Big O Notation

It allows us to measure the time and space complexity of code.

Idea of Algorithmic efficiency

- An algorithm is a method for accomplishing a specific task.

Program to add two numbers

- A=5
- B=7
- C=A+B
- print(c)

- It is important to measure efficiency of algorithm before applying it on large scale.

```
A=[1,2,3.4]
S=0
for i in X:
        S=S+i
print(s)
```

Factors of Performance of Algorithm

- Internal Factors- time required to run and memory required to run
- External factors- size of input to the algorithm , speed of computer.
- External factors are controllable.

- We will determine efficiency of algorithm in terms of computational complexity

- Computational Complexity- computation + complexity
- Computation involves the problems to be solve and algorithms to solve them.
- Complexity involves study of factors to determine how much resource is necessary for this algorithm to **run efficiency**.
- Resource- time to run algorithm, memory needed (time complexity is more important

Factors of Performance of Algorithm

- When analyzing the time complexity of an algorithm we may have three cases
- Worst case – The case when the program consumes maximum resources (time and memory) to process the given data.
- Best case – The case when the program consumes minimum resources (time and memory) to process the given data.
- Average case – The case when the program consumes average resources (time and memory) to process the given data.

- Program efficiency is inversely proportional to clock time", it means that a more efficient program will take less time than a less efficient program to process the given data. Or the more efficient a program is, the less time it will take to process the given data.

Big O Notation

It allows us to measure the time and space complexity of code.

Big O Notation

1. Rule (LOOPS)

```
for  i in range (n):
        m=m+2
```

All the steps in loop take constant time c and loop is executed n times

Total time = c*n = cn -> O(n)

Big O Notation

2 Rule (Nested LOOPS)

```
for  i in range (n):
        for j in range(n):
            k=k+1
```

All the steps in blue will take cn time and outer loop executed n times

Total time = cn*n = cn^2 -> $O(n^2)$

Big O Notation

3 Rule (Consective Statements)

```
x=x+1                          #constant time =a
for I in range(n):             #constant time=cn
        m=m+2
for  i in range (n):           #constant time=bn^2
        for j in range(n):
                k=k+1
```

Total time = $a+cn+bn^2 = O(n^2)$ (considering only the dominant term)

Big O Notation

4 Rule (if else statements)

```
x=x+1                                   #constant time =a
If len(x)!=len(y):                      #constant time=b
        return false                    #constant time=e
for  i in range (n):                    #constant time=(c+d)*n
        if(x[i] !=y[i]                  #constant time=c
                return false            #constant time=d
```

Total time = a+b+e+(c+d)*n= O(n)

Big O Notation

4 Rule (if else statements)

```
x=x+1                                   #constant time =a
If len(x)!=len(y):                      #constant time=b
        return false
for  i in range (n):                    #constant time=(c+d)*n
        if(x[i] !=y[i]                  #constant time=c
                return false            #constant time=d
```

Total time = a+b+(c+d)*n= O(n)

Worst case complexity /run time complexity /run time efficiency

If the efficiency of the algorithm doit can be expressed as $O(n) = n^2$

```
D=1
while d<=n:
        e=1
        while(e<n):
                doit(….
                e=e+1
        d=d+1
```

$n*(n-1)*n^2 = O(n^4)$

Worst case complexity /run time complexity /run time efficiency

Find Worst case complexity?

```
for i in range(n):
        a=i + (i+1)
        print(a)
for i in range(m):
        b=i + (i+1)
        print(b)
```

$n*(a+b)+m*(c+d) = O(n+m)$