Approach

Iterative

We use loops

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

Recursive

Recursive function

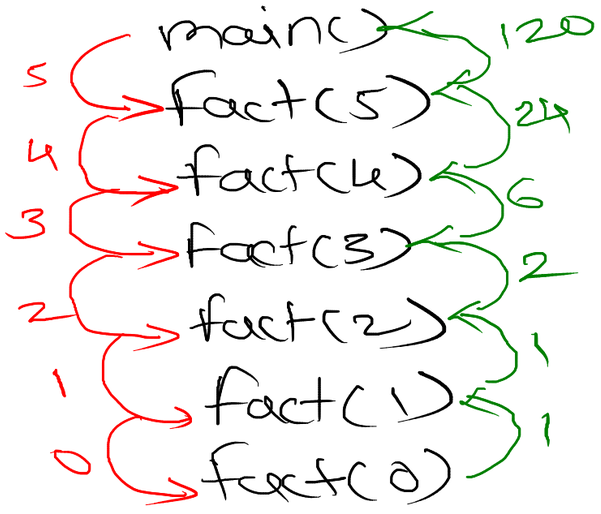
n! = n * (n - 1)!

n = 0, then stop

```
int fact(int n){
        if(n == 0)
                return 1;
        return n * fact(n - 1);
}
```

```
int fact(int n){
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}

int main(void){
    int num = 5;
    int f = fact(num);
    sysout(f);
    return 0;
}
```



For every function call, Function Activation Record is created stack.

FAR
1. Return Addr
2. Formal Argument
3. local variable

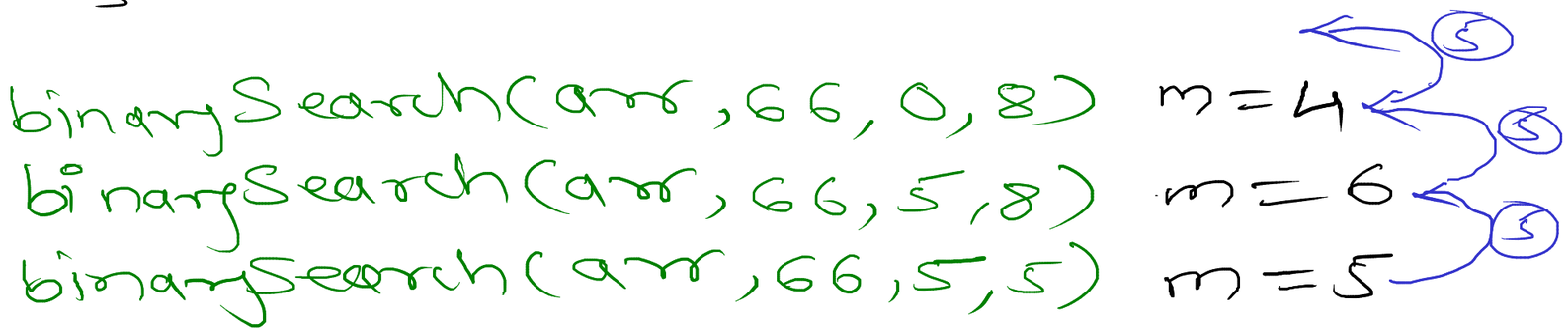
```
int binarySearch(int arr[], int key, int left, int right) {
    0> stop if partition is invalid
        if(left > right)
            return -1;

    1> find middle element of array
        mid = (left+right)/2

    2> Compare middle element with key
        if(key == arr[mid])
            return mid;   — if key is matching

    3> if key < middle element
        -search key into left partition
    return binarySearch(arr, key, left, mid-1);

    4> if key > middle element
        -search key into right partion.
    return binarySearch(arr, key, mid+1, right);
}
```



Algorithm Analysis

- find out the number of resources required to execute the algorithm
- resources are time and space

Exact Analysis

- finding out the exact time and space required to execute
- time (nS, uS, mS) - time is dependent on processor type, number of processes running at that time
- space (Bytes, Kb, Mb, Gb) - space is dependent on datatypes, processor type [Space required to store FAR vary from arch to arch]

Approximate Analysis

- finding out the approximate time and space required to execute
- Asymtotic analysis - mathematical way of finding out time and space required
- Big O / O() notation is used to indicate complexities(time and space)

Time Analysis

- unit time required to execute the algorithm
- time is equal to number of iterations of the loop

1. factorial of given number

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

itr = n

Time ∝ itr

Time ∝ n

$T(n) = O(n)$

2. Print 2D array on console

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

n n

itr = n

itr = n

Total itr = $n * n = n^2$

Time ∝ itr

Time ∝ n^2

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

3. Add two numbers

```
int sum(int num1 , int num2){
        int res = num1 + num2;
        return res;
}
```

Time requirement of this algorithm is constant

$T(n) = O(1)$

4. Print table of given number

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

itr = 10

Time ∝ itr

Time ∝ 10

$T(n) = O(1)$

5. Print binary of given decimal

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

9	-	9 % 2	-	1
	-	9/2=4		
4	-	4 % 2	-	0
	-	4/2=2		
2	-	2 % 2	-	0
	-	2/2=1		
1	-	1 % 2	-	1
=	-	1/2=0		

$n = 9, 4, 2, 1$

$n = 9, 9/2, 4/2, 2/2$

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

$= \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3} \dots \frac{n}{2^{k.(itr)}}$

for last value 1, condition of loop will be true

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

$n = 2^{itr}$

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

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

Time ∝ $\frac{\log n}{\log 2}$

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