

Time Complexity

1. Print 1D array on console

```
void print1DArray(int arr[], int n){  
    for(int i = 0 ; i < n ; i++)  
        sysout(arr[i]);  
}
```

No of iterations = n
time \propto No. of iterations

Time $\propto n$

Time Complexity = $T(n) = O(n)$

2. Print 2D Array on console

```
void print2DArray(int arr[][], int m, int n){  
    for(int i = 0 ; i < m ; i++){  
        for(int j = 0 ; j < n ; j++){  
            sysout(arr[i][j]);  
        }  
    }  
}
```

No of iterations = m
of outer loop

No. of iterations = n
of inner loop

Total iterations = $m * n$

Time $\propto m * n$

Time Complexity = $T(m, n) = O(m * n)$

\therefore if $m = n$, Time Complexity = $T(n) = O(n^2)$

Time Complexity

3. add two numbers

```
int addition(int num1, int num2){  
    return num1 + num2;  
}
```

Time requirement of this algorithm is not dependent on values of num1 & num2. Means, this algorithm is going to take constant amount of time.

$$\text{Time complexity} = T(n) = O(1)$$

4. print table of given number

```
void printTable(int num){  
    for(int i = 1 ; i <= 10 ; i++)  
        sysout(i * num);  
}
```

This loop is going to iterate 10 times for any value of num.

constant time requirement

$$\text{Time complexity} = T(n) = O(1)$$

Time Complexity

5. Print Binary of given Decimal

| | |
|---|---|
| 2 | 9 |
| | 4 |
| | 2 |
| | 1 |

1
0
0
1

```
void printBinary(int n){  
    while(n > 0){  
        sysout(n % 2);  
        n = n / 2;  
    }  
}
```

$$(9)_{10} = (1001)_2$$

| n | n > 0 | rem |
|---|-------|-----|
| 9 | T | 1 |
| 4 | T | 0 |
| 2 | T | 0 |
| 1 | T | 1 |
| 0 | F | |

$$n = 9, 4, 2, 1, 0$$

$$n = n, n/2, n/4, n/8, \dots$$

$$= \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \dots, \frac{n}{2^{\text{itr}}}$$

for $n=1$, last time condition will be true

$$\frac{n}{2^{\text{itr}}} = 1$$

$$n = 2^{\text{itr}}$$

$$\text{itr} \log 2 = \log n$$

$$\text{itr} = \frac{\log n}{\log 2}$$

$$\text{Time} \propto \text{itr}$$

$$\text{Time} \propto \frac{\log n}{\log 2}$$

$$T(n) = O(\log n)$$

Time Complexity

Time Complexities :

$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$,

Modification : '+' or '-' : in terms of n

Modification : '*' or '/' : in terms of $\log n$

$\text{for}(i=0; i < n; i++) \longrightarrow O(n)$

$\text{for}(i=n; i > 0; i--) \longrightarrow O(n)$

$\text{for}(i=0; i < 20; i++) \longrightarrow O(1)$

$\text{for}(i=n; i > 0; i /= 2) \longrightarrow O(\log n)$

$\text{for}(i=1; i < n; i *= 2) \longrightarrow O(\log n)$

$\text{for}(i=0; i < n; i++) \longrightarrow O(n^2)$
 $\text{for}(j=0; j < n; j++)$

$\text{for}(i=0; i < n; i++); \longrightarrow O(n)$
 $\text{for}(j=0; j < n; j++);$

Space Complexity

Total = Input space

(space required
to store actual
input)

+ Auxillary space

(space required to store
variables which are needed to
process actual input)

Find sum of array elements

```
int sumofArray(int arr[], int size){  
    int sum = 0;  
    for(int i = 0 ; i < size ; i++)  
        sum += arr[i];  
    return sum;  
}
```

Input Space = Input variable = arr = n

Auxillary space = Processing variables = size, sum = 3

Total Space = Input space + Auxillary space
= n + 3

space \propto total space
space \propto n + 3

$\therefore n \gg \gg$

Space complexity = $S(n) = O(n)$

Auxillary
space = $O(1)$
complex

Linear Search

for searching and sorting algorithms

time is directly proportional to number of comparisons

1. Best case - if key is found in initial locations - $O(1)$
2. Average case - if key is found at middle of array - $O(n)$
3. Worst case - if key is found at last locations
- if key is not found - $O(n)$

Binary Search

1. Best case - if key is found at few initial levels - $O(1)$
2. Average case - if key is found at middle levels - $O(\log n)$
3. Worst case - if key is found at last few levels
- if key is not found - $O(\log n)$

Algorithm Solving Approches

Iterative

loops are used

```
int factorial(int num){  
    int fact = 1;  
    for(int i = 1 ; i <= num ; i++)  
        fact *= i;  
    return fact;  
}
```

Time is proportional to number of iterations

$$T(n) = O(n)$$

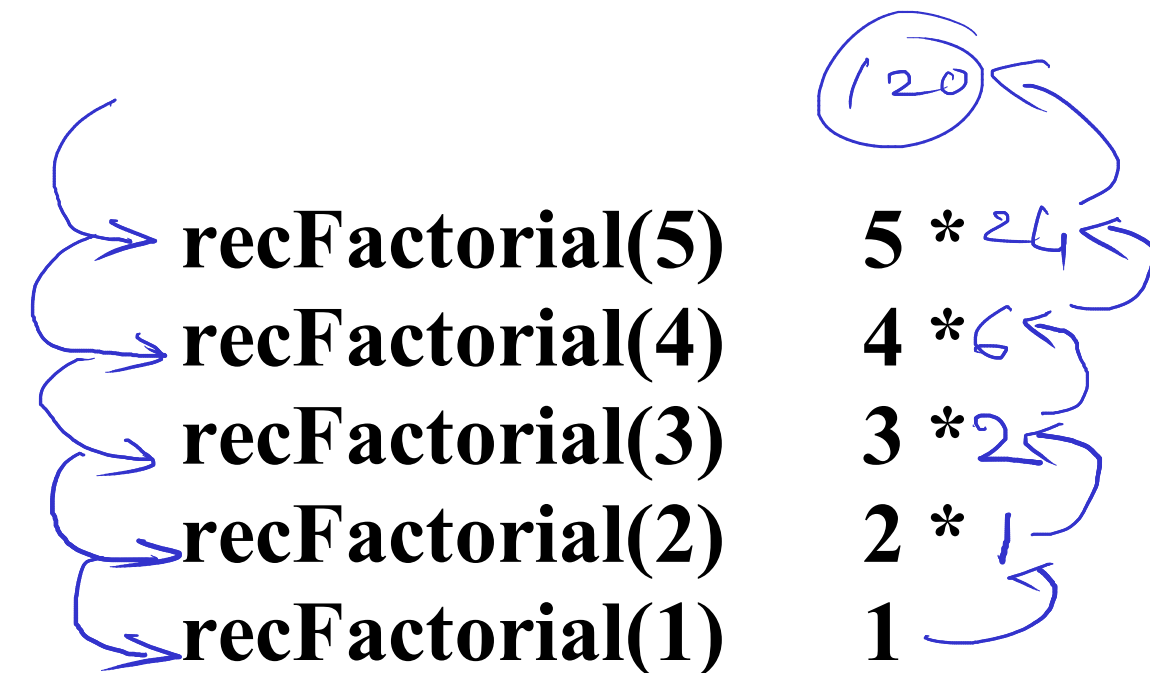
Recursive

Recursion is used

Formula : $n! = n * (n-1)!$

Terminating condition : $1!=1$

```
int recFactorial(int num){  
    if(num == 1)  
        return 1;  
    return num * recFactorial(num - 1);  
}
```



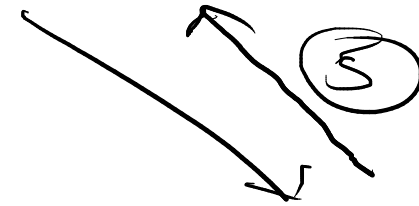
Time is proportional to number of recursive function calls

$$T(n) = O(n)$$

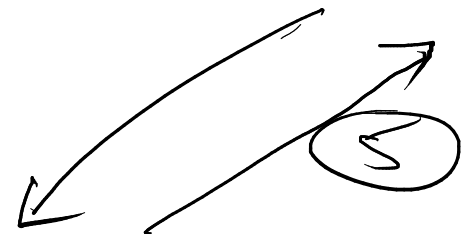
Binary Search Recursive

| | | | | | | | | | |
|----------|----|----|----|----|----------|----------|----------|----|----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| arr | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |
| key = 66 | | | | | <u>1</u> | <u>3</u> | <u>2</u> | | |

↑ 5
BS(arr, 66, 0, 8) m = 4



BS(arr, 66, 5, 8) m = 6



BS(arr, 66, 5, 5) m = 5

Selection Sort

$$\text{No. of comparisons} = 1 + 2 + 3 + \dots + n$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2 + n}{2}$$

$$\text{Time} \propto \frac{n^2 + n}{2}$$

$$\text{Time} \propto n^2 + n$$

$$\text{Time} \propto n^2$$

Best case
Average case
Worst case

$$\rightarrow T(n) = O(n^2)$$

In case of mathematical polynomial only consider term which has degree in power because it is highest growing term

| n | n^2 |
|-----|-------|
| 1 | 1 |
| 10 | 100 |
| 100 | 10000 |

Degree of polynomial —
highest power of variable