Data Structures and Algorithms

Data structure

Organising data into memory for efficient processing along with few operations like add, delete, search etc on orgnised data

1. Abstraction

- Abstract Data Type

2. Resuability

- can be reused as per our need.
- can be reused to implement another data structure
- can be reused to implement few algorithms

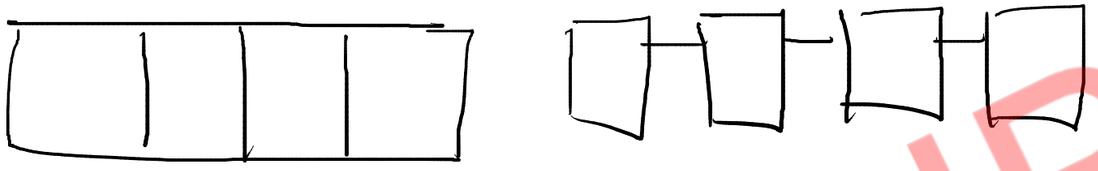
3. Efficiency

- efficiency can be measured into two terms:
  1) Time - required to execute
  2) Space - required to execute inside memory.

Types of Data structrure

Linear Data structure

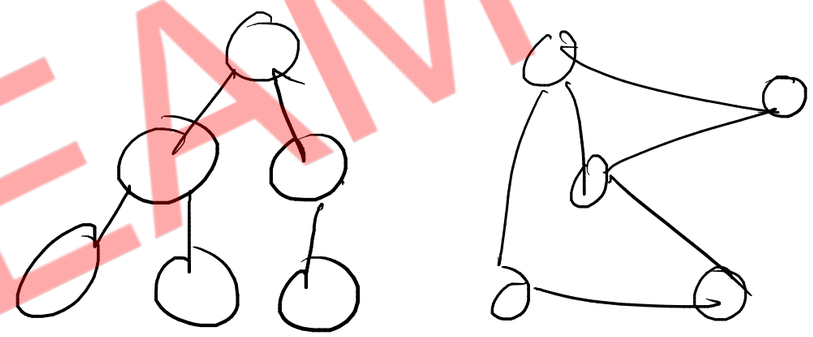
- data is orgnised sequentially (one after another)



- data can be accessed linearly (sequentially)
- Basic data structures
  1) Array
  2) structure/class
  3) stack
  4) Queue
  5) Linked List

Non Linear Data structure

- data is orgnised level by level (heirarchy)



- data can not be accessed sequentially
- Advanced data structures
  1) Tree
  2) Graph
  3) Heap

- Hash Table

Algorithm

Program – set of instructions to the machine(CPU)
Algorithm – set of instructions to the human
↓
programer/developer

- set of instructions to solve given problem statement
- step by step solution of given problem statement

eg. Find sum of array elements
step 1: create sum variable & sum = 0
step 2: traverse array from 0 to N-1 index
step 3: add each element of array into sum
step 4: print/return sum variable

- programming language independents

Templates → implementation
↓ ↓
Algorithms Program

Searching Algorithms

- finding data/key into collection(multiple) of data
- searching can be performed by using two ways
    1. Linear search (works on random data)
    2. Binary search (works on organised data - sorted)

Linear search

Problem statement - find key into collection of multiple data

```
//1. decide key/data to be searched
//2. traverse collection (array) from one end to another end
//3. compare key with each element of an array
//4. if key is matching, then return True/index of array
//5. if key is not matching, then return False/-1
```

Binary search

```
//1. divide collection(array) into two parts (find middle element)
//2. compare middle element with key
//3. if key is matching, then return true/index
//4. if key is less than middle element, search it into left partition
//5. if key is greater than middle element, search it into right partition
//6. repeat step 1 to 5 untill key is found
//7. if key is not matching, then  return false/-1
```

11	22	33	44	55	66	77	88	99
0	1	2	3	4	5	6	7	8

return 5

(3) binarySearch(arr, 0, 8, 66) (1)

mid = 4

return 5

(2) binarySearch(arr, 5, 8, 66) (2)

mid = 6

return 5

(1) binarySearch(arr, 5, 5, 66) (3)

mid = 5

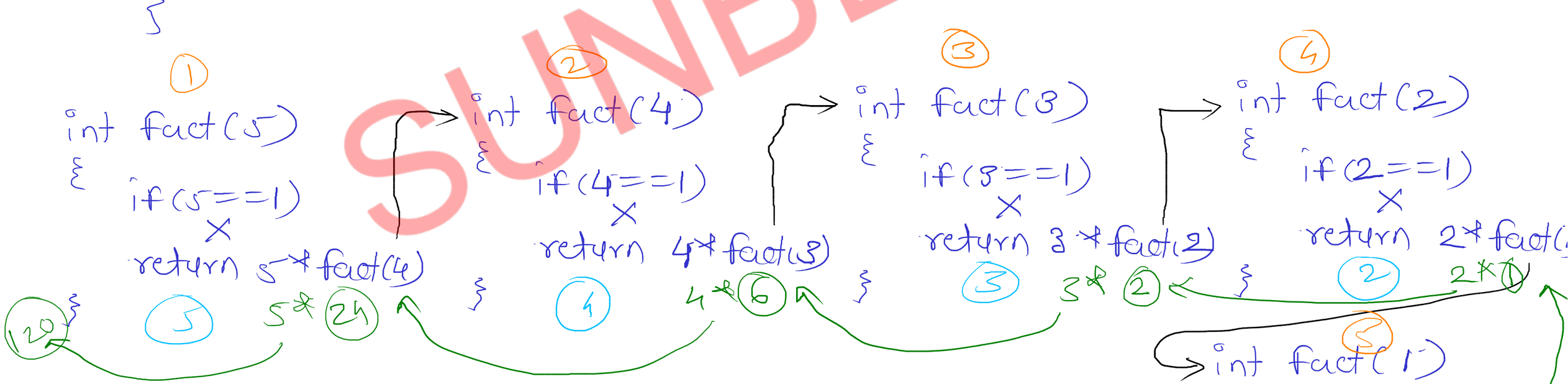
Recursion

- function calling itself
- we can use recursion if
    1. we know the process/formula in term of itself
    2. we know the terminating condition

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

$1! = 1$

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



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;
}
```

2. Recursive approach
- recursion is used

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