Recursion

- calling same function/method within itself
- to use recursion, we must know two things/conditions
    1. define formula/process in terms if itself
    2. terminating condition

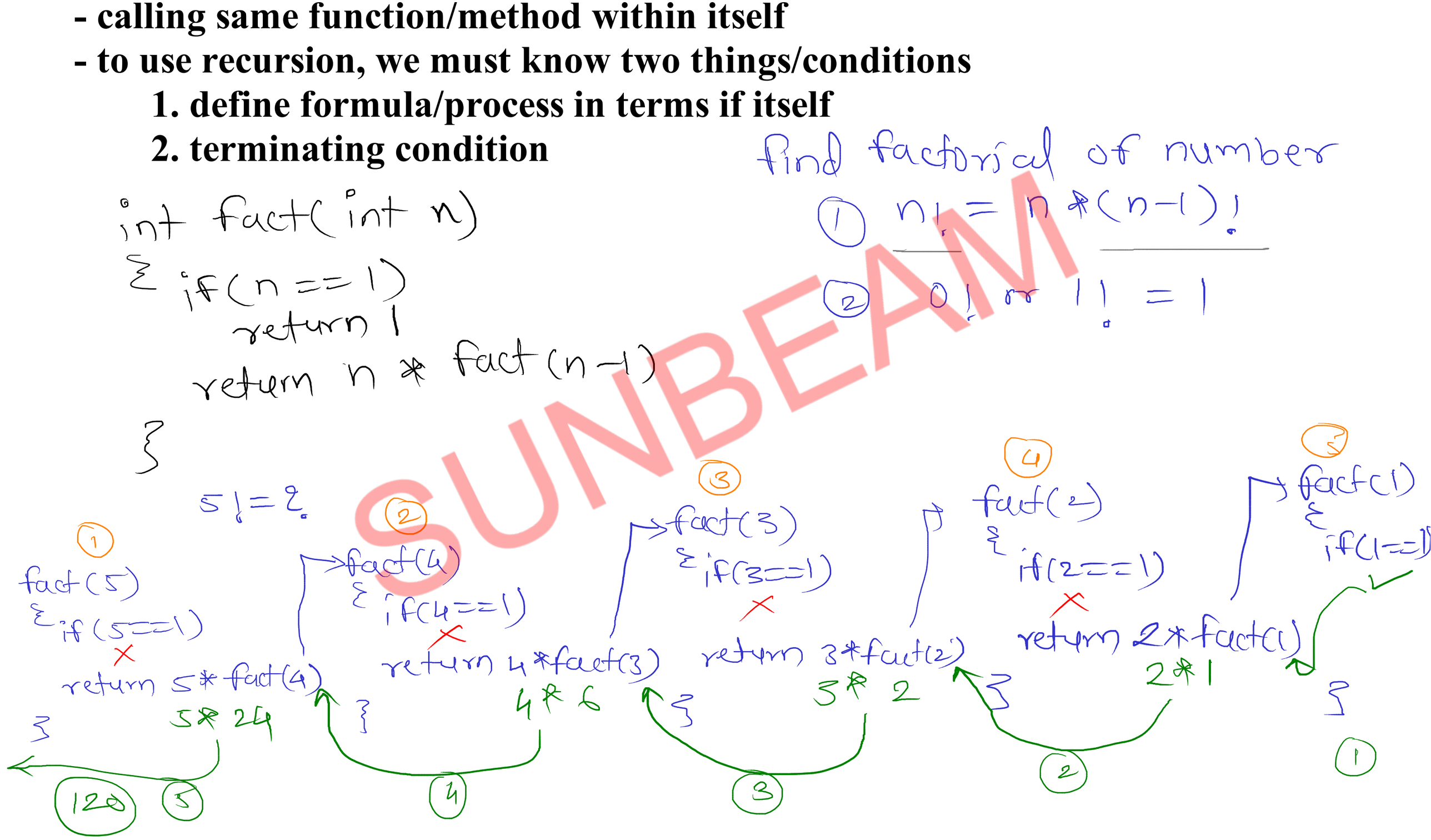
Find factorial of number

① $n! = n * (n-1)!$

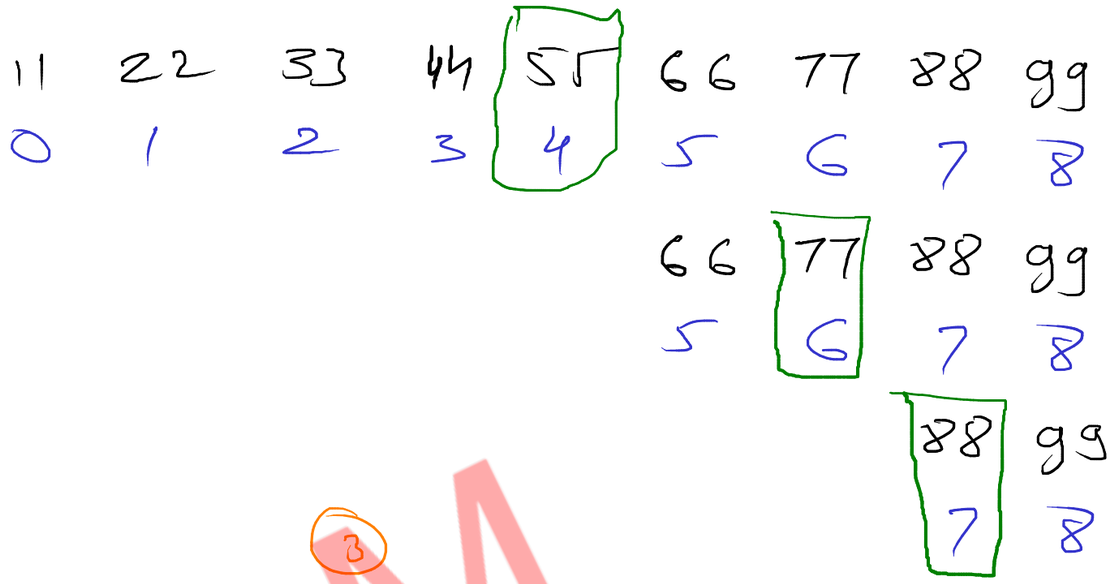
② $0! \text{ or } 1! = 1$

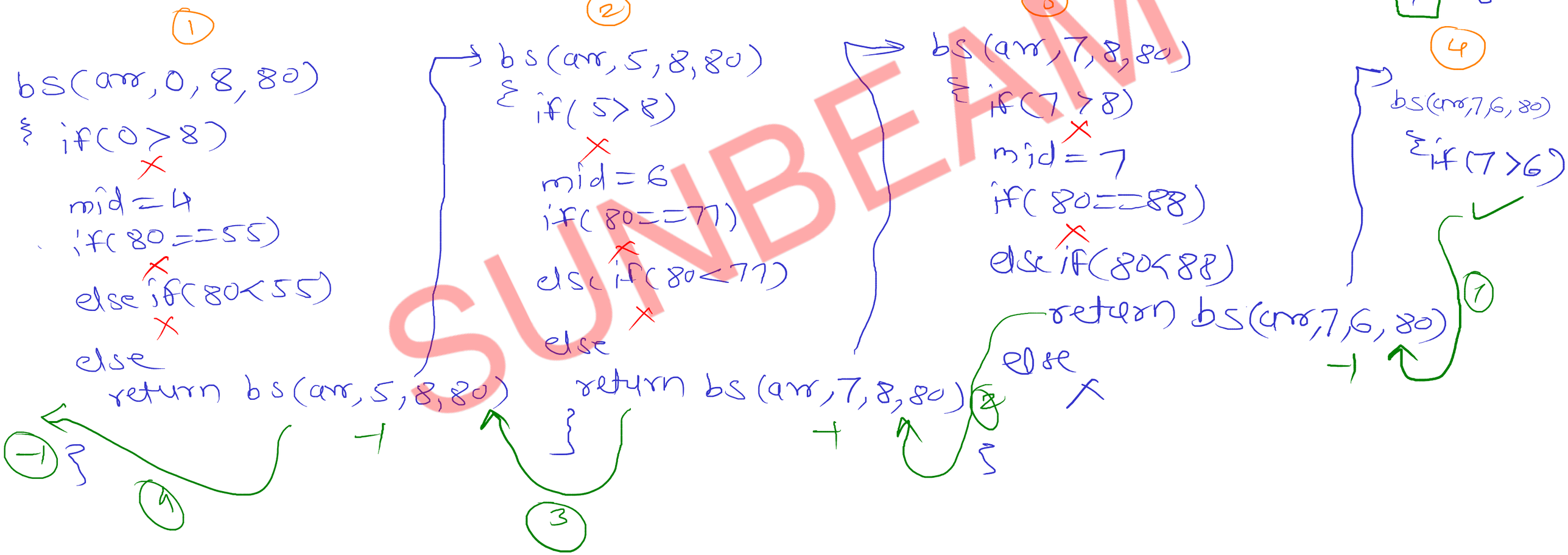
```
int fact(int n)
{
    if(n == 1)
        return 1
    return n * fact(n-1)
}
```

5! = ?



```
public static int binarySearch(int arr[], int left, int right, int key) {
        if(left > right)
            return -1;
        int mid = (left + right)/2;
        if(key == arr[mid])
            return mid;
        if(key < arr[mid])
            return binarySearch(arr, left, mid-1, key);
        else
            return binarySearch(arr, mid+1, right, key);
    }
```





Algorithm analysis / Efficiency measurement / Complexities

- finding time and space requirement of an algorithm
    1. Time - time required to execute the algorithm (ns, us, ms, s)
    2. Space - space required to excute the algorithm inside memory (bytes, kb, mb,..

1. Exact analysis

- finding exact space and time of the algorithm
- it depends on some exeternal factors
- time is dependent on type of machine(cpu), no of processes running at that time
- space is dependent on type of machine(architecture), data types

2. Approximate analysis

- finding approximate time and space of the algorithm
- mathematical approach is used to find time and space complexity of the algorithm and it is known as "Asymptotic analysis"
- it also tells about behavior of the algorithm when input is changed or sequence of input is changed
- behaviour of algorithm can be observed into three cases
    1. Best case
    2. Average case
    3. Worst case

to denote time and space complexity we use Big-O notation

Time Complexity

- count the number of iterations for the loop which is used inside the algorithm
- timp required is directly proportional to the iterations of the loop

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 ∝ iterations

Time ∝ n

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

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]);
        }
}
```

iterations of outer loop = m

iterations of inner loop = n

Total iterations = m * n

Time ∝ iterations

Time ∝ m * n

$\boxed{T(m,n) = O(m*n)}$

∴ m == n

Time ∝ n * n

$\boxed{T(n) = O(n^2)}$

3. add two numbers

```
int sum(int n1, int n2){
        return n1 + n2;
}
```

- time requirement of this algorithm is not dependent on values of n1 & n2
- mean it will take constant/fixed amount of time.
- Constant time requirement can be denoted as

$$T(n) = O(1)$$

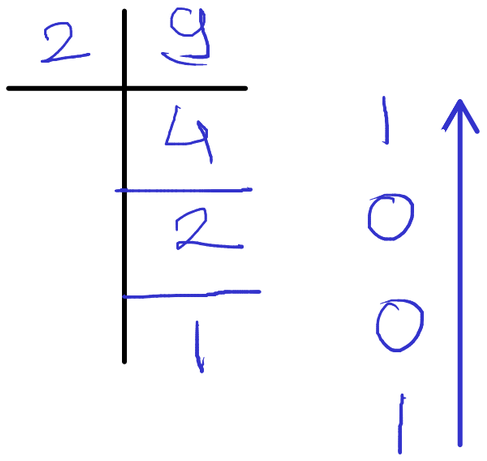
4. print table of given number

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

- this loop will execute fixed (10) no. of times.
- Constant time requirement

$$T(n) = O(1)$$

5. print binary of decimal number




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

n	n > 0	n % 2
9	T	1
4	T	0
4	T	0
2	T	0
1	T	1
0	F	

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

$n = 9, 4, 2, 1$

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

$= n/2^0, n/2^1, n/2^2, \ldots \; n/2^i$ — no. of iterations

∴ for $n=1$ last time loop will be executed

$n/2^i = 1$

$n = 2^i$

$\log 2^i = \log n$

$i \log 2 = \log n$

$i = \dfrac{\log n}{\log 2}$

Time $\propto$ iterations

Time $\propto \dfrac{\log n}{\log 2}$

Time Complexity ↓

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

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), ... O(2^n), ...

modification : '+' or '-' --> time complexity will be in terms of n
modification : '*' or '/' --> time complexity will be in terms of log n

```
for(i=0; i<n; i++)        → O(n)
for(i=n; i>0; i--)        → O(n)
for(i=0; i<n; i+=20)      → O(n)
for(i=n; i>0; i/=2)       → O(log n)
for(i=1; i<n; i*=2)       → O(log n)
for(i=1; i<=20; i++)      → O(1)
```

n=9 → 9,4,2,1 } 4 iters
n=9 → 1,2,4,8 }

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)        } → O(n^2)
```

```
for(i=0; i<n; i++);   } n
for(j=0; j<n; j++);   } n      += 2n
```

Time ∝ 2n
T(n) = O(n)

```
for(i=0; i<n; i++)
    for(j=1; j<n; j*=2)       } n
                              } log n      *= n log n
```

T(n) = O(n log n)

Space Complexity

- finding approximate space required to execute an algorithm

Total space = Input space + Auxillary space

(space of actual input/data)

(space required to process actual data/input)

search key into array (linear search)

```
int linearSearch(int arr[], int n, int key){
    for(int i = 0 ; i < n ; i++)
        if(key == arr[i])
            return i;
    return -1;
}
```

input variable = arr

processing variable = n, key, i

input space = n

Auxillary space = 3

Total space = n + 3

∴ n >>> 3

space ∝ n

$S(n) = O(n)$

Auxillary space Analysis

processing variable = n, key, i

Auxillary space = 3

$AS(n) = O(1)$

Searching Algorithms Analysis

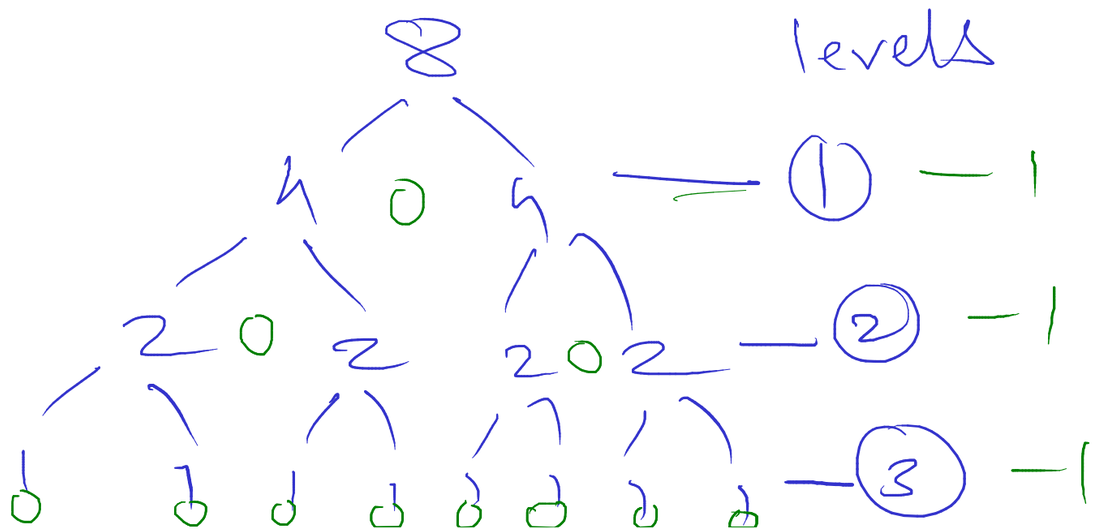
- for searching and sorting algorithms, we count number of comparisions
- time is directly proportinal to number of comparision

Linear Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(n)
Worst case :	key is found in last few comparitions	:	O(n)
	key is not found		

Binary Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(log n)
Worst case :	key is found in last few comparitions	:	O(log n)
	key is not found		



$$n = 2^{l}$$

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

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

Algorithm Implementation Approches

Any algorithm can be implemented using two approches

1. Iterative approach
- loops are used

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

Time ∝ no. of iterations

$T(n) = O(n)$

2. Recursive approach
- recursion is used

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

Time ∝ no. of recursive calls

$T(n) = O(n)$