What is a time complexity ?
" how long an algorithm takes to run based on the size of the input."

It does not measure time in seconds , because seconds can be change from ,
Computer to computer ,internet speed, processor speed ,
Instead we can measure time complexity in **number of operation**

How do we measure ?
we use **Big- O** notation like:

- O(1)-> constant time
- O(n) ->linear time
- O(log n)-->logarithmic times
- O(n^2)--> Quadratic time
- O(n^3)-->Cubic
- O(2^n)--> exponential
- O(n!) --> Factorial

Time complexity tell us how fast or slow an algorithm grows when the input size is increase
We do not count actual seconds
We count how many steps algorithm performs
Big O notation it can also shows worst case number of steps

12	8	20	9	15


```
A=[15,16,6,8,5]
n=4
For I in range(n-1):
{
   Flag=0;
     For j in range(n-i-1):
          {
               If A[j]>A[j+1]:
                    {
                    A[j],a[j+1]=A[j+1],a[j]
                    Flag=1
                    }
          }
          If(flag==0)
          Break;
}
```

N =?

- Pass 1 -->(n-1) comparison
- Pass 2 -->(n-2) comparisons
- .
- .
- .
- .
- Pass n-->1
- Total comparisons
- (n-1)+(n-2)+……….+1
  n(n-1)/2
  ..n2/2
  Time complexity of bubble sort
  O(n2) --> without optimization
  O(n)--> with optimization

Insertion Sort (

n=6

A n=6

5	4	10	1	6	2

Sorted sublist unsorted sublist--->

5	4	10	1	6	2

4	5	10	1	6	2

Time complexity :
Worst case:- list is given descending order wanted to make it as ascending order
O(n^2)
Best case is O(n)
2 3 5 6 8 9

4	5	10	1	6	2

4	5	?	10	6	2

4	?	5	10	6	2

1	4	5	10	6	2

1	4	5	6	10	2

1	4	5	6	?	10

1	4	5	?	6	10

1	4	?	5	6	10

1	?	4	5	6	10

1	2	4	5	6	10

```
for (i=1,i<n,i++)
{
     Temp= A[i]
     j=i-1;
While(j>=0 && A[J]>temp)
     {
     A[j+1]=A[j],
     j--;
     }
     A[j+1]=temp
}
```

Algorithm:
Step 1: start
Step 2: divide sorted and unsorted list
Step 3: assume first element is sorted
Step 4:pick the next element (key)
Step 5: compare it with sorted element previous to the unsorted list
Step 6: shift elements greater than key one position to the right side
Step 7:insert the key in the correct empty position
Step 8: repeat until array is fully sorted
Step 9: stop

Searching Algorithms:
Linear search and binary search

Linear Search :

A	0	1	2	3	4	5	6	7
	15	5	20	35	2	42	67	17

n=8

Searching element
Data=42

1 If element found then return the location
2 if element is not found

```
Int flag=0;
For(i=0;i<n;i++)
    {
        If(A[i]==Data)
        {
        Print the element found at ,i(index)
        Flag=1;
        Break;
        }

    }
            if (flag==0)
              {
              print element not found
              }
```

i=0 ;0<8;i++
A[0]==Data
15==42 f

i=1 ;1<8; i++
A[1]==data
5==42 f

i=2;2<8; i++
A[2]==data
20==42 f

i=3;3<8;i++
a[3]==data
35==42 fa

i=4;4<8;i++
A[4]==42;
2==42 f

i=5;5<8;i++
A[5]==data
42==42
Element found at 5

i=6;6<8;i++
A[6]==data
67==42 f

i=7;7<8;i++
A[7]==data
17==42 f
i=8;8<8;i++
False ended the loop

Time complexity:
Best case: O(1)
Worst case is O(n)
Average=1+2+3+...+n/n

$$\frac{\cancel{n}(n+1)}{2\cancel{n}} \qquad O\left(\frac{n+1}{2}\right)$$

Algorithm:
Step1 :start
Step2:read the Array
Step3: find the n value
Step4:read key, search data
Step5:flag=0,i=0
Step 6:I is value varies from 0 to n-1. for each value of I do step7
Step 7: check A[i] is equal to key. If yes set flag=1 and print element found ,otherwise goto step6;
Step 8: check flag=0 if yes data not found
Step9: stop

Binary search:

- Binary search is a fast searching algorithm
- It follows divide and conquer method
- The data is in **sorted** format only (ASC ,DSC)
- Binary search looks for particular search data by comparing middle most item of the collection

A,b,c,d,.....z
Dog=
......P......

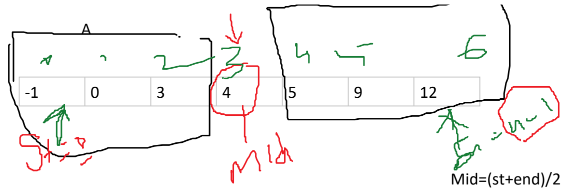
1274

Data=12

A

0	1	2	3	4	5	6
-1	0	3	4	5	9	12

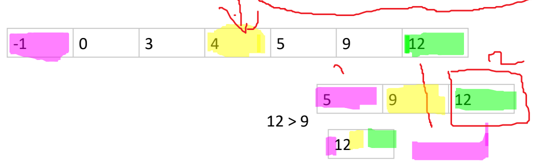
- The data is in **sorted** format only (ASC ,DSC)
- Binary search looks for particular search data by comparing middle most item of the collection
- If match occurs , index of item is return
- If the middle item is greater than the search data , then item is searched in the sub-array to the left of the middle item .
- Otherwise ,item is searched for the sub-array to the right side of the middle item.
- His process is continues on the sub-array as well until the size of the subarray reduces to zero



Start=0
End=n-1
Mid=(st+end)/2

end
starting
Mid point

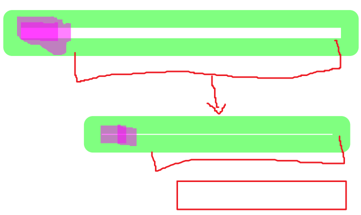
1 data>A[mid] --2nd half
ST=MID+1
2 data<A[mid] -->1st half
END=MID-1
3 data==A[mid] --> mid poinst answer




```
While (start<=end)
{
        Mid=(start + end)/2
        If(data > A[mid])
                Start=mid+1
        Else if( data < A[mid] )
             end=mid-1
        Else
        Return mid
}
Return -1
```

Recursion

Recursion is where a function can call itself to solve a smaller version of the same problem



Base case

Example: 4
N to 1

4	4 3 2 1	n=4
3	3 2 1	n=3
	2 1	n=2
2		
	1	n=1
1		

We have 2 essential parts of the recursion

1. Base case: stops the recursion --> prevent infinite loop
2. Recursive case : function calling itself with smaller input

Factorial for recursion :

n!=n * (n-1) * (n-2) *..... *1

f(n=4) = 4* f(n=3)
3 *f(n=2)
2 * f(n=1)
1 * f(n=0)=1 ans 1

```
def fact(n)
      If n==0;
            Return 1  //base case
      Return  n * fact(n-1) //recursive case
```

Caluculate fact(4) n=4

= 4 * fact(3)
= 4 * (3 * fact(2))
=4 * (3* (2 * fact(1)))

= 4 * (3* (2 *(1 * fact(0))
= 4* 3 * 2* 1* 1
=24

Fact(4)
  fact(3)
    Fact(2)
      fact(1)
        Fact(0)

Fibonacci

0,1,1,2,3,5,8,....

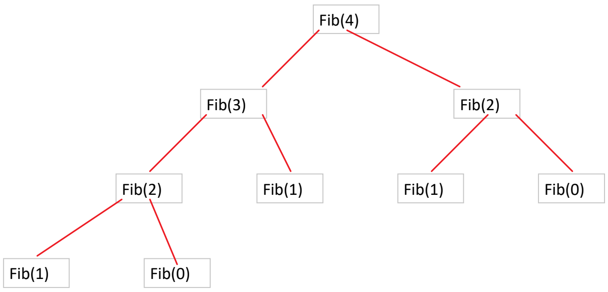
0,1
1+0=1
1+ 1=2
2+1=3
3+2=5
5+3=8 ....n

Formula=
f(n) =f(n-1)+f(n-2)

```
def fib(n)
    If n <= 1;
        Return n  //base case

    Return fib(n-1) + fib(n-2)
```

Calculate fib(4)



Fib(1)=1
Fib(0)=0

Fib(2) =1+0 =1
Fib(3) = fib(2) + fib(1) =1+1=2

Fib(4) =fib(3)+fib(2)=2+1 =3

Recursion is best for .

ds tree (binary tree traversal)
Backtracking (sudoku,maze)
Divide and conquer (sorting)
Mathematica functions

Time complexity
Space complexity
Recursive complexity (fib,factorial)

Time complexity:

How fast the code run when input grows
Example

For I in range (n) --> O(n)

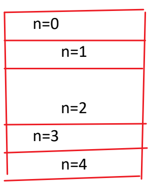
For I in range(n * n) --> O(n^2)

Binary search --> O (log n)

```
Def print_n(n):
    For I in range(n):
        Print(i)  //O(n)
```

Space complexity :

Space complexity tells how much extra memory your program uses



Sc=n *k

Sc=O(n)

x= 10 --> O(1) space

Arr= [1] * n // O(n) space

Recursive complexity (fib,factorial)

Time oplmexity --> O(n) ,space O(n) --fact

Fib-- O(2^n) , O(n)

Time complexity --> "how fast"

Space complexity --> "How much extra memory"

Recursive complexity --> "how many recursive calls ?"