Time and Space Complexity

In this article, we are going to discuss the terms time complexity and space complexity. Here, we will learn the basics of them and in the upcoming part, we will learn complex algorithms with a detailed discussion of their time complexity and space complexity.

What is Time Complexity?

We can solve a problem using different logic and different codes. Time complexity basically helps to judge different codes and also helps to decide which code is better. In an interview, an interviewer generally judges a code by its time complexity.

Now, the term, time complexity, seems that it is referring to the time taken by a machine to execute a particular code. But in real life, ***Time complexity does not refer to the time taken by the machine to execute a particular code***.

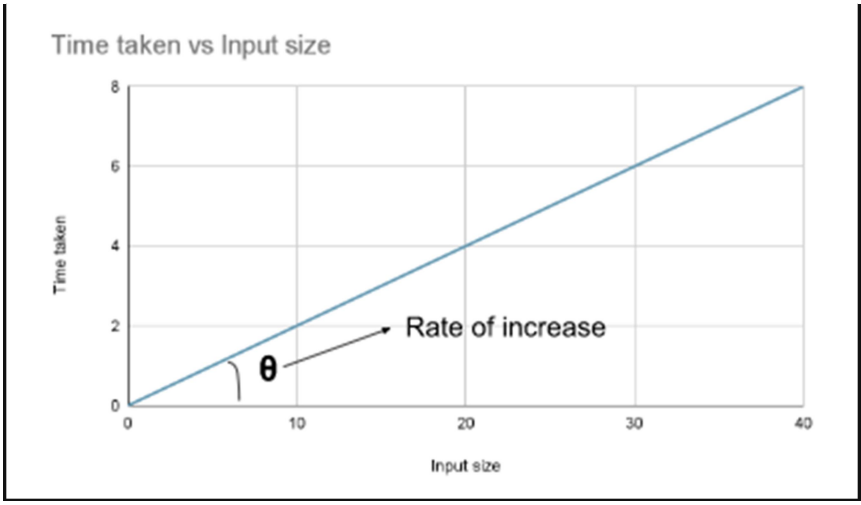
Let's understand why we should not judge any code on the basis of the time taken by a machine.

If we run the same code in a low-end machine(e.g. old windows machine) and in a high-end machine(e.g. Latest MacBook), we will observe that two different machines take different amounts of time for the same code. The high-end machine will take lesser time as compared to the low-end machine.

So, the time taken by a machine can be changed depending on the configuration. ***That is why we should not compare the two different codes on the basis of the time taken by a machine as the time is dependent on it.***

Definition:

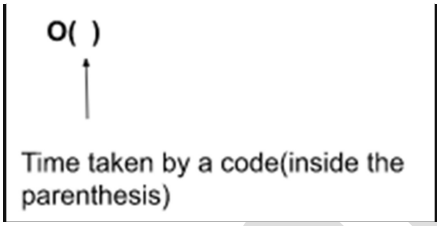
The rate at which the time, required to run a code, changes with respect to the input size, is considered the time complexity. **Basically, the time complexity of a particular code depends on the given input size, not on the machine used to run the code.**
Let's understand this using the following diagram:



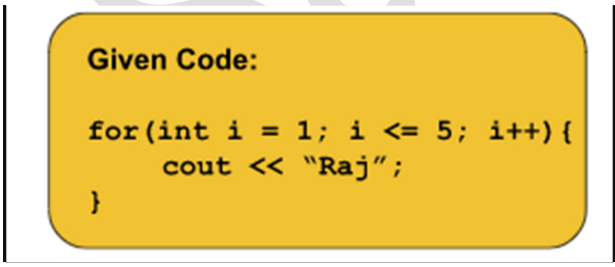
Now, the next question that comes to our mind is how we will represent the time complexity of a code as we are not going to use the standard units like minutes or seconds. Let's discuss it below:

How we will represent the time complexity of any code:

To represent the time complexity, we generally use the Big O notation. The Big O notation looks like the following:



Let's understand this using the following example:



The time complexity for this code will be nothing but the number of steps, this code will take to be executed. So, if we write this in terms of Big O notation, it will be like O(no. of steps).

Let's observe the steps for this code:

1. First, the assigning step(i = 1) will be done.

2. The second step will be the comparison i.e. i <= 5.
3. The third step will be the print statement (i.e. cout << "Raj";).
4. The fourth step will be the increment(i.e. i++).
5. In the fifth step, the updated value of i will be again checked i.e. the comparison(i <= 5).
6. In the sixth step, the print statement will be executed and so on.

This flow will continue until the value of i becomes greater than 5(i.e. 6). In a broader sense, we can observe that the 'for loop' will run 5 times and for each time three steps will be surely executed i.e. checking/comparison, printing, and increment. So, the total steps will be 5*3 = 15. And the time complexity in terms of Big O notation will be O(15).

Now, if we write N instead of 5, the number of steps will be then N*3 = 3N and the time complexity will be O(3*N).

But this manual counting process is not feasible for any code. As the 'for loop' might run a billion or million times and inside that 'for loop', there might be a large no. of operations or some other 'for loops' as well. So, we have to find out a better approach to calculate the time complexity of any given code.

Here come the three rules, that we are going to follow while calculating the time complexity:

1. We will always ***calculate the time complexity for the worst-case scenario***.
2. We will ***avoid including the constant terms***.
3. We will also ***avoid the lower values.***

Let's discuss the rules individually:

1. **Calculate the time complexity for the worst-case scenario:**

Before discussing the point we need to understand the three terms i.e. Best Case, Worst Case, and Average Case.

Let's understand these three terms considering the following piece of code:

```
if(marks < 25)       cout << "grade D";
else if(marks < 45)  cout << "grade C";
else if(marks < 65)  cout << "grade B";
else                 cout << "grade A";
```

1. **Best Case:** This term refers to the case where the code takes the least amount of time to get executed. For example, if the mark is 10(i.e. < 25), only the first line will be executed and the rest of the lines will be skipped. So, the least amount of steps i.e. only 2 steps are required in this case. This is an example of the best case.

2. **Worst Case:** This term refers to the case where the code takes the maximum amount of time to get executed. For examTime and Space Complexity - Sampanns A2Z DSA Course

3. 2862

4. In this article, we are going to discuss the terms time complexity and space complexity. Here, we will learn the basics of them and in the upcoming part, we will learn complex algorithms with a detailed discussion of their time complexity and space complexity.

5. What is Time Complexity?

6. We can solve a problem using different logic and different codes. Time complexity basically helps to judge different codes and also helps to decide which code is better. In an interview, an interviewer generally judges a code by its time complexity.

7. Now, the term, time complexity, seems that it is referring to the time taken by a machine to execute a particular code. But in real life, Time complexity does not refer to the time taken by the machine to execute a particular code.

8. Let's understand why we should not judge any code on the basis of the time taken by a machine.

9. If we run the same code in a low-end machine(e.g. old windows machine) and in a high-end machine(e.g. Latest MacBook), we will observe that two different machines take different amounts of time for the same code. The high-end machine will take lesser time as compared to the low-end machine.

10. So, the time taken by a machine can be changed depending on the configuration. That is why we should not compare the two different codes on the basis of the time taken by a machine as the time is dependent on it.

11. Definition:

12. The rate at which the time, required to run a code, changes with respect to the input size, is considered the time complexity. Basically, the time complexity of a particular code depends on the given input size, not on the machine used to run the code.

13. Let's understand this using the following diagram:

14.

15. Now, the next question that comes to our mind is how we will represent the time complexity of a code as we are not going to use the standard units like minutes or seconds. Let's discuss it below:

16. How we will represent the time complexity of any code:

17. To represent the time complexity, we generally use the Big O notation. The Big O notation looks like the following:

18.

19. Let's understand this using the following example:

20.

21. The time complexity for this code will be nothing but the number of steps, this code will take to be executed. So, if we write this in terms of Big O notation, it will be like O(no. of steps).

22. Let's observe the steps for this code:

23. First, the assigning step(i = 1) will be done.

24. The second step will be the comparison i.e. i <= 5.

25. The third step will be the print statement (i.e. cout << "Raj";).

26. The fourth step will be the increment(i.e. i++).

27. In the fifth step, the updated value of i will be again checked i.e. the comparison(i <= 5).

28. In the sixth step, the print statement will be executed and so on.

29. This flow will continue until the value of i becomes greater than 5(i.e. 6). In a broader sense, we can observe that the 'for loop' will run 5 times and for each time three steps will be surely executed i.e. checking/comparison, printing, and increment. So, the total steps will be 5*3 = 15. And the time complexity in terms of Big O notation will be O(15).

30. Now, if we write N instead of 5, the number of steps will be then N*3 = 3N and the time complexity will be O(3*N).

31. But this manual counting process is not feasible for any code. As the 'for loop' might run a billion or million times and inside that 'for loop', there might be a large no. of operations or some other 'for loops' as well. So, we have to find out a better approach to calculate the time complexity of any given code.

32. Here come the three rules, that we are going to follow while calculating the time complexity:

33. We will always calculate the time complexity for the worst-case scenario.

34. We will avoid including the constant terms.

35. We will also avoid the lower values.

36. Let's discuss the rules individually:

37. Calculate the time complexity for the worst-case scenario:

38. Before discussing the point we need to understand the three terms i.e. Best Case, Worst Case, and Average Case.

39. Let's understand these three terms considering the following piece of code:

40.

41. Best Case: This term refers to the case where the code takes the least amount of time to get executed. For example, if the mark is 10(i.e. < 25), only the first line will be executed and the rest of the lines will be skipped. So, the least amount of steps i.e. only 2 steps are required in this case. This is an example of the best case.

42. Worst Case: This term refers to the case where the code takes the maximum amount of time to get executed. For example, if the mark is 70(i.e. > 65), the last line will be executed after checking all the above conditions. So, the maximum amount of steps i.e. 4 steps are required in this case. This is an example of the worst case.

43. Average Case: This term is pretty self-explanatory. This is basically the case between the best and the worst.

44. Now, as we always want that our system serves the maximum number of clients, we need to calculate the time complexity for the worst-case scenario. With this, we can actually judge the robustness of any code or any system.

45. 2. Avoid including the constant terms:

46. Let's understand this rule considering the time complexity: O(4N3 + 3N2 + 8). Now, if we consider the value of N as 105 the time complexity will be like this: O(4*1015 + 3*1010 + 8). In this case, the constant term 8 is very less significant in terms of changing the time complexity with different values of N. That is why we should avoid the constant terms while calculating the time complexity.

47. If we want to think of this case in terms of code, we can consider the following code:

48.

49. Here, the first step (i.e. int x = 2) will be executed in unit time i.e. constant time. The precise time complexity is O(3N + 1) but in this case, the constant 1 is very less significant. So we will write the time complexity as O(3N) avoiding the constant term.

50. 3. Avoid the lower values:

51. Now, in the previous example, the given time complexity is O(4N3 + 3N2 + 8) and we have reduced it to O(4N3 + 3N2). Here, we can clearly observe if the value of N is a large number, the second term i.e. 3N2 will also be a less significant term. For example, if the value of N is 105 then the term 3*1010 becomes less significant with respect to 4*1015. So, we can also avoid the lower values and the final time complexity will be O(4N3)

52. Note: A point to remember is that we can actually ignore the constant coefficients as well. For example, considering the time complexity O(4N3) as O(N3) is also correct.

53. Apart from the widely used Big O notation, there are several other notations. Among them, the two most common are the Theta notation(θ) and the Omega notation(Ω). The differences are shown in the below table:

54.

55. These concepts are not much important from the interview perspective and so here we are not going to discuss these in detail. Please follow any standard textbook if you want the details and the mathematical derivations.

56. Let's quickly discuss some questions to make the concepts clear:

57. Question 1:

58. Given the following code:

59.

60. In order to calculate the time complexity of the code, we need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1 i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to N-1 i.e. N times. The following illustration depicts the process:

61.

62. Now, we can clearly observe the total number of steps i.e. N+N+N+N+.....+N times = N*N = N2. So, the time complexity will be O(N2). We can ignore other constant steps as well as the innermost block of code as it runs in constant time.

63. Question 2:

64. Given the following code:

65.

66. In order to calculate the time complexity of the code, we again need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1 i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to i i.e. (i+1) times. The following illustration depicts the process:

67.

68. Now, we can clearly observe the total number of steps i.e. 1+2+3+4+.....+N. Now we know the formula of the summation of the first N natural numbers i.e. (N*(N+1))/2 = N2/2 + N/2. So, the precise time complexity will be O(N2/2 + N/2). Now, we should ignore the lower values. So, the time complexity will be O(N2/2). It can be also written as O(N2) avoiding the coefficient 1/2.

69. These are the basics of time complexity. Now, let's move on to the space complexity part.

70. What is Space Complexity?

71. The term space complexity generally refers to the memory space that a code uses while being executed. Again space complexity is also dependent on the machine and so we are going to represent the space complexity using the Big O notation instead of using the standard units of memory like MB, GB, etc.

72. Definition:

73. Space complexity generally represents the summation of auxiliary space and the input space. Auxiliary space refers to the space that we use additionally to solve a problem. And input space refers to the space that we use to store the inputs.

74. Let's understand this using the following example:

75.

76. The variables a and b are used to store the inputs but c refers to the space we are using to solve the problem and c is the auxiliary space. Here the space complexity will be O(3).

77. Similarly, if we use an array of size n, the space complexity will be O(N).

78. Good coding practice:

79. If a question of adding two numbers like a and b is asked, one of the possible methods will be

80. b = a+b. In this case, the space complexity is definitely reduced as we are not using any extra variable but this is not a good practice to manipulate the given inputs or data. In an interview, we must be careful that we will not manipulate the given data even if the space complexity becomes 2N instead of N. If the interviewer specifically instructs us to manipulate, then only we should attempt this method.

81. Note: A company may use the same data for different purposes. That is why we should not attempt to manipulate the given data for reducing the space complexity. So, we will never manipulate the given data i.e. the inputs until the interviewer specifically says so.

82. We are now pretty much done with our concepts of time complexity and space complexity. Now, we will briefly discuss some points about competitive programming or the online judge.

83. Points to remember:

84. In competitive programming or in the platforms like Leetcode and GeeksforGeeks, we generally run our codes on online servers. Most of these servers execute roughly 108 operations in approximately 1 second i.e. 1s. We must be careful that if the time limit is given as 2s the operations in our code must be roughly 2*108, not 1016. Similarly, 5s refers to 5*108. Simply, if we want our code to be run in 1s, the time complexity of our code must be around O(108) avoiding the constants and the lower values.

85. ple, if the mark is 70(i.e. > 65), the last line will be executed after checking all the above conditions. So, the maximum amount of steps i.e. 4 steps are required in this case. This is an example of the worst case.

86. **Average Case:** This term is pretty self-explanatory. This is basically the case between the best and the worst.

Now, as we always want that our system serves the maximum number of clients, we need to calculate the time complexity for the worst-case scenario. With this, we can actually judge the robustness of any code or any system.

2. Avoid including the constant terms:

Let's understand this rule considering the time complexity: $O(4N^3 + 3N^2 + 8)$. Now, if we consider the value of N as 10^5 the time complexity will be like this: $O(4*10^{15} + 3*10^{10} + 8)$. In this case, the constant term 8 is very less significant in terms of changing the time complexity with different values of N. That is why we should avoid the constant terms while calculating the time complexity.

If we want to think of this case in terms of code, we can consider the following code:

```
int x = 2;
for(int i = 1; i <= N; i++){
    cout << "raj";
}
```

Here, the first step (i.e. int x = 2) will be executed in unit time i.e. constant time. The precise time complexity is O(3N + 1) but in this case, the constant 1 is very less significant. So we will write the time complexity as O(3N) avoiding the constant term.

3. Avoid the lower values:

Now, in the previous example, the given time complexity is $O(4N^3 + 3N^2 + 8)$ and we have reduced it to $O(4N^3 + 3N^2)$. Here, we can clearly observe if the value of N is a large number, the second term i.e. $3N^2$ will also be a less significant term. For example, if the value of N is 10^5 then the term $3*10^{10}$ becomes less significant with respect to $4*10^{15}$. So, we can also avoid the lower values and the final time complexity will be $O(4N^3)$

Note: *A point to remember is that we can actually ignore the constant coefficients as well. For example, considering the time complexity $O(4N^3)$ as $O(N^3)$ is also correct.*

Apart from the widely used Big O notation, there are several other notations. Among them, the two most common are the Theta notation(θ) and the Omega notation(Ω). The differences are shown in the below table:

Big O notation	Theta notation(θ)	Omega notation(Ω)
Represents the worst-case time complexity i.e. the upper bound.	Represents the average-case time complexity.	Represents the best-case time complexity i.e. the lower bound.

These concepts are not much important from the interview perspective and so here we are not going to discuss these in detail. Please follow any standard textbook if you want the details and the mathematical derivations.

Let's quickly discuss some questions to make the concepts clear:

Question 1:

Given the following code:

```
for(int i = 0; i < N; i++){
    for(int j = 0; j < N; j++){

        // Block of code
        // that runs in constant time.
    }
}
```

In order to calculate the time complexity of the code, we need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1 i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to N-1 i.e. N times. The following illustration depicts the process:

```
for i = 0, j = [0, 1, 2, 3, ……, N-1]
for i = 1, j = [0, 1, 2, 3, ……, N-1]
for i = 2, j = [0, 1, 2, 3, ……, N-1]
for i = 3, j = [0, 1, 2, 3, ……, N-1]
for i = 4, j = [0, 1, 2, 3, ……, N-1]
for i = 5, j = [0, 1, 2, 3, ……, N-1]
.
.
.
for i = N-1, j = [0, 1, 2, 3, ……, N-1]
```

Now, we can clearly observe the total number of steps i.e. N+N+N+N+.....+N times = N*N = N^2. So, **the time complexity will be $O(N^2)$**. We can ignore other constant steps as well as the innermost block of code as it runs in constant time.

Question 2:

Given the following code:

```
for(int i = 0; i < N; i++){
    for(int j = 0; j <= i; j++){

        // Block of code
        // that runs in constant time.
    }
}
```

In order to calculate the time complexity of the code, we again need to first observe how the loops are working. The outer loop i.e. i runs from 0 to N-1 i.e. N times and for every value of i, the inner loop i.e. j also runs from 0 to i i.e. (i+1) times. The following illustration depicts the process:

```
for i = 0, j = [0] // 1 time
for i = 1, j = [0, 1] // 2 times
for i = 2, j = [0, 1, 2] // 3 times
for i = 3, j = [0, 1, 2, 3] // 4 times
for i = 4, j = [0, 1, 2, 3, 4] // 5 times
for i = 5, j = [0, 1, 2, 3, 4, 5] // 6 times
.
.
.
for i = N-1, j = [0, 1, 2, 3, ……, N-1] // N times
```

Now, we can clearly observe the total number of steps i.e. 1+2+3+4+.....+N. Now we know the formula of the summation of the first N natural numbers i.e. (N*(N+1))/2 = $N^2/2$ + N/2. So, the precise time complexity will be O($N^2/2$ + N/2). Now, we should ignore the lower values. So, **the time complexity will be O($N^2/2$)**. It can be also written as **O(N^2)** avoiding the coefficient 1/2.

These are the basics of time complexity. Now, let's move on to the space complexity part.

What is Space Complexity?

The term space complexity generally refers to the memory space that a code uses while being executed. Again space complexity is also dependent on the machine and so we are going to **represent the space complexity using the Big O notation** instead of using the standard units of memory like MB, GB, etc.

Definition:

Space complexity generally represents the summation of auxiliary space and the input space. Auxiliary space refers to the space that we use additionally to solve a problem. And input space refers to the space that we use to store the inputs.

Let's understand this using the following example:

```
Input(a)//1 Input space
Input(b)//1 Input space
c = a+b
//c-> 1 auxiliary space
```

The variables a and b are used to store the inputs but c refers to the space we are using to solve the problem and c is the auxiliary space. Here the space complexity will be O(3).

Similarly, if we use an array of size n, the space complexity will be O(N).

Good coding practice:

If a question of adding two numbers like a and b is asked, one of the possible methods will be

b = a+b. In this case, the space complexity is definitely reduced as we are not using any extra variable but this is not a good practice to manipulate the given inputs or data. In an interview, we must be careful that we will not manipulate the given data even if the space complexity becomes 2N instead of N. If the interviewer specifically instructs us to manipulate, then only we should attempt this method.

Note: *A company may use the same data for different purposes. That is why we should not attempt to manipulate the given data for reducing the space complexity. So, we will never manipulate the given data i.e. the inputs until the interviewer specifically says so.*

We are now pretty much done with our concepts of time complexity and space complexity. Now, we will briefly discuss some points about competitive programming or the online judge.

Points to remember:

In competitive programming or in the platforms like Leetcode and GeeksforGeeks, we generally run our codes on online servers. Most of these servers execute roughly 10^8 operations in approximately 1 second i.e. 1s. We must be careful that if the time limit is given as 2s the operations in our code must be roughly $2*10^8$, not 10^{16}. Similarly, 5s refers to $5*10^8$. Simply, if we want our code to be run in 1s, the time complexity of our code must be around $O(10^8)$ avoiding the constants and the lower values.