

Time and Space Complexity

In today's fast-paced world, the efficiency of algorithms is crucial for software development. Time and space complexity analysis plays a significant role in evaluating algorithm performance and resource utilization. It helps developers understand how algorithms scale with input size, guiding them in selecting the most efficient solutions. By quantifying runtime and memory requirements, complexity analysis aids in predicting algorithm behavior and optimizing code for better performance, ensuring the development of scalable and efficient software solutions.

I have tried making time and space complexity simple and less complicated so that people reading this find this easy to understand.

There are mainly three things we consider when calculate time and space complexity:

1. Best case : Big Omega(Ω)
2. Average Case: Big Theta(Θ)
3. Worst case: Big-Oh (O)

Rule to follow :

1. Ignore all the constants from the result.
Eg: If result : $3n+1$.
Ignore 3 and 1 and consider only n as it doesn't contribute much to the final solution.
2. Always consider the number in the highest order.
Eg: If your result : $3n^3+2n^2+n$.

Ignore $3, 2n^2, n$ as only consider n^3 as it doesn't contribute much to the final solution.

Time Complexity:

Time complexity measures the amount of time an algorithm takes to run as a function of the size of its input. It helps in understanding how the runtime of an algorithm grows with respect to the size of the input data.

Let's see some examples:

1. Ex:1

```

           1   2   4
for(int i = 1; i < n; i++)
{
           3
    Print "Hii";
}
```

Here this code executes from $4+3+3+\dots+3$ times.

4 because it takes one extra step in the beginning to initialize the i variable.

So,

$4+3+3+3+\dots+3$

Can also be written as

$1+3+3+3+\dots+3$

$= 3n+1$

As per the rule remove constant to the result will be $= n$

Time complexity for this algorithm would be:

a. Best case $= \Omega(n)$

b. Average case = $\Theta(n)$

c. Worst case = $O(n)$

Explanation: This is because even if the input n is passed as 1 or maybe 100 or may be 10,00,000 the program has to still print Hii n times.

2. Ex: 2 : Linear search

```
for(int i = 0; i < n; i++)
{
    if(nums[i] == x)
    {
        Print "Hii";
        Break;
    }
}
```

Time complexity for this algorithm would be:

a. Best case : $\Omega(1)$, this is because if the search number was present at the 1st position, then this algorithm just had to run 1 time.

b. Average case : $\Theta(n)$

c. Worst case : $O(n)$

Explanation: This is because the algorithm must either run $n/2$ (as once the number is found, the execution breaks out of the loop) or n times to find the given value.

3. Ex: 3

```
for(int i = 0; i < 10; i++)
```

```

{
    Print "Hii";
}

```

Time complexity for this algorithm would be:

- a. Best case: $\Omega(10) = \Omega(1)$
- b. Average case: $\Theta(10) = \Theta(1)$
- c. Worst case: $O(10) = O(1)$

Explanation: This is because no matter what input you provide the program will execute for the value you have specified. i.e in this example it is 10.

4. Ex: 4 : Print Total sum for a given value.

Print $n(n+1)/2$

Time complexity for this algorithm would be:

- a. Best case: $\Omega(1)$
- b. Average case: $\Theta(1)$
- c. Worst case: $O(1)$

Explanation: This is because no matter what the value of n is, it will take a constant time to execute.

Eg: If $n = 100$ then it will print 5050 in one execution.

If $N = 10000$ then it will print 50005000 in one execution.

5. Ex: 5

for(int i = 1; i <= n; i++) -> Executes n times

```

{
    for(int j = 1; j < n; j++)    -> For i executes n times
    {
        Print "Hii";
    }
}

```

Total : n^2

Lets dry run this:

For:	For:	For:		For:
i=1	i=2	i=3		i=n
J=1 to n	J=1 to n	J= 1 to n	J= n
Print n times.	Print n times.	Print n times.		Print n times.

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

is the time complexity for this algorithm.

Note: This can be directly calculated as $O(n^2)$ only when i and j are not dependent on each other.

6. Ex: 6

```

for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= i; j++)
    {
        Print "Hii";
    }
}

```

}

Note in this problem j is dependent on i lets see how can we calculate time complexity of this algorithm.

Lets dry run this:

For:	For:	For:	For:
i=1	i=2	i=3		i=n
J=1	J=1 to 2	J= 1 to 3		J= 1 to n
Print 1 times.	Print 2 times.	Print 3 times.		Print n times.

$$= 1+2+3+.....n$$

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

$$= \boxed{O(n^2)}$$

is the result.

7. Ex: 7

```
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= i^2; j++)
    {
        Print "Hii";
    }
}
```

Let's dry run this:

For:	For:	For:	For:
i=1	i=2	i=3		i=n
J=1 to 1	J=1 to 4	J= 1 to 9		J= 1 to n ²
Print 1 times.	Print 2 times.	Print 3 times.		Print n times.

$$\begin{aligned}
&= 1^2 + 2^2 + 3^2 + \dots + n^2 \\
&= n(n+1)(2n+1)/6 = n^2 + n(2n+1)/6 \\
&= 2n^3 + n^2 + 2n^2 + n \\
&= \boxed{O(n^3)} \quad \text{is the answer.}
\end{aligned}$$

8. Ex: 8

```

for(int i = 0; i <= n; i++)
{
    for(int j = 1; j <= m; j++)
    {
        Print "Hii";
    }
}

```

Here in this problem both i and j have their own data hence the algorithm would be

$\boxed{O(nm)}$

9. Ex: 9

```

for(int i=1; i <= n; i++)
{
    for(int j = 1; j <= i^2; j++)

```

```

    {
        for(int k =1 ;k <= n/2;k++)
        {
            Print "Hii";
        }
    }
}

```

Let's dry run this:

For:	For:	For:		For:
i=1	i=2	i=3		i=n
J=1 to 1	J=1 to 4	J= 1 to 9	J= 1 to n^2
K = 1 to $n/2$	K = 1 to $n/2$	K = 1 to $n/2$		K = 1 to $n/2$
Print $n/2$ times.	Print $4n/2$ times.	Print $9n/2$ times.		Print $nn/2$ times.

$$= n/2 + 4n/2 + 9n/2 + \dots + nn/2$$

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

$$= n/2 (n(n+1)(2n+1)/6) = n/2 * n^3 \text{ (As we saw in}$$

problem 7)

$$= n^4 = \boxed{O(n^4)}$$

10. Ex: 10

```

for(int i = 1; i <= n; i = i*2)
{
    Print "Hii";
}

```


Let's dry run this:

For:	For:	For:		For:
i=1 2^0	i=2 2^1	i=4 2^2		i=n 2^k
Print 1 times.	Print 1 times.	Print 1 times.	Print 1 times.

$$= 1+1+1+1 \dots\dots\dots 1$$

When the printing is constant

$$= 2^0+2^1+2^3+\dots\dots\dots+2^k$$

This can be written as,

$$n = 2^k$$

When we add logs on both side,

$$\log(n) = \log(2^k)$$

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

$$1/k = \log(n)/\log(2)$$

$$K = \log_2 n$$

If you check the algorithm at a particular iteration this program prints 'Hii' 2^{k+1} times.

Eg: At $i = 2^2$ the program prints 'Hii' 3 times that is 2^{k+1} .

At $i = 2^3$ the program prints 'Hii' 4 times that is 2^{k+1} .

Then the above equation will be

$$= \log_2 n + 1$$

Since 1 is constant ,



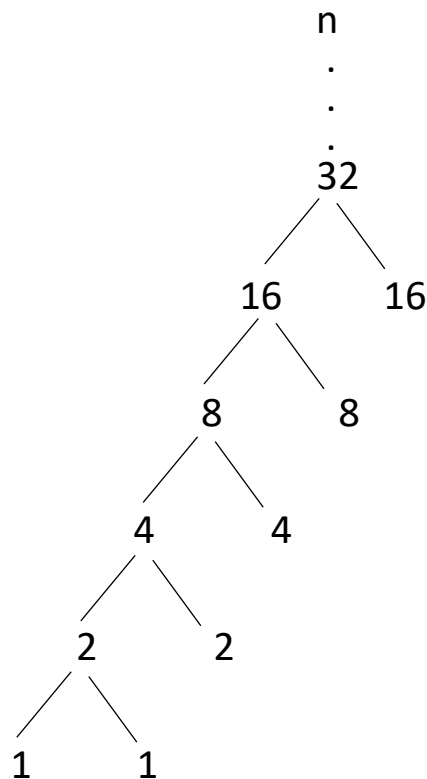
= $O(\log_2 n)$ is the result.

Another shortcut way to find this is, now the i will be

1 – 2 – 4 – 8 – 16 – 32n

Now if you look closely from n it is dividing the number by 2.

1 – 2 – 4 – 8 – 16 – 32n
 ——— ——— ——— ——— ——— ———
 half half half half half half half



Now the time complexity can be conclude as,

$O(\log_2 n)$

Similarly,
 If

1 – 3 – 9 – 27 – 81n

— — — — —
half half half half half half

Now the time complexity can be conclude as,

$O(\log_3 n)$

Similarly 4 and so on.

11. Ex: 11

```
for(int i = n/2; i <= n; i++)
{
    for(int j = 1; j <= n/2; j++)
    {
        for(int k = 1; k <= n; k++)
        {
            Print "Hii";
        }
    }
}
```

Now, for(int i = n/2; i <= n; i++) executes n/2 times
 for(int j = 1; j <= n/2; j++) executes n/2 times
 for(int k = 1; k <= n; k++) executes n times

$$n/2 * n/2 * n = n^3$$

$$= O(n^3)$$

12. Ex: 12

```
for(int i = n/2; i <= n; i++)
{
    for(int j = 1; j <= n; j = 2*j)
    {
        for(int k = 1; k <= n; k = k*2)
        {
            Print "Hii";
        }
    }
}
```

Now, for(int i = n/2; i <= n; i++) executes n/2 times

for(int j = 1; j <= n; j = 2*j) executes $\log_2 n$ times
 for(int k = 1; k <= n; k = k*2) executes $\log_2 n$ times

$$n/2 * \log_2 n * \log_2 n = n/2 (\log_2 n)^2 = n (\log_2 n)^2$$

$$= \boxed{n (\log_2 n)^2}$$

13. Ex: 13

```
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= n; j = j+i)
    {
        Print "Hii";
    }
}
```

Let's dry run this:

For:	For:	For:	For:
i=1	i=2	i=4	i=n
j=1, 1+1, 2+1 to n+1	j = 1, 1+2, 3+2, 5+2 to n+2	j = 1, 1+3, 4+3 to n+3 j = 1 to n+n
Print n/1 times.	Print n/2 times.	Print n/3 times.	Print n/n times.

For:	For:	For:	For:
i=1	i=2	i=3	i=n
J=1 , 2, 3 to n	J=1, 3, 5,7....n	J= 1,4,7.... J= 1 to n
Print n/1 times.	Print n/2 times.	Print n/3 times.	Print n/n times.

For:	For:	For:	For:
i=1	i=2	i=3	i=n
J=1 to n	J=1 to n/2	J= 1to n/3 J= 1 to n/n
Print n times.	Print n/2 times.	Print n/3 times.	Print n/n times.

$$\begin{aligned}
&= n + n/2 + n/3 + \dots + n/n \\
&= n(1 + 1/2 + 1/3 + \dots + 1/n) \rightarrow \text{Harmonic series.} \\
&= n \log_e n \\
&= \boxed{O(n \log_e n)} \text{ hence the result.}
\end{aligned}$$

Space Complexity :

Space complexity refers to the amount of memory space an algorithm requires to execute, measured in terms of the input size. It's vital for understanding an algorithm's memory usage and scalability.

There are two types while calculation space complexity:

- Auxiliary space : This does not include space of given input.
- Total space complexity: This includes space of given input.

Eg: 0 1 2 3 4
Input :

4	6	7	8	1
---	---	---	---	---

 0 1 2 3 4 5
Output:

1	16	23	4	3	8
---	----	----	---	---	---

Here ,

- Auxiliary space : $n = O(n)$ (Only considers output space)
- Total space complexity : $n + n = 2n = O(n)$ (considers both input and output space)

1. Ex: 1

```
for(int i=1;i<=n;i++)
{
    Print "Hi";
}
```

i took constant space.

Auxiliary = 1 = $\boxed{O(1)}$

Total space = 1+1 = $\boxed{O(1)}$

2. Ex: 2

```
int add (int n){  
    if (n <= 0){  
        return 0;  
    }  
    return n + add (n-1);  
}
```


Here each call add a level to the stack :

1. add(4)
2. add(3)
3. add(2)
4. add(1)
5. add(0)

Each of these calls is added to call stack and takes up actual memory.

So, it takes $O(n)$ space.

Order of Space Complexity:

- $O(N!)$
 - $O(2^n)$
 - $O(n^3)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(n)$
 - $O(\sqrt{n})$
- > Worst case
- 

- $O(\log n)$
- $O(1)$ \rightarrow Best Case