Recursion

Function calls itself to solve smaller version of the same problem.

Two parts of Recursion

(i) Base Case – Stop Condition (when to stop calling itself)

(ii) Recursive Case – part where function call itself

Real Life Example

(i) Queue of people (ii) Comment threads

(iii) Organisational Hierarchies

Common Mistake

1. Missing Base Case – stack overflow
2. Not simplyfying the input – never reaches base case
3. Too deep recursion – Large input
4. Keeping in mind the time complexity

When to use Recursion

1. Problem can be broken into sub problems.
2. Trees & Graphs
3. Backtracking, DP, Divide & Conquer.

Recursion is a technique where a function calls itself to solve a problem by breaking down into smaller sub-problems.

Base-Condition : Every function call in recursion is stored in the call stack. If the recursion is too deep or has no base condition, the call stack keeps growing untill memory is exhausted, causing a stack overflow error. A base condition is essential in recursion. It stops the recursion when a certain condition is met. Without it, recursion goes infinite and causes a stack overflow.

```
if (num === 0) return;
```

Approach

problem : Print numbers from n to 1 using recursion

Recurse with num – 1

Stop when num === 0

Time Complexity : O(n) Space Complexity : O(n)

```
function printDescending(num){
    if(num===0) return;
    console.log(num);
    printDescending(num-1);
}
printDescending(5);
```

problem : Calculate sum of first n natural number using recursion.

```
function sum(n){
    if(n===0) return 0;
    return n+sum(n-1);
}
console.log(sum(5));
```

T(c) = O(n) S(c) = O(n)

Write a function sum(n) that calculates the sum of all numbers in an array using recursion. It sums from index 0 to n

```
let arr=[5,2,6,1,3]
function sum(n){
    if(n==0) return arr[0]
    return arr[n] + sum(n-1);
}
console.log(sum(arr.length-1));
```

Q Sum of add numbers in array using Recursion.

```
let arr = [5,2,6,1,3]
function sum(n){
    let isOdd = arr[n]%2 !== 0;
    if (n===0) return isOdd ? arr[0] : 0;
    return (isOdd ? arr[n] : 0) + sum(n-1);
}
console.log(sum(arr.length-1));
```

Function to calculate factorial of a number using Recursion

```
function fact(n){
    if (n===1) return 1;
    return n+(n-1)fact
}
console.log(fact(5));
```

Q Write a recursive function isPowerOfTwo(n) that returns true if n is a power of 2, otherwise false.

```
function isPowerOfTwo(n){
if(n===1) return true;
else if( n<1 || n%2 !== 0) return false;
}
console.log(isPowerOfTwo(8)); // true
Console.log(isPowerOfTwo(9)); // false
```

Q fibbonaci number

```
var fib = function (n) {
    if (n<=1)
    return n;
return fib(n-1) + fib(n-2);
};
```

Linear Search :- Linear search is a simple search algorithm used to find a specific element in an array. It checks each element of the array one by one until the target value is found or the end of array is reached.

Approach

1. Start from the first element of the arr.
2. Compare the current element with the target value.
3. If a match is found, return the index.
4. If the loop ends without finding the target return -1.

Time Complexity (TC)

- In the worst case, the algorithm traverses the entire arr
- Each element is checked exactly once.

TC = O(n), where n is the size of the array.

Space Complexity

The algorithm does not use any extra space.

SC = O(1) (Const Space)

Code

```
let arr = [4, 5, 1, 3, 9]
function linearSearch (arr, target){
        for (let i=0; i<arr.length; i++){
        if (arr[i] === target){
            return i;
        }
    }
    return -1
}
let result = linearSearch (arr, 5);
consde.log ("Element found at index", result);
```

Binary Search is an efficient algorithm used to find the position of a target value within a sorted array. Unlike linear Search, it repeatedly divides the search interval in half, significantly reducing the number of comparisons.

Approach

- Set left = 0, right = nums.length - 1;
- while left <= right:
- Calculate middle = Math.floor((left + right)/2)
- if nums[middle] === target, return middle.
- if target < nums[middle], discard the right half : right = middle - 1;

Else discard the left half : left = middle + 1

If the target is not found, return -1.

Code

```
var search = function (nums, target){
    let left = 0;
    let right = nums.length - 1;
  while (right >= left){
    let middle = Math.floor((left + right)/2)
  if (target === nums[middle]){
     return middle;
  } else if (target < nums[middle]){
      right = middle - 1;
   } else{
        left = middle + 1
    }
  }
  return -1;
};
```

Time complexity

Best Case : O(1)

Worst case : O(log n)

Space Complexity

- O(1)

Bubble Short

Bubble short is a simple shorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the array is sorted. After each pass, the largest unsorted element 'bubbles up' to its correct position at the end of the array. It's called Bubble short because smaller elements slowly "bubble" to the top of the list.

Approach

1. Iterate through the array multiple times.
2. In each pass, compare adjacent elements.
3. If the current element is greater than the next one, swap them.
4. After each pass, the largest unsorted element bubbles up to its correct position at the end
5. Use a boolean flag (isSwapped) to track whether any swapping happened.
6. If no swaps occured in a full pass, the array is already sorted → early exit (optimization)
7. Repeat this process for n-1 passes (where n is the array length), or until no swaps are needed.

Best case : O(n) when arr is sorted

TC - $O(n^2)$ in all cases Worst case array in reverse order

Space Comp - O(1)

Code

```
let arr = [4,5,1,3,9]
function bubbleSort (arr){
    let n = arr.length;
for (let i=0; i<n-1; i++) {
   let isSwapped = false;
for (let j=0; j<n-1-i; j++){
    if (arr[j] > arr[j+1]){
      let temp = arr[j]
      arr[j] = arr[j+1];
      arr[j+1] = temp;
      isSwapped = true;
    }
}
  if (!isSwapped)
    break;
}
  return arr;
}
let result = bubbleSort (arr)
console.log ("Sorted Array:", result)
```

Selection Sort

Selection sort is a simple comparision-based sorting algorithm. It divides the array into two parts: a sorted subarray and an unsorted subarray. Initially, the sorted part is empty and the unsorted part is the entire array. In each iteration it finds the minimum element from the unsorted part and moves is to the end of the sorted part.

Approach

1. Iterate over the array from index 0 to n-2.
2. for each index i, assume the element at i is the minimum in the unsorted part
3. Run an inner loop, ~~swap the element~~ from j = i+1 to n-1 to find the actual min^m element
4. After the inner loop, swap the element at i with the element at min (if they're not the same).
5. Repeat untill the array is sorted.

Time Complexity : $O(n^2)$ in all cases
Roughly $n*(n-1)/2$ comparisons are always performed.
Space Complexity : $O(1)$
Selection Sort is an in-place sorting algorithm, so it doesn't require extra space.

```
let arr = [4,5,1,3,9];
function selectionSort (arr){
    let n = arr.length
  for (let i=0; i<n-1; i++){
      let min = i;
  for(let j=i+1; j<n; j++){
    if (arr[j] < arr[min]) {
        min = j;
    }
  }
  if (min != i){
    let temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
  }
}
return arr;
}
```

```
let result = selection sort (arr);
console.log (result);
```

Insertion Sort

Insertion Short is a simple and intuitive sorting algorithm that buids the final sorted array one element at a time. It works by taking each element from the input and inserting it into its correct position in the already sorted part of the array. Starting from the second element, it compares the current element with the previous ones, shifting larger elements one position ahead to make space for the current element. This process continues until all elements are sorted Insertion Sort is efficient for small or nearly sorted datasets and operates in-place without requiring extra memory.

Approach

1) Start from the second element (idx 1)
   Since the first element is trivially "sorted".
2) Store the current element (curr) and compare it with all previous elements.
3) Shift the previous elements one position forward if they are greater than the curr element
4) Insert the current element (curr) at it correct sorted posth
5) Repeat until the whole arr is sorted.

Time Complexity

Best Case (already sorted) : O(n)
Average case : $O(n^2)$
Worst Case (reverse sorted) : $O(n^2)$

Why?
Best case : No shifting happens.
Worst case : Every element has to be compared and shifted back to the start.

Space Complexity :-
O(1) → No extra array is used : sorting is done in place.

```
let arr = [4,5,1,3,9]
function insertionSort(arr){
    let n = arr.length;
for( let i =1; i<n; i++){
    let curr = arr[i];
    let prev = i-1;
    while(prev>=0 && arr[prev]>curr){
        arr[prev+1] = arr[prev];
        prev--;
    }
    arr[prev+1] = curr;
}
return arr;
}
let result = insertionSort(arr);
console.log("Sorted array", result);
```

```
for(i=1;i<n;i++){
curr=a[i]
prev=i-1;
while(a[prev]>curr){
    a[p+1]=a[p];
    p--
}
a[p+1]=a[curr];
}
```

Merge Sort

Merge Sort is a popular divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts them, and then merges the sorted halves into on sorted result.

it is an example of a stable sort that guarantees O(nlogn) performance in all cases best, worst and average.

Approach : Divide & Conquer

Divide : Split the arr into two halves.

Conquer : Recursively sort each half using merge sort.

Combine : Merge the two sorted halves into one sorted array.

Key Concept : Merge Step

The key step is merging two sorted arrays efficiently into one sorted array. This is done using a two-pointer approach, comparing elements from both arrays and adding the smaller one into a new result array.

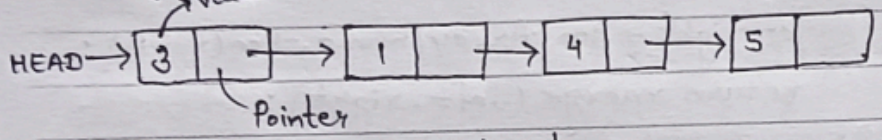
Time Complexity : O(nlogn) — Divide takes log n steps and merging takes linear time.

Space Complexity : O(n) — Additional space is needed to store the merged arrays.

```
Var sortArray = function (nums) {
    if (nums.length <= 1) return nums;
    let mid = Math.floor (nums.length/2);
    let left = sortArray (nums.slice (0, mid));
    let right = sortArray (nums.slice (mid));
    return merge (left, right);
};
function merge (left, right){
    let res = [], i = 0, j = 0;
while (i < left.length && j < right.length) {
    if (left[i] < right[j]) {
        res.push (left[i++]);
    } else {
        res.push (right[j++]);
    }
}
return [...res, ...left.slice(i), ...right.slice(j)];
}
```

Linked list

A linked list is a linear data structure in which elements (called nodes) are connected using pointers. Each node contains:



→ Linear data structure
→ nodes are linked together by a reference field.

- A value (the actual data)
- A reference (or pointer) to the next node (and optionally to the previous node in doubly linked list)

Types of Linked List

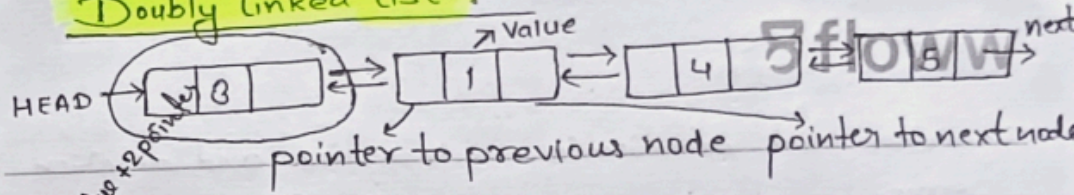
Singly linked list :-

Value : the data part
next : a pointer / reference to the next node.
It moves only in one direction (forward)

Structure : [value | next] -> [value | next]
-> [value | null]

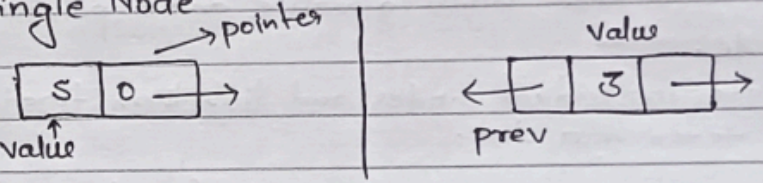
- each node pointer to the next node.

Doubly linked list :



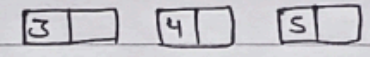
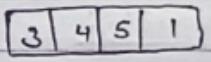
Value : the data
next ~~prev~~ : pointer to the next node
prev : pointer to the previous node
Allows bidirectional traversal
Structure : null <- [prev | value | next] <-> [prev | value | next] -> null

★ Single Node



HEAD : Starting Node of the LL.

How LL will differ from arr.

Linked List	~~Array~~
	
· Linear	· linear
· non-contigeous	· contigeous
· Dynamic size (changes easily)	· fixed size (can be dynamic)
· Node = (value + pointer)	· Just value
· getting and fetching ~~easy~~ is hard. O(n)	getting and fetching is easy O(1)

Insertion and deletion is easy.	Insertion and deletion is easy.
Extra Memory	Memory efficient

8. Design and implement a simple singly linked list using basic building blocks like nodes and references (or pointers). The goal is to:

- Define a node with a value and a pointer to next node.
- Create individual nodes and link them together to form a list.
- Traverse the list from the head and print each node's value.

Example: Input: Create 3 nodes with values 10, 20, 30 and link them in order.

Output: Print values in sequence: 10 20 30

Approach:

1. Define a Node class or structure with a val and next pointer.
2. Create individual nodes
3. Links the nodes: node1.next → node2,
4. Set the head of the list of the first node
5. Traverse the list using a loop until the current node becomes null (or None).
6. Print value of each node

Explanation

- Each node store data and a reference to the next node.
- The list is connected by linking nodes via the next field.
- The traversal starts from the foundation for the head and moves node by node until the end.
- The basic structure forms the foundations for many advanced data structures like stacks, queues and graph adjacency lists.

8floww

```
function Node(val) {
    this.val = val
    this.next = null;
}
function MyLinkedList() {
    this.head = null;
    this.size = 0;
}
const list = new MyLinkedList();

const node1 = new Node(10);
const node2 = new Node(20);
const node3 = new Node(30);

node1.next = node2;
node2.next = node3;

list.head = node1;
list.size = 3;

let current = list.head;
while (current){
    console.log(current.val)
    current = current.next;
}
```

Design LinkedList

Operations

get (index) : Return the value of the nodes at the specified index (0-indexed). If index is invalid, return -1.

addAtHead (val) : Insert a node at the beginning of the list.

addAtTail (val) : Append a node at the end of the list.

addAtIndex (index, val) : Insert a node before the index-th node in the list.

deleteAtIndex (index) : Delete the node at the given index

Approach

1. Use a custom Node class that holds val and next attributes.
2. maintain a head pointer to the first node and a size to track the list length.
3. Each operation ensures index bounds are checked.
4. Traverse the list up to the required index using a loop.
5. All operations are implemented with time complexity O(n) in the worst case

```
function Node(val){
    this.val = val;
    this.next = null;
}
var MyLinkedlist = function(){
    this.head = Null;
    this.size = 0;
};
MyLinkedList.prototype.get = function(index){
    if(index<0 || index>=this.size) return -1;
    let curr = this.head;
    for(let i=0; i<index; i++) curr = curr.next;
    return curr.val;
}
MyLinkedlist.prototype.addAtTail = function(val){
    const newNode = new Node(val);
    if(!this.head) this.head = newNode;
    else{
        let curr = this.head
        while(curr.next) curr = curr.next;
        curr.next = newNode;
    }
    this.size++;
};

MyLinkedList.prototype.addAtHead = function(val){
    const newNode = new Node(val);
    newNode.next = this.head;
    this.head = newNode;
    this.size++;
};
```

```
MyLinkedlist.prototype.addAtIndex = function (index,val){
    if(index<0 || index>this.size) return;
    if(index===0) return this.addAtHead(val);
    const newNode = new Node(val);
    let curr = this.head;
for(let i=0;i<index-1;i++) curr = curr.next;
    new Node = curr.next;
    curr.next = newNode;
    this.size++;
};

MyLinkedlist.prototype.deleteAtIndex = function(index){
    if(index<0 || index>=this.size) return
if(index===0) this.head = this.head.next;
else{
    let curr = this.head
for(let i=0; i<index-1; i++) curr= curr.next;
    curr.next = curr.next.next;
    }
this.size--;
}
```

15 Middle of Linkedlist

Approach

→ Use two pointer: slow and fast
→ Initialize both at the head of the list.
→ Move slow one step and fast two steps at a time.
→ when fast reaches the end, slow will be at the middle.

Time complexity
O(n)

Space Complexity
O(1)

```
var middleNode = function(head){
    let slow = head;
    let fast = head;
    while(fast && fast.next){
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
};
```

⇒ Reverse Linked List

Approach

→ Use three pointers prev, curr, and temp
In each step
i) point current node's next to previous
Move prev and curr forward.
Return prev as the new head

Time and Space Complexity
O(n), O(1)

```
var reverselist = function(head){
    let prev = null
    let curr = head
    while(curr){
        let temp = curr.next
        curr.next = prev;
        prev = curr;
        curr = temp;
    }
    return prev;
};
```

Linked-List Cycle using Hash Table

Approach

→ Use a Set to track visited nodes
→ while traversing, if we encounter a node already in set, we have found a cycle.
→ if we reach null, there's no cycle.

Time and Space Complexity

O(n) , O(n)

Code

```
var hasCycle = function (head){
    let seenNodes = new Set();
    let curr = head;
  while (curr!==null){
      if (seenNodes.has(curr)) {
          return true;
    }
    seenNodes.add(curr);
      curr = curr.next;
    }
  return false;
};
```

→ Using Floyd's Cycle detection

- Use floyd's cycle Detection (also called the Tortoise and Hare algorithm)
- Use two pointers; slow moves one step, fast moves two steps.
- If there is cycle, they will eventually meet. if fast.next becomes null, there's no cycle.

Time and Space Complexity

O(n) , O(1)

```
var hasCycle = function (head){
  if (!head) return false;
  let slow = head;
  let fast = head.next
  while (slow!==fast){
      if (!fast||!fast.next) return false;
      slow = slow.next;
      fast = fast.next.next;
  }
  return true;
};
```

Palindrome Linked list

Approach 1

→ Traverse the linked list and store values in an array.
→ check whether the array is a palindrome by comparing elements from start and end moving towards the center.

Time and Space Complexity

O(n) , O(n)

```
var isPalindrome = function(head) {
    let arr = [];
    let curr = head;
  while (curr!==null) {
      arr.push(curr.val);
      curr = curr.next;
  }
  let left = 0, right = arr.length-1;
  while (left<right){
```

```
    if (arr[left++] !== arr[right--])
        return false;
    }
    return true;
};
```

floww

Approach 2

- Use two pointers (slow and fast) to find the middle of the linkedlist.
- Reverse the second half of the list.
- Compare the first and second halves node by node.

Time and Space Complexity :- O(n), O(1)

Code

```
var isPalindrome = function(head) {
    let slow = head, fast = head;
While (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;
    }
// Reverse second half
 let prev = null;
 While (slow) {
    let temp = slow.next;
    slow.next = prev;
    prev = slow;
    slow = temp;
 }
// compare two halves
let first = head, second = prev
While (second) {
  if (first.val !== second.val) retur false
  first = first.next
   second = second.next;
}
return true;
};
```

Intersection of two Linked lists

floww

Approach

- Store all nodes of headB in a set.
- Traverse headA; return the node when one exists in the set.

If no match is found, return null

Time Complexity and Space Complexity

O(n+m) O(m)

```
var getIntersection = function (headA, headB) {
    let set = new Set();
    While (headB) {
      set.add(headB);
      headB = headB.next
    }
    While (headA) {
      if (set.has(headA)) return headA;
      headA = headA.next;
    }
    return null;
};
```

Linked List Element

Approach

→ Use a dummy (sentinel) node before the head to simplify edge cases.

→ Iterate through the list using a pointer.

→ Skip any node whose value matches val.

→ Return the next of sentinel as the new head.

Time and Space Complexity : O(n) O(1)

Code

```
var removeLinkedList = function(head, val){
    let sentinel = new ListNode(0, head)
    let current = sentinel;
    while(current && current.next){
        if (current.next.val === val){
            current.next = current.next.next;
        } else {
            current = current.next;
        }
    }
    return sentinel.next;
};
```

Remove Nth Node from End of List - two pass

Given the head of linked list, remove the nth node from the end of the list and return its head.

Approach

- Use a sentinel (dummy) node before the head to simplify edge cases
- first, calculate the total length of the list.
- find the previous node of the one to be deleted using the length - n formula.
- Update the links to skip the target node.

Time and Space Complexity: O(n), O(1)

```
var removeNthFromEnd = function(head, n){
    let sentinel = new ListNode();
    Sentinel.next = head;
    let length = 0;
    while(head){
        head = head.next;
        length++;
    }
    let prevPos = length - n;
    let prev = sentinel;
    for(let i = 0; i < prevPos; i++){
        prev = prev.next;
    }
    prev.next = prev.next.next
    return sentinel.next;
};
```

Remove nth Node from end — one Pass

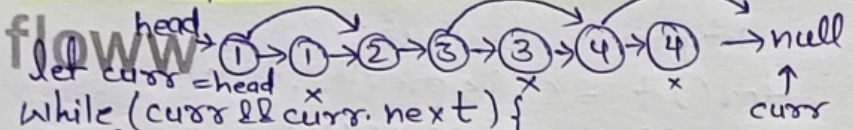
Approach (optimized Two-Pointer)

- Use a dummy node before the head to handle edge cases only.
- Move the first pointer n steps ahead.
- Move both first and second pointers until first reaches the end.
- Now second is just before the node to be deleted. Skip it using second.next = second.next.next

T(c): O(n) S(c): O(1)

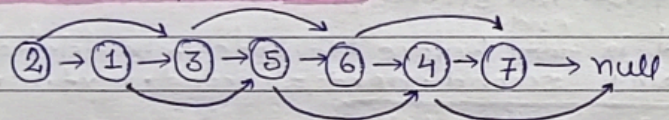
```
var removeNthFromEnd = function(head, n){
    let sentinel = new ListNode(0, head);
    let first = sentinel
    for(let i=0; i<n; i++){
        first = first.next;
    }
    let second = sentinel;
    while(first.next){
        first = first.next;
        second = second.next;
    }
    second.next = second.next.next;
    return sentinel.next;
};
```

Q Remove Duplicates from sorted list




```
let curr = head
while (curr && curr.next) {
    if (curr.val == curr.next.val) {
        curr.next = curr.next.next;
    } else {
        curr = curr.next;
    }
}
return head;
```

Q ODD Even Linkedlist




```
if (!head || !head.next) return head;
odd = head
even = head.next
evenStart = even
while (even.next && odd.next) {
    odd.next = odd.next.next;
    even.next = even.next.next
    odd = odd.next
    even = odd.next
}
odd.next = evenStart;
return head;
```

Add two number

L1 ③ → ④ → ⑨

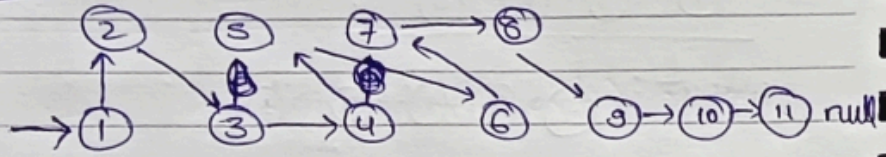
L2 ④ → ⑧ → ① → ⑨

output: ⑦ → ② → ① → ⓪ → ①

```
let ans = new ListNode();
let ansHead = ans;
let carry = 0;
while (l1 || l2 || carry) {
let sum = (!l1 ? 0 : (l1.val) + (!l2 ? 0 : l2.val)
          + carry ;
carry = Math.floor(sum/10)
let digit = sum%10
let newNode = new ListNode(digit);
  ans.next = newNod
  ans = ans.next
  l1 = l1 && l1.next;
  l2 = l2 && l2.next;
}
return ansHead.next
};
```

Merg Two Sorted list

l1



compare next 2 values in l1 & l2
next smaller value.

```
if (!l1) return l2
if (!l2) return l1

let curr = null
if (l1.val < l2.val) {
  curr = l1;
  l1 = l1.next
}
if (l2.val < l1.val) {
  curr = l2
  l2 = l2.next
} else {
  curr = l1
  l1 = l1.next
}

let start = curr

while (l1 && l2) {
  if (l1.val < l2.val) {
    curr.next = l1
    l1 = l1.next
  } else {
    curr.next = l2
    l2 = l2.next
  }
  curr = curr.next;
}
if (!l1) {
  curr.next = l2
}
if (!l2) {
  curr.next = l1
}
return start;
}
```

With dummy node

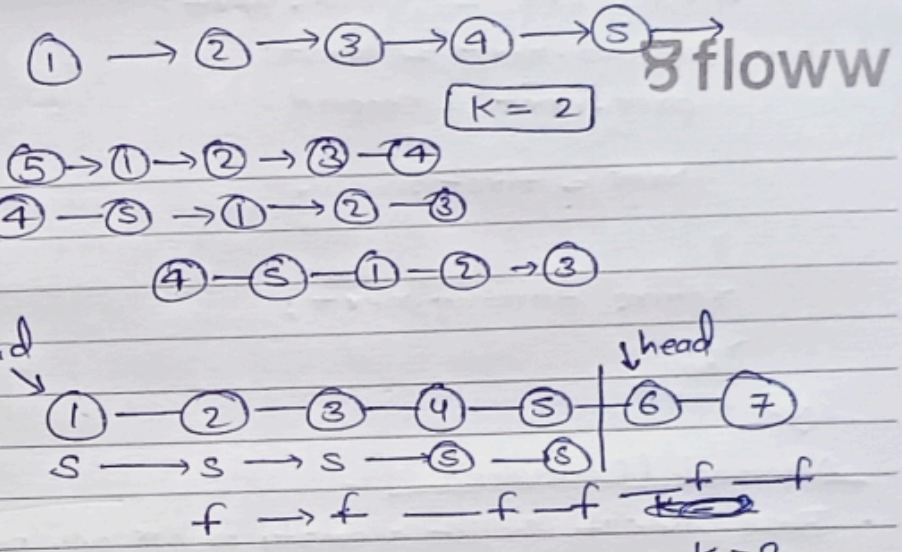
```
let start = new ListNode();
let curr = start

while (l1.val < l2.val) {
  curr.next = l1;
  l1 = l1.next;
}
else {
  curr.next = l2;
  l2 = l2.next;
}
curr = curr.next
}
if (!l1) {
  curr.next = l2
}
if (!l2) {
  curr.next = l1
}
```

return start ;
}

floww

Rotate list

```
if (!head || !head.next) return head;
  // caluclate length
let length = 0;
 let curr = head
while (curr){
      curr = curr.next;
        length++;
  }
  K = K% length
let s = head;
 let f = head;
for(let i=0; i<K; i++){
    f = f.next;
 }
while (first.next){
      s = s.next
      f = f.next
}
 f.next = head
  let newHead = s.next
   s.next = null
   return newHead;
  }
```




```
new Head = s.next
f.next = head
```

reach k from last
mark it as a newhead
link last node to front

fast point start from K = 2

Swaps Nodes in Pairs

```
var swapPairs = function(head){
  if(!head || head.next) return head;

   let dummy = new ListNode(0);
     dummy.next = head;
     let prev = dummy ;
while (prev.next && prev.next.next){
     let first = prev.next;
     let second = first.next;
```

```
        first.next = second.next;
        second.next = first;
        prev.next = second;

        prev = first;

    }
    return dummy.next;
};
```

Approach (Recursive)

- Base case: if the list is empty or has only one node, return the head.
- let l be the first node and r be the second
- Call swapPairs recursively for the rest of the list starting from r.next.
- Set l.next to the result of recursive call.
- Make r point to l and return r as the new head;

T(c) = O(n), where n is the number of nodes in the list
S(c) = O(n), due to recursive call stack.

```
var swapPair = function (head) {
    if (!head || !head.next) return head;
    let l = head;
    let r = head.next;
    l.next = swapPairs(r.next);
    r.next = l;
    return r;
};
```

String

length of word

trim() → remove extraspace from front & back.
Split(" ") → give you arr of all of words

not a write approach

```
var lengthOfLastWord = function(s){
    s = s.trim()
    s = s.split(" ");
    return s[s.length-1].length;
};
```

```
var lengthOfLastWord = function (s) {
    // trim all the spaces at the end
    let n = s.length - 1; // start from last char
    while (n >= 0) {
        if (s[n] === " ") {
            --n;
        } else {
            break;
        }
    }
    // n is the point where my last word starts
    // count the charates till you reach a space
    let count = 0;
    while (n >= 0) {
        if (s[n] != " ") {
```