# Time complexity

[[Kapil Lanjewar](https://medium.com/@kapil.lanjewar.100?source=post_page-----9a47408d188f--------------------------------)](https://medium.com/@kapil.lanjewar.100?source=post_page-----9a47408d188f--------------------------------)

[Kapil Lanjewar](https://medium.com/@kapil.lanjewar.100?source=post_page-----9a47408d188f--------------------------------)

4 min read

· A means of measuring the efficiency of an algorithm.

· A means of comparing different algorithms achieving same result.

· Also measured in number of operations performed in an algorithm to achieve specific task.

Asymptotic notations used to express time complexity

*Big-O: O*

· Predicts worst case time complexity.

· Predicts maximum time an algorithm needs considering all possible input values.

· One can see an algorithm grows slower than or equal to what is predicted by expression.

*Omega: Ω*

· Predicts best case time complexity.

· Predicts minimum time an algorithm needs considering all possible input values.

· One can see an algorithm grows faster than or equal to what is predicted by expression.

*Theta: θ*

· Predicts average case time complexity.

· One can see an algorithm can be predicted by both Big-O as well as Omega notations.

Let’s talk about some of the time complexities

**Note: We will only talk about worst case time complexity henceforth, i.e. expressed in Big-O.**

*O(1):*

def constantTime(array):  
 i = array.pop()  
 print("-> ", i)  
  
  
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
constantTime(array)

*Output:*

*-> 10*

Observations:

1. Such algorithm runs in constant time, meaning operation will always complete in constant time irrespective of input size.
2. Alternatively, as the input size increases, the cost of such algorithms or number of operations required by all those algorithms to achieve the result stays unchanged or constant making it the most efficient.
3. It also means such algorithms will achieve/complete their task in precisely one operation denoted as O(1).

*O(n):*

def ntimes(t, n):  
 for i in range(n):  
 print(t, i)  
  
ntimes("-> ", 3)

*Output:*

*-> 0*

*-> 1*

*-> 2*

Observations:

1. You can see we are printing 3 items in a for loop.

2. Each item is visited once for printing it.

3. If we generalize the above algorithm for n items to be printed, we will have to visit all n items performing n number of operations.

4. Therefore, the time complexity for this algorithm is expressed as O(n).

5. As the input size grows, the cost of such algorithms or the number of operations required by all those algorithms to achieve/complete the task also grows proportionally or linearly.

*O(n****²****):*

def nsquare(t, n):  
 for i in range(n):  
 for j in range(n):  
 print(t, i, j)  
  
  
nsquare("-> ", 3)

*Output:*

*-> 0 0*

*-> 0 1*

*-> 0 2*

*-> 1 0*

*-> 1 1*

*-> 1 2*

*-> 2 0*

*-> 2 1*

*-> 2 2*

Observations:

1. You can see we are printing 3 items, and for each item we are printing all 3 items again.

2. The outer loop prints 3 items, and the inner loop prints all 3 items again for each iteration of outer loop.

3. So in total we are making 9 operations which is (3 \* 3 = 9) or (3**²** = 9).

4. If we generalize above algorithm for n items, we will have to perform n**²** operations.

5. Therefore, the time complexity for this algorithm is O(n**²**).

6. As the number of input size grows, the cost of such algorithms or number of operations required by all those algorithms to achieve/complete the task also grows quadratically.

*O(logn):*

def logn(a, i, e, s):  
 print(i, e)  
 if e > i:  
  
 m = int((i + e) / 2)  
  
 if s == a[i]:  
 return i  
  
 if s == a[e]:  
 return e  
  
 if s <= a[m]:  
 e = m  
 else:  
 i = m + 1  
  
 return logn(a, i, e, s)  
  
 else:  
 return -1  
  
  
# ------------------------------------------------------------>  
  
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
start = 0  
end = len(array)-1  
search = 9  
  
result = logn(array, start, end, search)  
  
  
# ------------------------------------------------------------>  
  
print("======================")  
  
if result == -1:  
 print("Not Found")  
else:  
 print("Index ", result)  
 print("Searched element ", array[result])  
  
print("======================")

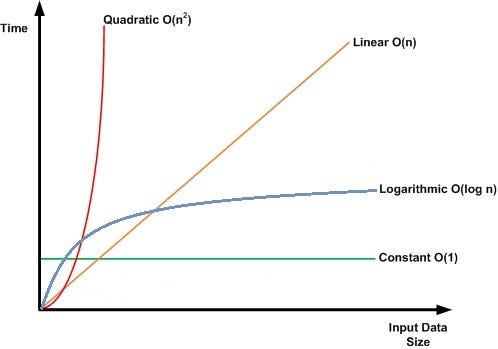
*Output:*

*0 9  
5 9  
8 9  
======================  
Index 8  
Searched element 9  
======================*

Observations:

1. Pre-requisite to achieve such algorithm is that you would need sorted and unique elements in the collection to be used as an input.
2. This is the most efficient algorithm after one that leads to constant time rate of growth.
3. The reason behind efficiency of algorithm is that it reduces the search space by half each iteration and then compare the element to be searched.
4. The methodology used by such algorithms is known as Divide and Conquer.
5. As the number of input size grows, the cost of such algorithms or the number of operations required by all those algorithms to achieve/complete the task grows very slowly.

# Growth rate graph



Growth rate of algorithms against growing input size

[Algorithms](https://medium.com/tag/algorithms?source=post_page-----9a47408d188f---------------algorithms-----------------)

[Python](https://medium.com/tag/python?source=post_page-----9a47408d188f---------------python-----------------)