotic Notation

- O big-oh → upper bound
- Ω big-omega → Lower bound
- Θ big-theta → Avarage bound

Big-oh

The function $f(n) = O(g(n))$ if $\exists$ +ve constant c and n_0

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

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

$2n + 3 \leq 10n \quad n \geq 1$

$f(n) \quad c \; g(n)$

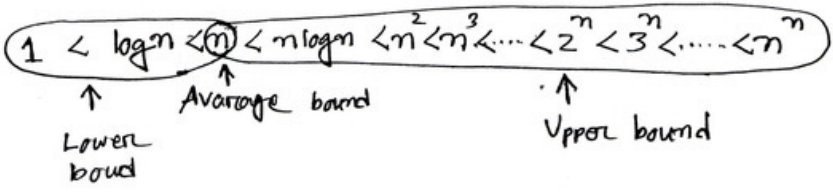
$\therefore f(n) = O(n)$

Or,

$2n + 3 \leq 2n^2 + 3n^2 \quad n \geq 1$

$\rightarrow 2n + 3 \leq 5n^2$

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



Big - Omega

The function $f(n) = \Omega(g(n))$ iff $\exists$ +ve constant c and n_0.

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

$\Rightarrow$

$c_g = f(n) = 2n + 3$

$2n + 3 \geqslant 1 \times n \quad \forall \quad n \geqslant 1$

$f(n)$ c $g(n)$

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

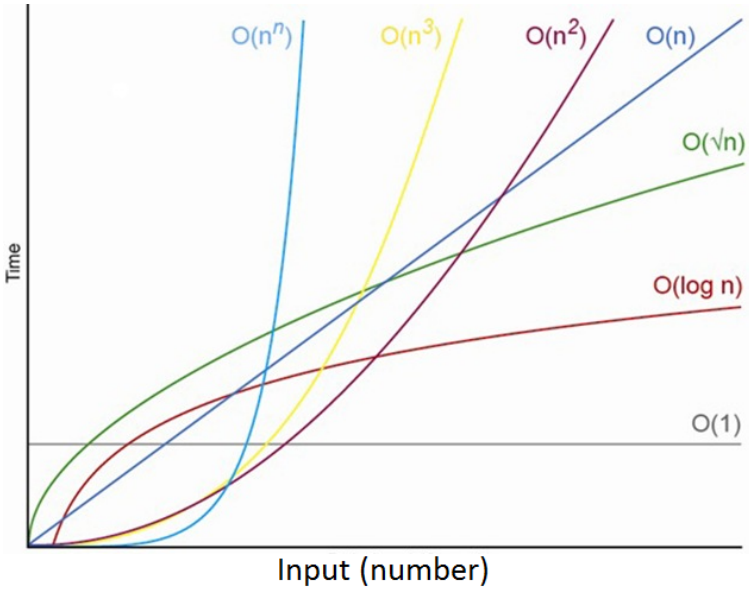
or, $f(n) = \Omega(\log n)$

Some common rates of growth

Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth.

$O(1)$	constant time
$O(\log n)$	logarithmic time
$O(n)$	linear time
$O(n \log n)$	linearithmic time
$O(n^2)$	quadratic time
$O(n^c)$	polynomial time
$O(c^n)$	exponential time
$O(n!)$	factorial time
$O(\infty)$	infinite time

Rate of Growth



Time complexity (Worst Case) of some data Structures

	Time complexity (Worst Case)			
	Access	**Search**	**Insertion**	**Deletion**
Array	O(1)	O(n)	O(n)	O(n)
Singly Linked List	O(n)	O(n)	O(1)	O(1)
Stack	O(n)	O(n)	O(n) or O(1)	O(n) or O(1)

Recurrence Relation

Time Complexity in Recurrance Relation:
Number of function call.

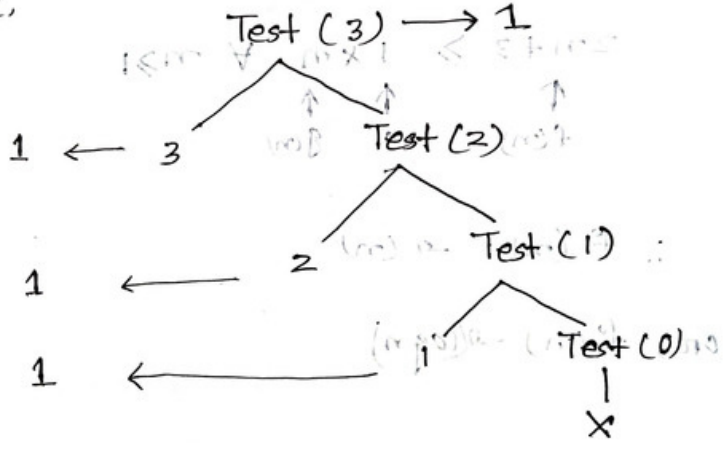
Space Complexity in Recurrance Relation:
Highest level of the tree of the Recursion.

(1)

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

Here,

Tree method



$f(n) = (n+1)$ call

$= n$

Time complexity: $O(n)$

Space Complexity: $O(n)$

Now,

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

Assume $n - k = 0$

$\therefore n = k$

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

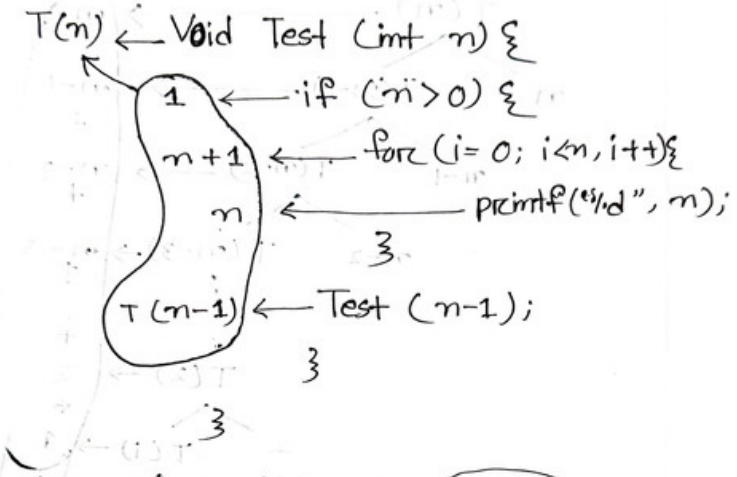
$$\Rightarrow T(n) = T(0) + n$$

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

$$O(n)$$

Time complexity : $O(n)$

(2)



$$T(n) = T(n-1) + \underbrace{2n+2}_{n} = T(n-1) + n$$

→ Asymptotic notation (n)

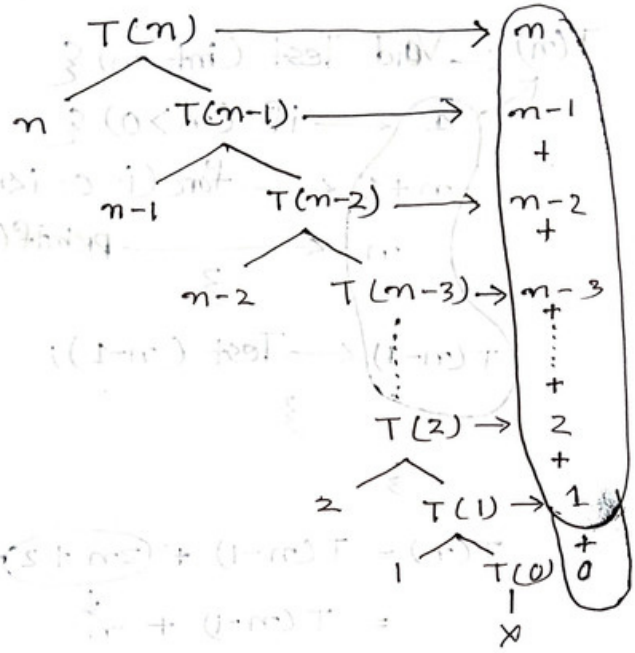
Here,

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

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

Here

Tree method



$$0 + 1 + 2 + \cdots + n-1 + n = \frac{n(n+1)}{2}$$

$$T(n) \doteq \frac{n \cdot (n+1)}{2}$$

$$\doteq n^2$$

Time complexity : $O(n^2)$

Here, Substitude method

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

$T(n) = T(n-1) + n$ — (1) $\because T(n) = T(n-1) + n$

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

$\Rightarrow T(n) = T(n-2) + (n-1) + n$ — (2) $\because T(n-2) = T(n-3) + n-2$

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

$\Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n$ — (3)

⋮

continue for k time

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

$\cdots + (n-1) + n$

Assume

$n - k = 0$

$\therefore n = k$

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

$+ (n-1) + n$

$\Rightarrow T(n) = T(0) + 1 + 2 + 3 + \cdots + (n-1) + n$

$\Rightarrow T(n) = 1 + \frac{n(n+1)}{2}$

$\therefore$ Time complexity: $O(n^2)$

③

$T(n)$ ← void Test (int n) {

if (n>0) {

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

printf("%d", i);

}

Test (n-1);

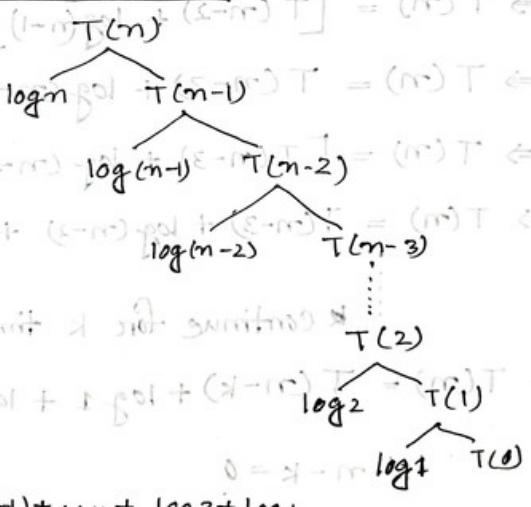
}

}

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

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

Tree method



$\log n + \log(n-1) + \cdots + \log 2 + \log 1$

$= \log(n \times (n-1) \times \cdots \times 2 \times 1)$

$= \log(n!)$

$= \log n^n$

$= n \log n$

Time complexity: $O(n \log n)$

Substitude method

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

$$T(n) = T(n-1) + \log n \quad \text{——— (1)}$$

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

$$\Rightarrow T(n) = T(n-2) + \log(n-1) + \log n \quad \text{——— (2)}$$

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

$$\Rightarrow T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n \quad \text{——— (3)}$$

⋮ continue for k time

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

$\therefore n - k = 0$

$\therefore n = k$

$T(n) = T(0) + \log(1 * 2 * 3 * \cdots * (n-1) * n)$

$\Rightarrow T(n) = T(0) + \log n!$

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

Time complexity : $O(n \log n)$

Recurrence Relation

(Without coefficient)

$T(n) = T(n-1) + 1 \longrightarrow O(n)$

$T(n) = T(n-1) + n \longrightarrow O(n^2)$

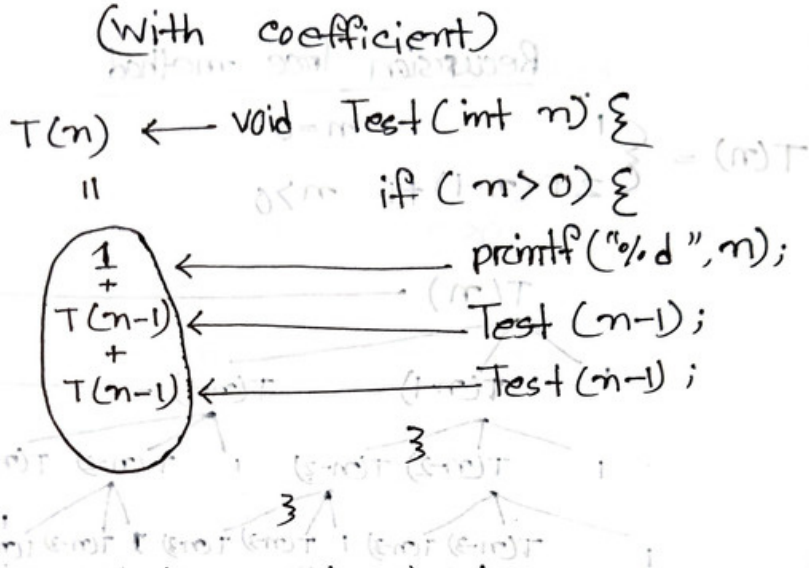
$T(n) = T(n-1) + \log n \longrightarrow O(n \log n)$

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

$T(n) = T(n-2) + 1 \longrightarrow \frac{n}{2} \longrightarrow O(n)$

$T(n) = T(n-100) + n \longrightarrow O(n^2)$

④ (With coefficient)

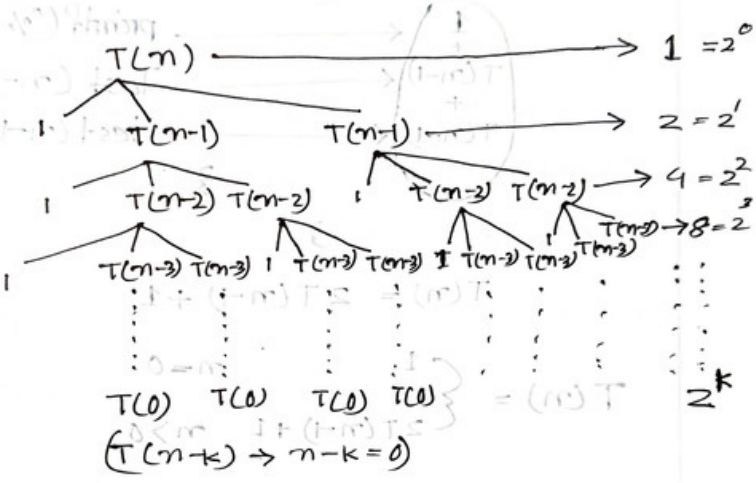


$$T(n) = 2T(n-1) + 1$$

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

Recursion Tree method

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



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

GP series:

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

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

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

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

Assume $n - k = 0$
$\therefore n = k$

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

Time complexity: $O(2^n)$

Back Substitution method

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

$T(n) = 2T(n-1)+1$ ——— (1)

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

$\Rightarrow T(n) = 2^2 T(n-2)+2+1$ ——— (2)

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

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

⋮

continue for k time

$\Rightarrow T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2 + 1$ ——— (4)

Assume, $n-k = 0$

$n = k$

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

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

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

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

Time complexity : $O(2^n)$

Recurrence Relation

Master theorem for Decreasing Functions

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

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

$T(n) = T(n-1) + 1 \longrightarrow O(n)$

$T(n) = T(n-1) + n \longrightarrow O(n^2)$

$T(n) = T(n-1) + \log n \longrightarrow O(n \log n)$

Case 1:

if $a = 1$

$O(n * f(n))$

or $O(n^{k+1})$

$T(n) = 2T(n-1) + 1 \longrightarrow O(2^n)$

$T(n) = 3T(n-1) + 1 \longrightarrow O(3^n)$

$T(n) = 2T(n-1) + n \longrightarrow O(n 2^n)$

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

Case 2:

if $a > 1$

~~$O(n^k a^{n/b})$~~

$O(n^k a^{\frac{n}{b}})$

or, $O(f(n) \cdot a^{\frac{n}{b}})$

Case 3:

if $a < 1$

$O(n^k)$

or, $O(f(n))$

Recurrence Relation

Dividing Functions

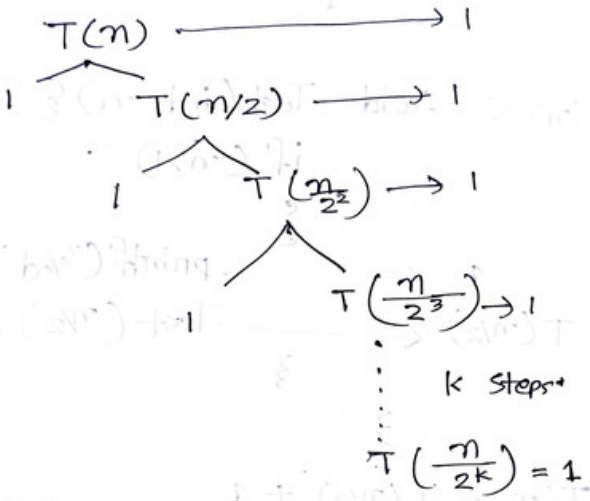
(5)

```
T(n) ←— void Test (int n) {
            if (n>1)
            {
1 ←————————— printf ("%d", n);
T(n/2) ←———— Test (n/2);
            }
}
```

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

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

Recursion Tree method



k steps

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

$n = 2^k$

$\Rightarrow k = \log_2 n$

Time Complexity : $O(\log n)$

Subtitution method

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

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad \text{---(1)}$$

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

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

$$\Rightarrow T(n) = \left[T\left(\frac{n}{2^2}\right) + 1\right] + 1$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^2}\right) + 2 \quad \text{---(2)}$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^3}\right) + 3 \quad \text{---(3)}$$

$$\vdots$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^k}\right) + k \quad \text{---(4)}$$

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

$$\Rightarrow n = 2^k$$

$$\therefore k = \log n$$

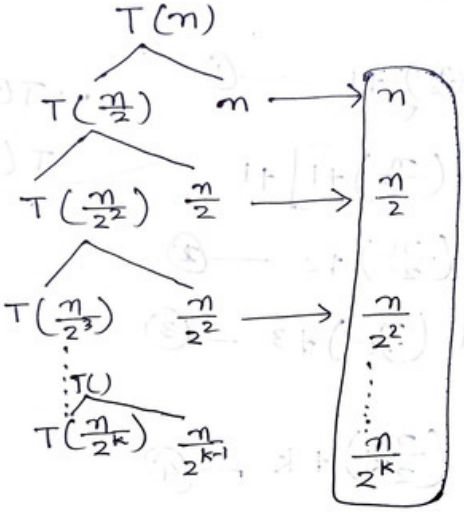
$$\Rightarrow T(n) = T(1) + \log n$$

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

Time complexity: $O(\log n)$

(6)

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



$$T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k}$$

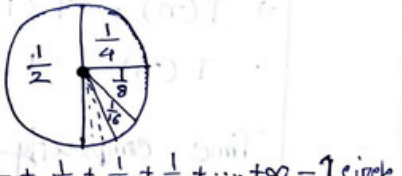
$$\Rightarrow T(n) = n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^k}\right]$$

$$= n \sum_{i=0}^{k} \frac{1}{2^i}$$

$$= n \times 1$$

$$\Rightarrow T(n) = n$$

Time complexity: $O(n)$



Substitute method

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

$$T(n) = T\left(\frac{n}{2}\right) + n \quad \text{——} (1)$$

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

$$\Rightarrow T(n) = T\left(\frac{n}{2^2}\right) + \frac{n}{2} + n \quad \text{——} (2)$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + \frac{n}{2} + n \quad \text{——} (3)$$

$\vdots$

Continue to k time

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

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

$\therefore n = 2^k$ and $k = \log n$

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

$$\Rightarrow T(n) = 1 + n[1+1]$$

$$\Rightarrow T(n) = 1 + 2n$$

$\therefore$ Time complexity: $O(n)$

(7)

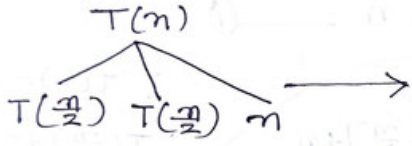
```
T(n) ←—— Void Test (int n) {
            if (n>1)
            {
                for (i=0; i<n; i++)
                {
n ←————————                 statement;
                }
T(n/2) ←—— Test (n/2);
T(n/2) ←—— Test (n/2);
            }
        }
```

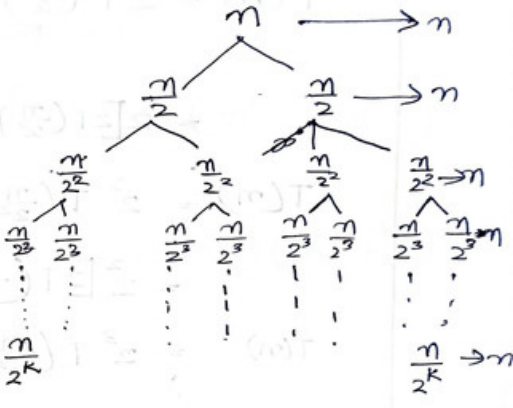
$\therefore T(n) = 2T\left(\frac{n}{2}\right) + n$

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

Tree method

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





$T(n) = nk$

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

$\Rightarrow n = 2^k$

$\therefore k = \log n$

$\Rightarrow T(n) = n \log n$

$\therefore$ Time complexity : $O(n \log n)$

Back Substitution method

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

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \text{---} (1)$$

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

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

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

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

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

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

$$\vdots$$

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

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

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

$\Rightarrow n = 2^k$

$\therefore k = \log n$

$$T(n) = 2^k T(1) + kn$$

$$= n \times 1 + \log n \times n$$

Time complexity: $O(n \log n)$

Master Theorem for Dividing Functions

① $\log_b a$ $\quad T(n) = aT\left(\frac{n}{b}\right) + f(n)$

② k $\quad a \geq 1,\ b > 1$ $\quad f(n) = O\left(n^k \cdot (\log n)^p\right)$

or, $f(n) = O\left(n^k \cdot \log^p n\right)$

Case 1: if $\log_b a > k$ Then, $O\left(n^{\log_b a}\right)$

Case 2: if $\log_b a = k$

if $p > -1 \rightarrow O\left(n^k \cdot \log^{p+1} n\right)$

if $p = -1 \rightarrow O\left(n^k \cdot \log \log n\right)$

if $p < -1 \rightarrow O\left(n^k\right)$

Case 3: if $\log_b a < k$ if $p \geq 0 \rightarrow O\left(n^k \cdot \log^p n\right)$

if $p < 0 \rightarrow O\left(n^k\right)$

⊛ $T(n) = 2T(\frac{n}{2}) + 1$

$a = 2$

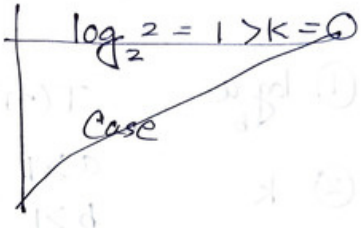
$b = 2$

$f(n) = O(1)$

$= O(n^0 \cdot \log^0 n)$

$k = 0, \; p = 0$

$\log_2 2 = 1 > k = 0$



Case 1: $O(n^1)$

⊛ $T(n) = 4T(\frac{n}{2}) + n$

$\log_2 4 = 2 > k = 1 \quad p = 0$

Case 1: $O(n^2)$

⊛ $T(n) = 8T(\frac{n}{2}) + n^1$

$\log_2 8 = 3 > k = 1, \; p = 0$

∴ Time complexity: $O(n^3)$

⊛ $T(n) = 8T\left(\frac{n}{2}\right) + n^2$

$\log_2 8 = 3 > k = 2, \; p = 0$

∴ Time complexity: $O(n^3)$

⊛ $T(n) = 9T\left(\frac{n}{3}\right) + 1$

$\log_3 9 = 2 > k = 0$

∴ Time complexity: $O(n^2)$

⊛ $T(n) = 9T\left(\frac{n}{3}\right) + n^1$

$\log_3 9 = 2 > k = 1$

∴ Time complexity: $O(n^2)$

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

$\log_2 2 = 1 = k=1 \;,\; P=0$

∴ Time complexity : $O(n \log n)$

(*) $T(n) = 8T(\frac{n}{2}) + n^3$

$\log_2 8 = 3 \quad k=3$

∴ Time complexity : $O(n^3 \log n)$

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

$\log_2 2 = 1 \quad k=1 \quad P=-1$

∴ Time complexity : $O(n \log(\log n))$

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

$\log_2 2 = 1 \;,\; k=1 \;,\; P=-2$

∴ Time complexity : $O(n)$

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

$\log_2 1 = 0 < k = 2$

∴ Time complexity : $O(n^2)$

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

$\log_2 2 = 1 < k = 2$

∴ Time complexity : $O(n^2)$

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

$\log_2 2 = 1 < k = 2, \; p = 1$

∴ Time complexity : $O(n^2 \log n)$

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

$\log_2 2 = 1, \; k = 2, \; p = 2$

∴ Time complexity : $O(n^2 \log^2 n)$

* $T(n) = 4T(\frac{n}{2}) + \frac{n^3}{\log n}$

$\log_2 4 = 2, \; k = 3, \; p = -1$

∴ Time complexity : $O(n^3)$

Case 1 :

$T(n) = 2T\left(\frac{n}{2}\right) + 1 \longrightarrow O(n^1)$

$T(n) = 4T\left(\frac{n}{2}\right) + 1 \longrightarrow O(n^2)$

$T(n) = 4T\left(\frac{n}{2}\right) + n^1 \longrightarrow O(n^2)$

$T(n) = 8T\left(\frac{n}{2}\right) + n^2 \longrightarrow O(n^3)$

$T(n) = 16T\left(\frac{n}{2}\right) + n^2 \longrightarrow O(n^4)$

Case 3 :

$T(n) = T\left(\frac{n}{2}\right) + n^1 \longrightarrow O(n)$

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

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

$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \log^2 n \longrightarrow O(n^3 \log^2 n)$

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

Case 2 :

$T(n) = T(\frac{n}{2}) + 1 \longrightarrow O(\log n)$

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

$T(n) = 2T(\frac{n}{2}) + n \log n \longrightarrow O(n \log^2 n)$

$T(n) = 4T(\frac{n}{2}) + n^2 \longrightarrow O(n^2 \log n)$

$T(n) = 4T(\frac{n}{2}) + (n \log n)^2 \longrightarrow O(n^2 \log^3 n)$

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

$T(n) = 2T(\frac{n}{2}) + \frac{n}{\log^2 n} \longrightarrow O(n)$

Root Function

```
T(n) ← void Test (int n) {
          if (n>n)
          {
1 ←———— Statement;
T(√n) ←———— Test (√n);
          }
}
```

$\therefore T(n) = T(\sqrt{n}) + 1$

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

$T(n) = T(\sqrt{n}) + 1$

$\Rightarrow T(n) = T\left(n^{\frac{1}{2}}\right) + 1$ —— (1)

$\Rightarrow T(n) = T\left(n^{\frac{1}{2^2}}\right) + 2$ —— (2)

$\Rightarrow T(n) = T\left(n^{\frac{1}{2^3}}\right) + 3$ —— (3)

$\vdots$

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

Assume $n = 2^m$

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

Assume $T\left(2^{\frac{m}{2^k}}\right) = T(2^1)$

$$\therefore 2^{\frac{m}{2^k}} = 2^1$$

$$\therefore \frac{m}{2^k} = 1$$

$$\Rightarrow m = 2^k$$

$$\therefore k = \log_2 m$$

$$\because n = 2^m$$

$$\therefore m = \log_2 n$$

$$\therefore k = \log \log_2 n$$

$\therefore$ Time Complexity: $O(\log \log_2 n)$