Time Complexity and Space Complexity

(Analysing Algo)

Time Complexity means CPU time
Space Complexity means main memory (RAM).

| 1 Byte | 1000 ns :-> cheap, but accessing time is slow
Hard Disk

| 1 Byte | 10 ns :-> moderate ✓
RAM

| 1 Byte | 1 ns :-> costlier, but fast.
Cache Memory

* Time Complexity is more important than Space Complexity.

Time Complexity

A : Any Algorithm
C(A) : Compile time (A)
R(A) : Run time (A)

$$T(A) = C(A) + R(A)$$

C(A) ⇓ Compiler + ⇓ Programming language of compiler.

R(A) ⇓ Processor ⇓ type of processor.

∴ Compilation time is faster in java-compiler than C-compiler.

i.e. C program Run is faster than Java program Running.

C-Compilation: 6-phases — Machine-dependent

J-compilation: 4-phases — M/C-indi

C - program is faster than Java program.

* java compiler is faster as it contains only 4 stages (lexical, syntax, semantic and intermediate code).

* C - ~~compiler~~ program Run time is faster.

Types of Analysis

Relative Analysis	Absolute Analysis
1> It is programming language of compiler and type of processor dependent analysis.	1> It is programming language of compiler and type of processor independent analysis.
2> System to System answer will be changed.	2> System to System answer will not be changed.
3> Always give exact answer.	3> It will be give approximate answer (due to independent analysis)
4>. It depend on P/L of compiler & type of processor.	4> logic of programmer matters

Super compiler → Relative Analysis
Super Algo → Absolute Analysis

In Software industries, Absolute Analysis is used.

Date

In Absolute Analysis, system-to-system, the constant are removed.
∴ Approximate answer.

100n
10n → O(n).
n
n
~~1000~~

but cannot be log(n), as it cannot increase and decrease the fun.

Absolute Analysis

It is a determination of order of magnitude of a statement.

Q1. main ()
{
1. x=y+Z; ⇒ 1
}

only one Statement

how many times the Statement is executed (run) ⇒ order of magnitude.

Ans:- O(1) is time Complexity.

e.g.(2) main ()
{
1. x=y+Z; → 1
for (i=1 ; i≤n ; i++)
{
2. x=y+z; → n
}
}

$(n+1) = O(n)$
→ constant is removed.

e.g.(3)

```
main ()
{
1. x = y + z;                      ---> 1
   for (i = 1; i <= n; i++)
   {
2.    x = y + z;                   ---> n
      for (i = 1; i <= n; i++)
      {
         for (j = 1; j <= n; j++)
         {
3.          x = y + z;             ---> n^2
         }
      }
   }
}
```

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

n^2 is larger

$i = 1$	$i = 2$	$i = 3$	$\ldots$	$i = n$
$j = 1, 2, 3, \ldots n$	$j = 1, 2, 3, \ldots n$	$j = 1, 2, 3, \ldots n$		$j = 1, 2, 3, \ldots n$

temporal locality of references : loop
Spatial " " " : next statement.

statement
1
2
3
4 a()
5
6
7

cache

contains next statement to be executed.

Many fun () calls ⇒ more miss

Finding time complexity means find largest loop e.g (4) Find time complexity

```
main ()
{
    for (i=1; i<=n; i++)
    {
        for (j=1; j<=n/2; j++)
        {
            1. x=y+z;
        }
    }
}
```

$\rightarrow n/2$

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

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

constant

$= O(n^2)$

e.g (5)

```
main ()
{
    for (i=1; i<=n; i++)
    {
        for (j=1; j<=i; j++)
        {
            for (k=1; k<=133; k++)
            {
                1 x=y+z;
            }
        }
    }
}
```

$\rightarrow 133n^2$

$\rightarrow 133n \simeq O(n^2)$

$\rightarrow 133$

i=1	i=2	i=3	. . .	i=n
j=1	j=1,2	j=1,2,3		j=1,2,3, ... n
K=133	K=133, 133	K=133,133, 133		K = 133, 133, ... 133

total $= 1 \times 133 + 2 \times 133 + 3 \times 133 + 4 \times 133 + \ldots + n \times 133$

$= 133(1+2+3+\ldots n)$

$= n^2 + n$

$= O(n^2)$

GATE

```
main()
{
  for (i=1; i<=n; i++)
  {
    for (j=1; j<=i^2; j++)
    {
      for (K=1; K<=n; K++)
      {
        x=y+z;
      }
    }
  }
}
```

$i=1$	$i=2$	$i=3$	$i=4$	...	$i=n$
$j=1$	$j=1,2,3,4$	$j=1,2,3,\dots 9$	$j=1,2,\dots 4^2$	...	$j=1,2,\dots n^2$
$K=n$	$K=n,n,n,n$	$K=n,n,\dots n$	$K=n,n,\dots n$	...	$K=n,n,\dots n$

total $= 1*n + 2^2*n + 3^2*n + 4^2*n + \dots + n^2*n$

$= n[1^2+2^2+3^2+\dots+n^2]$

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

$= O(n^4)$

Date________

GATE

```
main ()
{
for (i=1; i<=n; i++)
{
for (i=1; i<=n^2; i++)
{
for (i=1; i<=n3; i++)
{
x=y+z;
}
}
}
}
```

$n^3+2 \le n$ False

n^3+2

$n^3+1 \le n^2 \Rightarrow$ False

n^3+1

$\Rightarrow n^3$

1st for loop, 2nd for loop

$\therefore 1 * 1 * n^3 = O(n^3)$

$i = 1$
$j = n^2$
$K = n^2$

```
main()
{
for (i=1; i<=n; i++)
{
for (K=1; K<=n^2; k++)
{
for (i=1; i<=n^3; i++)
{
x=y+z;
}
}
}
}
```

$1 * n^2 * n^3 = O(n^5)$

```
main()
{
  for (i=1 ; i<=n ; i++)
  {
    1. x = y+z;
  }
}
```

$\rightarrow n$ times $= O(n)$

$i=1$

$i=2$

Q main()

```
{
  for (i=1 ; i<=n; i=i+2)
  {
    x = y+z;
  }
}
```

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

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

$i=1$

$i=3$

$i=5$

$\vdots$

Note:

$i = i+1 \Rightarrow \frac{n}{1}$

$i = i+2 \Rightarrow n/2$

$i = i+10 \Rightarrow n/10$

$i = i+100 \Rightarrow n/100$

$O(n)$

Q. main()

```
{
  for ( i=1 ; i<=n; i=2*i)
  {
    x = y+z;
  }
}
```

$\Rightarrow \log_2 n$

1

2

4

8

16

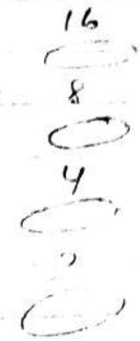
32

128 $\Rightarrow \log_2 128 = 7$

Date.

for (i=1 ; i ≤ n; i = 3*i)

⇒ $O(\log_3 n)$



Q. main()

{

for (i=n ; i ≥ 1 ; i = i/2)

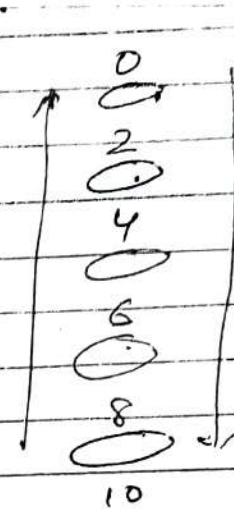
{

x = y+z;

}

}

⇒ $O(\log_2 n)$



Q for (i=n ; i>1 ; i=i-1)

⇒ $\frac{n}{1}$ times = $O(n)$

Q for (i=n ; i>1 ; i=i-2)

⇒ $\frac{n}{2}$ times = $O(n)$

Q for (i=n ; i>1 ; i=i/10)

⇒ $O(\log_{10} n)$

$n * \log_3 n * \frac{\log n}{10}$

$= O(n (\log n)(\log_3 n))$

Q. main()

{

for (i=1; i ≤ n; i++) → n times

{

for (j=1; j ≤ n; j=3*j) ⇒ $\log_3 n$

{

for (K=1; K ≤ log n; K=K+10) ⇒ $\frac{\log n}{10}$ times

{ x=y+z;

}}}

Date

$n * \log_2 n * \log_3 n$

$= O[n(\log n)^2]$

$(\log_1 n) * \log_2 n * \log_3 n) \longrightarrow$ for $(i=1; i \leq n; i=2*i)$

$= O(\log n)^3$

i) while $(i \leq n)$ $\quad i=1$

{ $i = 2*i;$

}

$2^K . i = n$

$2^K = n$

K times $\quad K \log_2 2 = \log_2 n$

$i = ((2*i)2)2) \cdots 2)$

$K = \log_2 n$

Q. $\frac{n}{2^K} = 1 \Rightarrow 2^K = n$

$K = \log_2 n$

```
main()
{
  for(i=1; i≤n; i++)
  {
    j=2;
    while(j≤n)
    {
      j=j^2
    }
  }
}
```

→ n times

⇒ O(n log

→ $\log(\log n)$

```
while(j≤n)
2 j=j+1
}
```

a) $O(n)$ b) $O(n^2)$ c) $O(n\log n)$ d) $O(n\log(\log n))$

e) $O(n^3)$

```
while(j≤n^2)
{ j=j+1
}
```

$j=2$

2	2
4	4
8	16
16	256
32	⋮
64	⋮
$\log n$	$\log(\log n)$

```
while (j≤n)
{
  j=2*j
}
```

$\log(\log n) << \log n$

$$((((2)^2)^2)^2)^{2\cdots \text{ k times}}$$

$$= 2^{2^k} = n$$

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

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

$(2 * j) * 2 * 2 * 2 * 2 \ldots \quad n \quad = 2^{k} = n$

$k = \log n$

Q $A(n)$

{

if ($n \leq 2$) return;

else

return ($A(\sqrt{n})$)

}

a) $O(n)$ b) $O(1)$ c) $O(\log n)$ ✓d) $O(\log \log n)$ e) $O(n^2)$

```
c = 0
A(c)        <- recursive function.     (or) for ()     (or) while ()
{                                          {                {
if (c < 99) return                         [code]           [code]
[code]                                     }                }
c++
A(c)
}
```

$\left(\left(n^{1/2}\right)^{1/2}\right)^{1/2} \ldots$ k times $= n^{\frac{1}{2^k}} = 2$

Ans:- A(1000)

A(33)

A(6)

A(2)

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

$\log n = 2^k$

$\Rightarrow k = \log_2 \log n$