Ascending order of General Time Complexities

In [ ]:
```
Ascending order of General Time Complexities --> O(1) , O(Log n), O(n),O(nlog n),O(n^2),O(n^3),O(2^n)
```

Python Program to find the sum of N natural Number

In [1]:
```
n=int(input())
sum=0
for i in range(0,n+1):
    sum=sum+i
print(sum)

#Time Complexity for the above code is O(n) which is more we can also optimize it
```
```
10
55
```

Optimized Code

In [ ]:
```
n=int(input())
sum=(n*(n+1))//2    #O(1)
print(sum)

#Time Complexity of this code is O(1) which is the least one and furthur we cannot optimized it.
```

Searching

In [ ]:
```
--> The process of finding the desired information from the set of items stored in the form of elements in the
    computer memory is referred to as 'searching in data structure'.
```

Types of Searching Algorithms

In [ ]:
```
Two types of searching algorithms:
    1.Linear Search  --> O(N) --> Sorted as well as Unsorted
    2.Binary Search  --> O(log N) --> Sorted Array
```

Linear Search Algorithm

In [ ]:
```
--> Linear search is the simplest method for searching.
--> In Linear search technique of searching; the element to be found in searching the elements to  be found is searched
    sequentially in the list.
--> This method can be performed on a sorted or an unsorted list (usually arrays).
```

Implementation of Linear Search Algorithm

In [2]:
```
def linear_search(array,key):  #5
    for i in range(len(array)):  #O(n)
        if array[i]==key:
            print("Element found at index :",i)
            break
    else:
        return "Element not found"

array=[10,20,30,40,50]
key=20
linear_search(array,key)
```
```
Element found at index : 1
```

Discussion on Time Complexity of Linear Search

Best Case(Big Omega) of Linear Search

In [ ]:
```
--> The element being searched could be found in the first position.
--> In this case, the search ends with a single successful comparison.
--> Thus, in the best-case scenario, the linear search algorithm performs O(1) operations.
```

Average Case(Big Theta) of Linear Search

In [ ]:
```
--> When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is O(n).
```

Worst Case(Big O) of Linear Search

In [ ]:
```
--> The element being searched may be at the last position in the array or not at all.
--> In the first case, the search succeeds in ‘n’ comparisons.
--> In the next case, the search fails after ‘n’ comparisons.
--> Thus, in the worst-case scenario, the linear search algorithm performs O(n) operations.
```

Binary Search Algorithm

In [ ]:
```
--> Binary searches are efficient algorithms based on the concept of “divide and conquer” that improves the search by
    recursively dividing the array in half until you either find the element or the list gets narrowed down to one piece
    that doesn’t match the needed element.

--> Binary searches work under the principle of using the sorted information in the array to reduce the time complexity
    to zero (Log n).
```

The binary search approach’s basic steps:

In [ ]:
```
--> Sort the array in ascending order.
--> Set the low index to the first element of the array and the high index to the last element.
--> Set the middle index to the average of the low and high indices.
--> If the element at the middle index is the target element, return the middle index.
--> If the target element is less than the element at the middle index, set the high index to the middle index – 1.
--> If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
--> Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.
```

Implementation of Binary Search

In [4]:
```
def binary_search(array,key):
    low=0
    high = len(array)-1
    mid=0
    while low<=high:
        mid=(high+low)//2
        if array[mid]<key:
            low=mid+1
        elif array[mid]>key:
            high=mid-1
        else:
            return mid
    return -1

array = [1,2,3,4,5,6]
key=6
x=binary_search(array,key)
if x!=-1:
    print("Element present at index :",x)
else:
    print("Element not present")
```
```
Element present at index : 5
```

Analysis of input size at each iteration of Binary Search:

In [ ]:
```
At Iteration 1:

    Length of array = n

At Iteration 2:

    Length of array = n/2

At Iteration 3:

    Length of array = (n/2)/2 = n/22

Therefore, after Iteration k:

    Length of array = n/2k

Also, we know that after k iterations, the length of the array becomes 1 Therefore, the Length of the array

     n/2k = 1
     => n = 2k

Applying log function on both sides:

    => log2n = log22k

    => log2n = k * log22

    As (loga (a) = 1) Therefore, k = log2(n)
```

Discussion of time Complexity of Binary Search

Best Case Time Complexity(Big Omega)

In [ ]:
```
--> The best time complexity of binary search occurs when the required element is found in the first comparison itself,
    and only one iteration occurs. Therefore we use O(1).
--> Essentially for this case, the element needs to be in the exact middle of the list because, in binary search,
    the first competition occurs with the middle element. Once the middle element does not return the correct answer,
    the iteration begins for the lesser half of the greater half.
```

Worst Case Time Complexity(Big O)

In [ ]:
```
--> The worst time complexity of binary search occurs when the element is found in the very first index or the very last
    index of the array. For this scenario, the number of comparisons and iterations required is logn, where n is the
    number of elements in the array. It is called the worst time complexity because it consumes a lot of time for large
    arrays containing hundreds and thousands of values. Accordingly, hundreds and thousands of iterations must occur.
```

Average Case Time Complexity(Big Theta)

In [ ]:
```
The average case Time Complexity in Binary Search  is n* logn / (n+1), which is approximately logn.
```

Summary of Data Structures and Algorithms

In [ ]:
```
Data Structures: is a way of storing the data so that we can use that data in an  effiicent manner for our furthur use.

    Types of Data Structures:

    1. Primitve Data Structure(Hold a single value) --> int,float,long,bool complex.....
    2. Non Primitive Data Structure(Hold multiple values)

    --> Based on Implemetaion
                         -->  Physical
                                  --> Array linkedlist
                         -->  logical(ADT)
                                  --> Stack and Queue,Graph and Trees

    --> Based on Storage
                         --> Linear Data Structures
                                   --> Array,linkedlist, Stack and Queue
                         --> Non Linear Data Structures
                                   --> Trees and Graphs


Linkedlist: Linkedlist is a linear data structure. It is collection of nodes and each node is having
    data and next (Memory location of next element)

    Time Complexity of Linkedlist:
        accessing any element of a linkedlist --> O(n)
        Traversing over linkedlist --> O(N)
        Insertion at first position --> O(1)
        Insertion at given position --> O(N)
        Insertion at last --> O(N)
        Deletion at first position --> O(1)
        Deletion at given position --> O(N)
        Deletion at last --> O(N)

   Operations on linkedlist:
        1.Insertion --> Insertion  at begin , Insertion at end , Insertion at given pos
        2.Deletion --> Deletion at begin , Deletion at end , Deletion at given pos
        3.Traversal

Array: Array is a linear Data Structure in which elements are stored at contigous memeory location.

    Accessing any element of array --> O(1)
    Traversing over array --> O(N)
    Insertion at first position --> O(1)
    Insertion at given position --> O(N)
    Insertion at last --> O(N)
    Deletion at first pos --> o(1)
    Deletion at given pos --> o(N)
    deletion at last --> o(n)

    Operations on Array:
        1.Insertion --> Insertion  at begin , Insertion at end , Insertion at given pos
        2.Deletion --> Deletion at begin , Deletion at end , Deletion at given pos
        3.Traversal


Stack : Stack is a linear data structure which is following the principle of LIFO(Last in first out)
      : Insertion and Deletion in stack will be done from one end(Top of stack)

      Terminologies and Operations of stack:
            1.Push --> insertion at top
            2.Pop  --> Deletion from top
            3.Peek --> Getting Top Value

     Time Complexities of stack:
        push --> O(1)  --> O(N)
        pop -->  O(1)
        peek --> O(1)


Queue : Queue is a linear data structurw which si following the principle of FIFO(First in first out).
      : Insertion will be done from Rear end and deletion will be done from front end

    Terminologies and Operations of queue:
            1.Enqueue --> insertion at Rear --> o(1)
            2.Dequeue  --> Deletion from Front -->  o(1)
            3.PeekinQueue --> Getting front Valu

    Time Complexity of Queue:

        1.Enqueue --> O(1)
        2.Dequeue --> O(1)
        3.PeekinQueue --> O(1)

Never ending Queue(Circular Queue) -->By linkedlist only
```