Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Binary Search is great to search through large sorted arrays.It has a time complexity of **O(log n)**

0(1) time complexity return n*n

of N. When N doubles, the running time increases by N * N.

```
while(low <= high)

{

    mid = (low + high) / 2;

    if (target < list[mid])

        high = mid - 1;

    else if (target > list[mid])

        low = mid + 1;

    else break;

}
```

Copy

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)

{

    int pivot = partition(list, left, right);

    quicksort(list, left, pivot - 1);

    quicksort(list, pivot + 1, right);

}
```

Copy

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Big O Notation	Name	Example(s)
O(1)	Constant	# Odd or Even number, # Look-up table (on average)
O(log n)	Logarithmic	# Finding element on sorted array with **binary search**
O(n)	Linear	# Find max element in unsorted array, # Duplicate elements in array with Hash Map
O(n log n)	Linearithmic	# Sorting elements in array with **merge sort**
O(n^2)	Quadratic	# Duplicate elements in array **(naïve)**, # Sorting array with **bubble sort**
O(n^3)	Cubic	# 3 variables equation solver
O(2^n)	Exponential	# Find all subsets

Sorting

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)	O(nk)
Radix Sort	Array	O(nk)	O(nk)	O(nk)	O(n+k)

priority_queue<data_type, vector<data_type>, greater<data_type>> Q

The table containing the time and space complexity with different functions given below:

Function	Time Complexity	Space Complexity
Q.top()	O(1)	O(1)
Q.push()	O(log n)	O(1)
Q.pop()	O(log n)	O(1)
Q.empty()	O(1)	O(1)

STL Containers

Containers library

1. Sequence containers
Sequence containers implement data structures which can be accessed sequentially.

1. array : (C++11) static contiguous array
2. vector : dynamic contiguous array
3. deque : double-ended queue
4. forward_list (C++11) : singly-linked list
5. list : doubly-linked list

2. Associative containers
Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

1. set : collection of **unique keys**, sorted by keys
2. map : collection of key-value pairs, sorted by keys, **keys are unique**
3. multiset : collection of keys, sorted by keys
4. multimap : collection of key-value pairs, sorted by keys

3. Unordered associative containers
Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

1. unordered_set : collection of unique keys, hashed by keys
2. unordered_map : collection of key-value pairs, hashed by keys, keys are unique
3. unordered_multiset : collection of keys, hashed by keys
4. unordered_multimap : collection of key-value pairs, hashed by keys

4. Container adaptors
Container adaptors provide a different interface for sequential containers.

1. stack
2. queue
3. priority_queue

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

The time complexity, in Big O notation, for each function:

Determining complexity for recursive functions (Big O notation)

```
int recursiveFun1(int n)
{
  if (n <= 0)
    return 1;
  else
    return 1 + recursiveFun1(n-1);
}
```

This function is being called recursively n times before reaching the base case so its O(n), often called **linear**.

```
int recursiveFun2(int n)
{
  if (n <= 0)
    return 1;
```

```
  else
    return 1 + recursiveFun2(n-5);
}
```

This function is called n-5 for each time, so we deduct five from n before calling the function, but n-5 is also O(n). (Actually called order of n/5 times. And, O(n/5) = O(n)).

```
int recursiveFun3(int n)
{
  if (n <= 0)
    return 1;
  else
    return 1 + recursiveFun3(n/5);
}
```

This function is log(n) base 5, for every time we divide by 5 before calling the function so its O(log(n))(base 5), often called **logarithmic** and most often Big O notation and complexity analysis uses base 2.

```
void recursiveFun4(int n, int m, int o)
{
  if (n <= 0)
  {
    printf("%d, %d\n",m, o);
  }
  else
  {
    recursiveFun4(n-1, m+1, o);
    recursiveFun4(n-1, m, o+1);
  }
}
```

Here, it's O(2^n), or **exponential**, since each function call calls itself twice unless it has been recursed **n** times.

```
int recursiveFun5(int n)
{
  for (i = 0; i < n; i += 2) {
    // do something
  }

  if (n <= 0)
    return 1;
  else
    return 1 + recursiveFun5(n-5);
}
```

And here the for loop takes n/2 since we're increasing by 2, and the recursion takes n/5 and since the for loop is called recursively, therefore, the time complexity is in

$(n/5) * (n/2) = n^2/10$,

due to Asymptotic behavior and worst-case scenario considerations or the upper bound that big O is striving for, we are only interested in the largest term so $O(n^2)$.