Algorithm Analysis

1. Time

- time is directly proportional to number of iterations of loop

① `for(int i=0 ; i<n ; i++)` → i = 0, 1, ... n-1

no. of itr = n

Time ∝ n

T(n) = O(n)

② `res = num1 + num2` → constant time

T(n) = O(1)

③ `for(int i=1 ; i<10 ; i++)` → fix 10 iterations

T(n) = O(1)

④
```
for(int i=0 ; i<n ; i++){ → n
    for(int j=0 ; j<n ; j++){ → n
    }
}
```

total itrs = n*n = n^2

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

(5) for(int i=n; i>0; i=i/2)

$$i = n, n/2, n/4, n/8 \text{ ---- }$$

$$= n/2^0, n/2^1, n/2^2, n/2^3 \text{ ------ } n/2^{itr}$$

$i = 1$ —> last time condition will be true

$$n/2^{itr} = 1$$

$$n = 2^{itr}$$

$$itr \log 2 = \log n$$

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

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

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

11 22 33 44

11 22 33 44

11 22 33 44

$n = 4$

comparisions $= n-1$

Time & compr

$T(n) = O(n)$

2. Space

- amount space needed to execute an algorithm inside memory

Total space = Input space + Auxillary space

↑ space of actual data

↑ space required to process actual data

e.g. sum of array elements

arr[n] — input space → n

sum, i, size — Auxillary space → 3

n + 3

S(n) = O(n)

Input space complexity → O(n)

Auxillary space complexity → O(1)

Iterative Approach

loops

Recursive Approach

recursion

Find factorial of number

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

- No of iterations

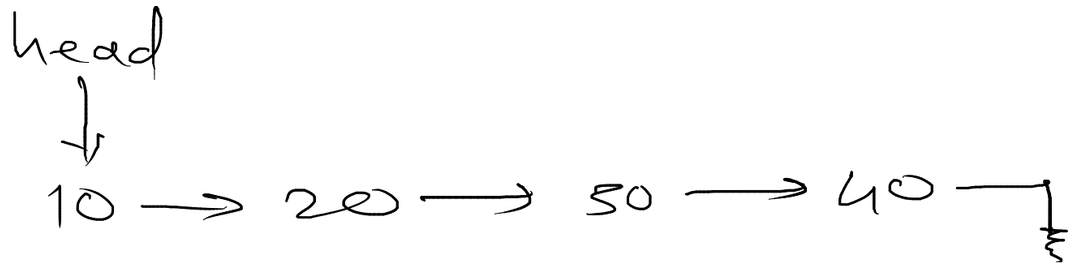
T(n) = O(n)

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

recFactorial(5)
recFactorial(4)
recFactorial(3)
recFactorial(2)
recFactorial(1)

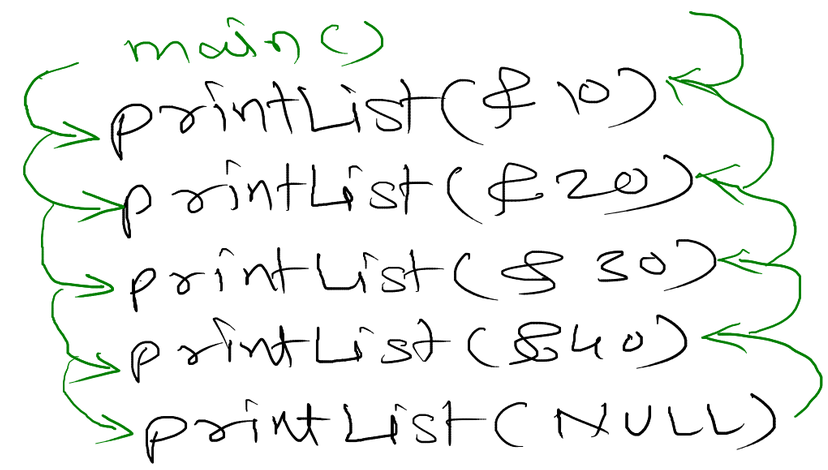
- No of function calls

T(n) = O(n)



Tail Recursion

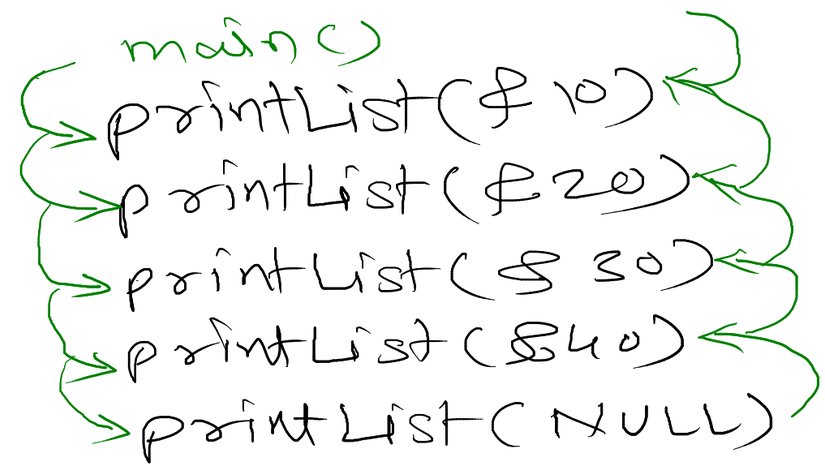
```
void fprintList(node_t *trav)
{
        if(trav == NULL)
                return;
        printf(trav->data);
        fprintList(trav->next);
}
```



List : 10, 20, 30, 40

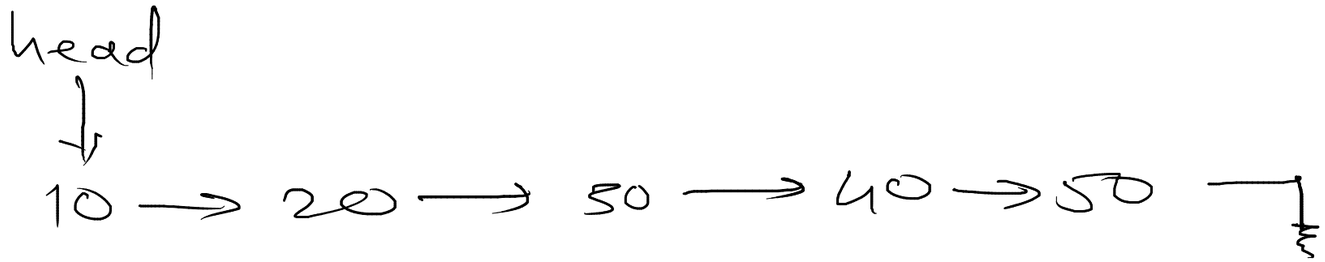
Non tail recursion

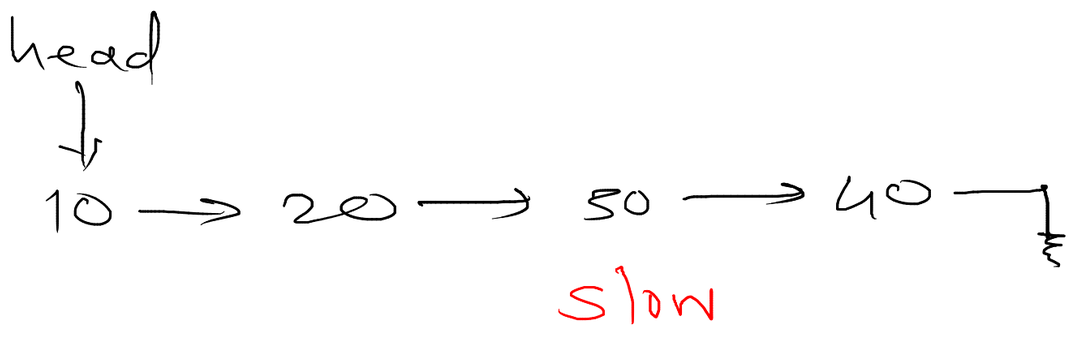
```
void rprintList(node_t *trav)
{
        if(trav == NULL)
                return;
        rprintList(trav->next);
        printf(trav->data);
}
```



List : 40, 30, 20, 10

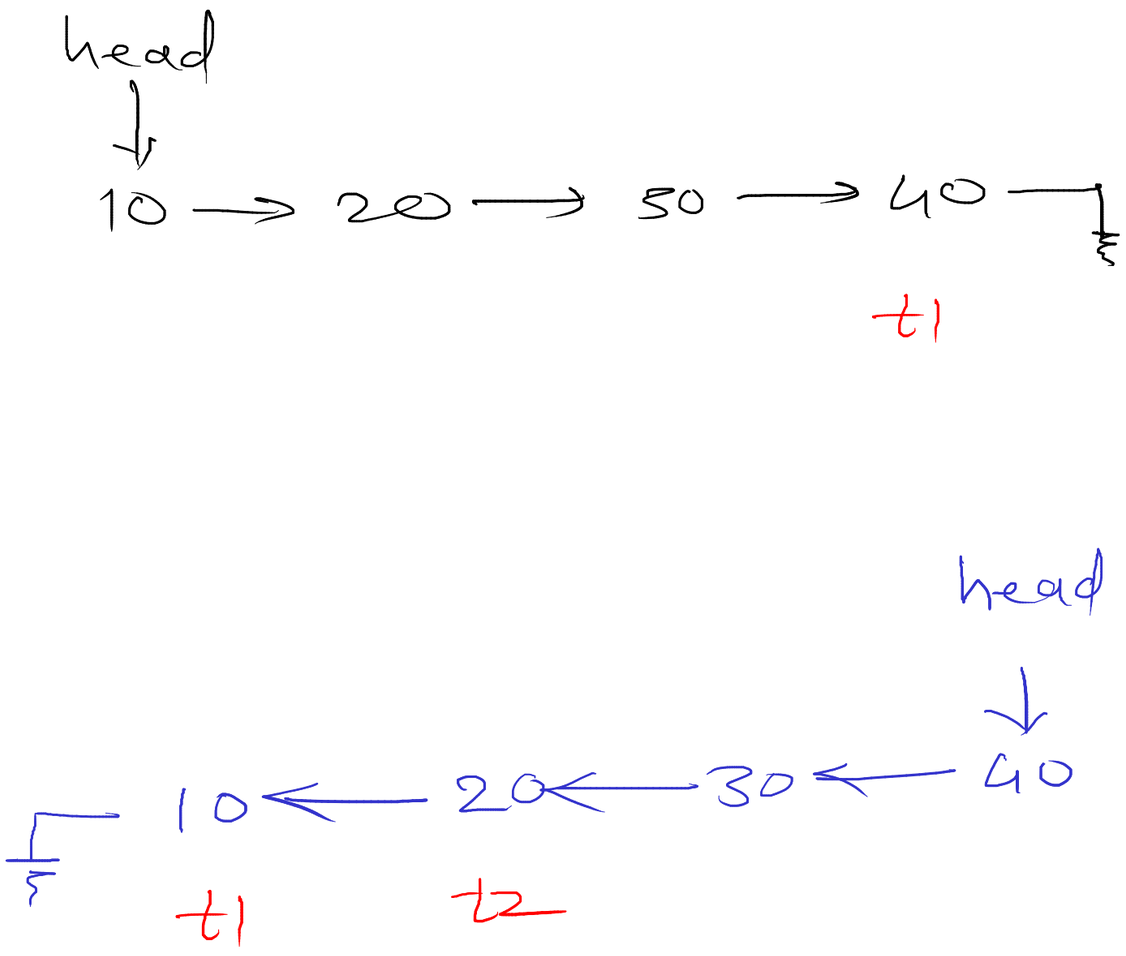
Find Mid of Linked List






```
fast = head;
slow = head;
while( fast != NULL && fast -> next != NULL)
{
        slow = slow -> next;
        fast = fast -> next -> next);
}
```

Reverse Linked List



$T(n) = O(n)$

```
t1 = head;
t2 = head->next;
while (t2 != NULL)
{
    t3 = t2->next;
    t2->next = t1;
    t1 = t2;
    t2 = t3;
}
head->next = NULL;
head = t1;
```

Linked List Applications

- Linked list is dynamic data structure
- Due to this nature, linked list is used to implement other data structures
    - stack
    - queue
    - graph (Adjacency list)
    - Hash Table (Seperate chaining)

- OS Data structures - Job queue, Ready queue, Waiting queue

(Doubly Circular Linked List)

Stack
(LIFO)

1. Add First
   Delete First

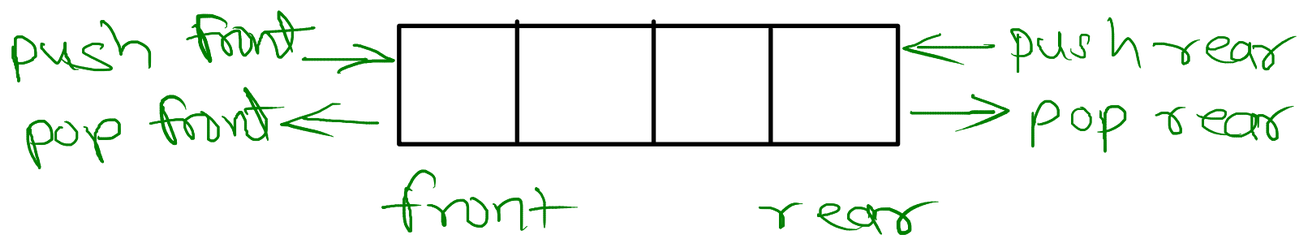
2. Add Last
   Delete Last

Queue
(FIFO)

1. Add First
   Delete Last

2. Add Last
   Delete First

Deque
(Double Ended Queue)



1. Input Restricted Deque
   input - allowed from only one end
   output - allowed from both ends

2. Output Restricted Deque
   input - allowed from both ends
   output - allowed from only one end

```
//1. create node with given data
//2. if tree is empty
        //a. add newnode into root itself
//3. if tree is not empty
        //3.1 create trav pointer and start at root
        //3.2 compare data with current node data
                //3.2.1 if data is less than current node data
                        //3.2.1.1 if left of current node is empty
                                // add newnode into left of current node
                        //3.2.1.2 if left of current node is not empty
                                // go into left of current node
                //3.2.2 if data is greater than current node data
                        //3.2.2.1 if right of current node is empty
                                // add newnode into right of current node
                        //3.2.2.2 if right of current node is not empty
                                // go into right of current node
//3.3 repeat step 3.2 untill node is added
```