

# Recursion & Backtracking

- Karun Karthik

## Contents →

- (1) Power of two
- (2) Power of three
- (3) Power of four
- (4) Subsets
- (5) Combination sum
- (6) Rat in a maze
- (7) N - Queens
- (8) Sudoku solver
- (9) Knight's tour problem
- (10) Letter combination of a phone number.
- (11) Subsets II
- (12) Combination sum II
- (13) N - Queens II

DI

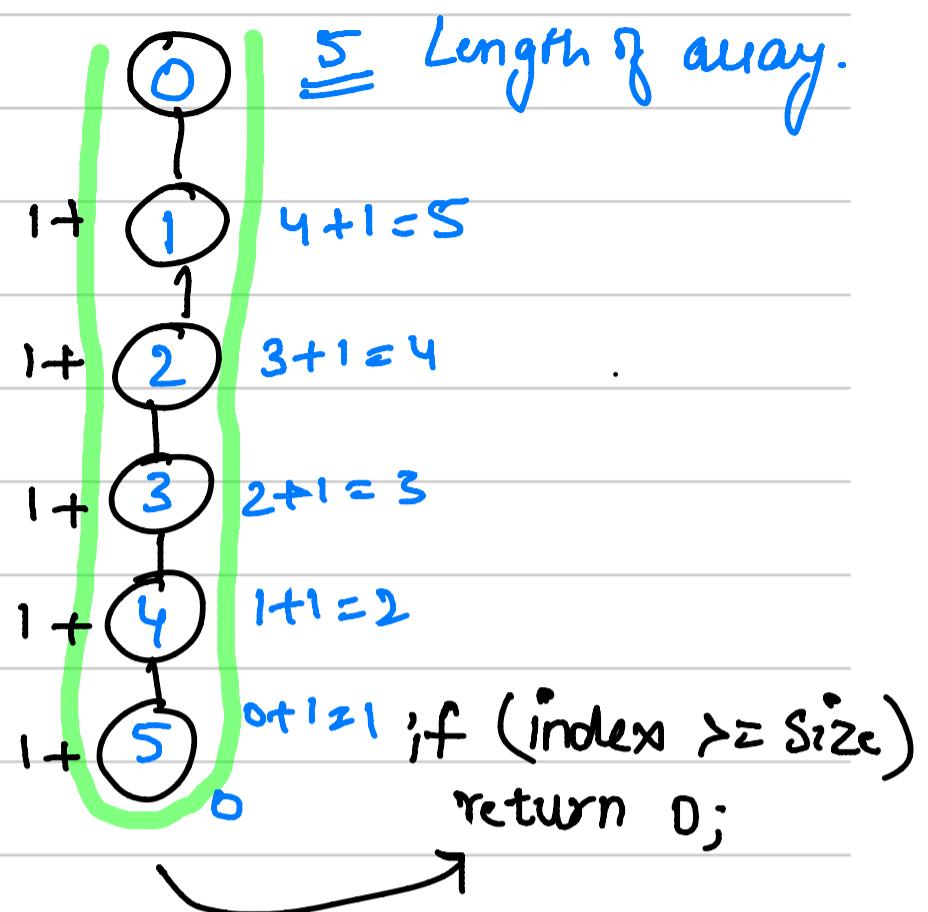
## Recursion

- ① Length of an array

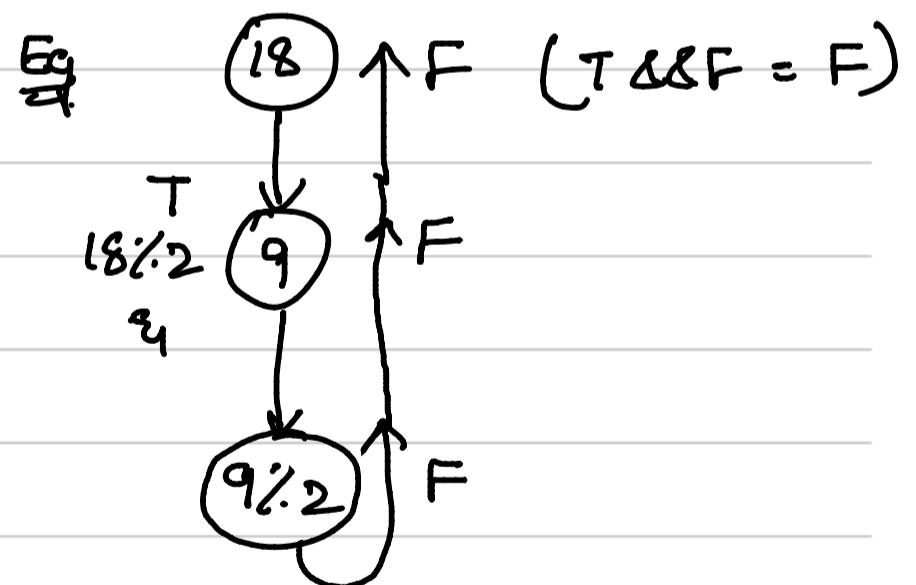
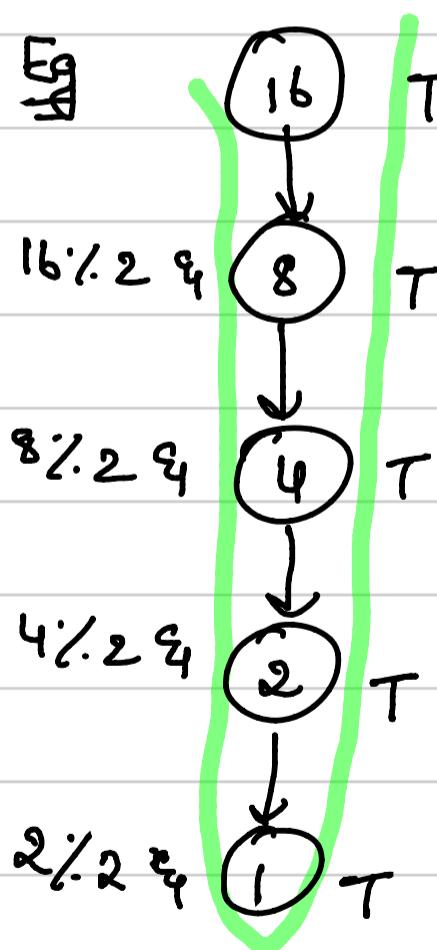
[20, 10, 40, 50, 30]  
0 1 2 3 4

$$TC = O(n)$$

$$SC = O(n).$$



- ① Power of 2  $\rightarrow 2^x = 2^0 \cdot 2^1 \cdot 2^2 \dots 2^n$ .



$$\underline{TC = O(\log_2 n)}$$

if 1 then  
return true.

## Power of 2

```
C++ ▾  
class Solution {  
public:  
    bool isPowerOfTwo(int n) {  
        if(n==1) return true; //need to write it first else it might c  
        if(n<=0 || n%2!=0) return false;  
        return isPowerOfTwo(n/2);  
    }  
};
```

## ② Power of 3

```
C++ ▾  
class Solution {  
public:  
    bool isPowerOfThree(int n) {  
        if(n==1) return true; //need to write it first else it might  
        if(n<=0 || n%3!=0) return false;  
        return isPowerOfThree(n/3);  
    }  
};
```

## ③ Power of 4

```
C++ ▾  
class Solution {  
public:  
    bool isPowerOfFour(int n) {  
        if(n==1) return true; //need to write it first else it might c  
        if(n<=0 || n%4!=0) return false;  
        return isPowerOfFour(n/4);  
    }  
};
```

## D2 Subsets

④ Given an integer array nums, generate all the subsets. (subsequences)

If size =  $n$  then no. of subsets =  $2^n$ .

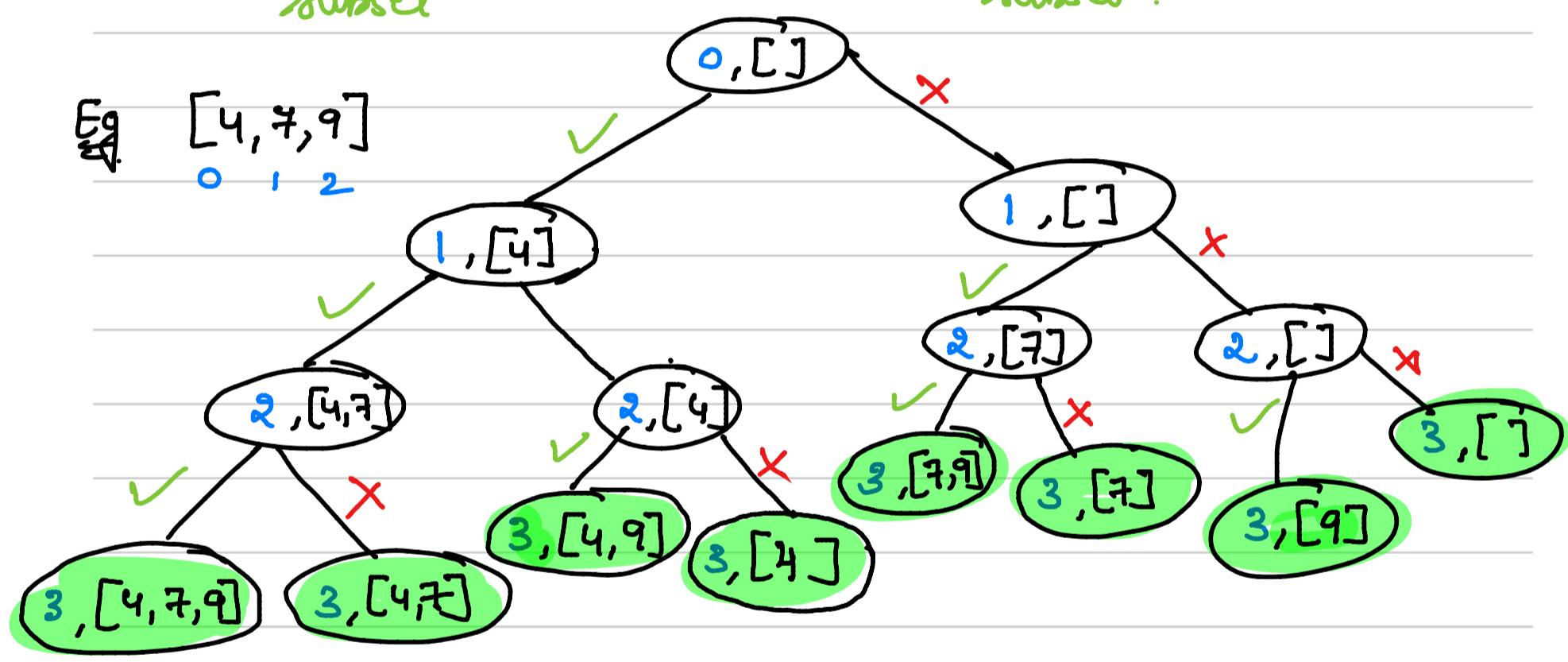
Eg.  $\text{nums} = [1, 2, 3]$

$\text{Set} = [\underline{[ ]}, \underline{[1]}, \underline{[1, 2]}, \underline{[1, 3]}, \underline{[1, 2, 3]}, \underline{[2]}, \underline{[2, 3]}, \underline{[3]}]$

For every, element  
 can be a part of subset      cannot be a part of subset.

Eg.  $[4, 7, 9]$

0 1 2



$\Rightarrow [ [4, 7, 9], [4, 7], [4, 9], [4], [7, 9], [7], [9], [ ] ]$

\* Once index is greater than or equal to size then store in result

$Tc = O(2^n) \rightarrow$  as there are 2 possibilities at every element.

$Sc \approx O(2^n)$

## Code

```
class Solution {
public:
    void generateAllSubsets(vector<int>&nums, int currentIndex, vector<int>&res, vector<vector<int>> &powerSet){
        // base condition
        if(currentIndex >= nums.size()){
            powerSet.push_back(res);
            return;
        }
        int currentVal = nums[currentIndex];
        res.push_back(currentVal);
        generateAllSubsets(nums, currentIndex+1, res,powerSet);

        // remove the currentVal (not considering)
        res.pop_back();
        generateAllSubsets(nums, currentIndex+1, res,powerSet);
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> powerSet;
        vector<int> res;
        generateAllSubsets(nums, 0, res, powerSet);
        return powerSet;
    }
};
```

⑤ Combination sum :-  $\text{nums} = [2, 3, 5]$  target = 8  
 $\underset{0, 1, 2}{}$

Sol.  $[[2, 2, 2, 2], [2, 3, 3], [3, 5]]$

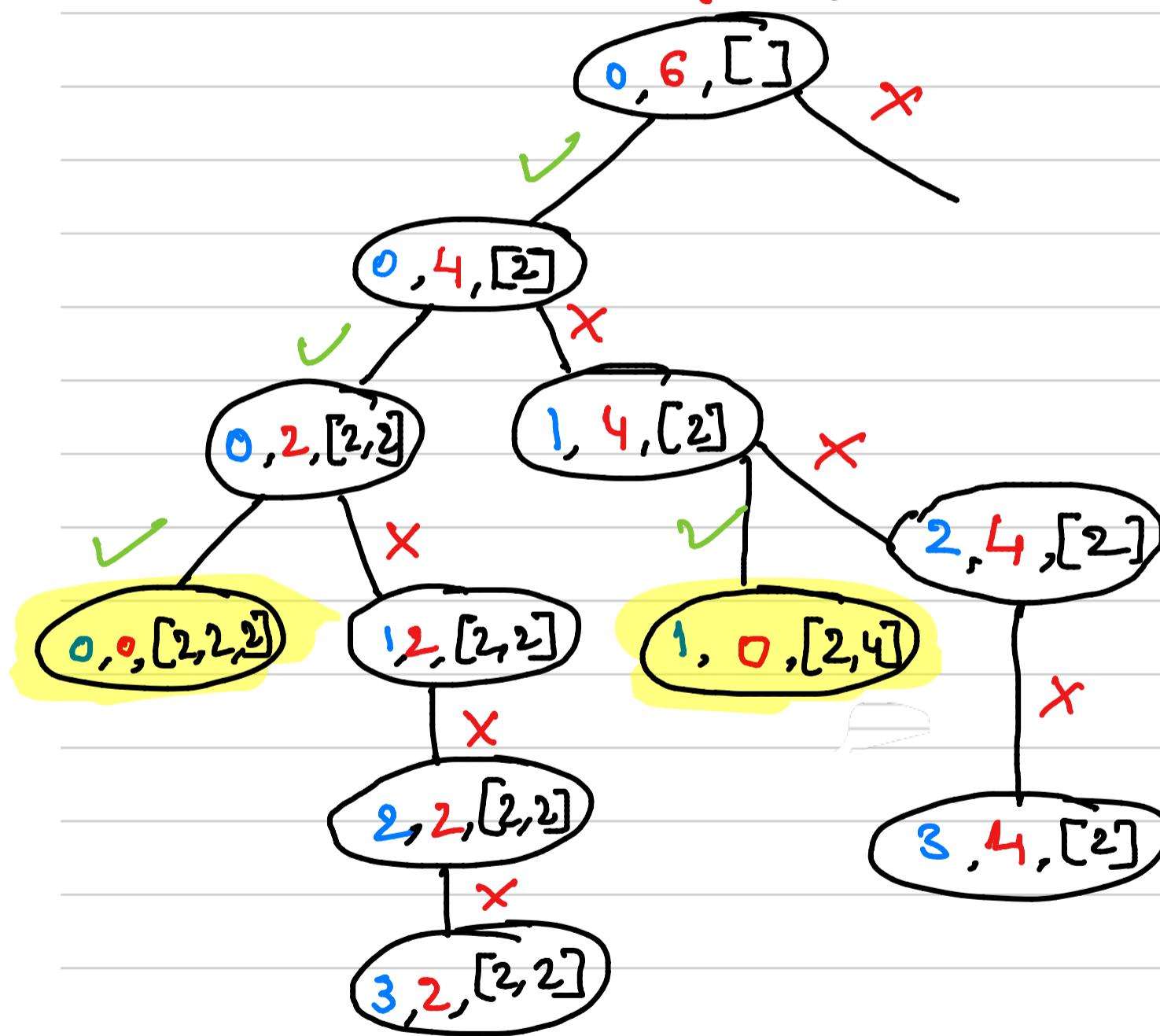
Eg  $[2, 4, 5]$   
 $\underset{0, 1, 2}{}$   
 target = 6

For every index  $\Rightarrow$   $CI, t$

$CI, t - \text{nums}[CI]$

$CI+1, t$

Index Target Subset



$\Rightarrow [[2, 2, 2], [2, 4]]$

\* Store the result when target sum = 0

## Code →

```
class Solution {
public:
    void totalWays(vector<int>& candidates, int target, int curr, vector<vector<int>>& res, vector<int>& aux ) {
        if(curr==candidates.size()){
            if(target==0){
                res.push_back(aux);
            }
            return;
        }
        // feasible only if curr value is less than the target
        if(candidates[curr]<=target){
            aux.push_back(candidates[curr]);
            totalWays(candidates, target-candidates[curr], curr+1, res, aux);
            aux.pop_back();
        }
        // back-tracking
        totalWays(candidates, target, curr+1, res, aux);
    }

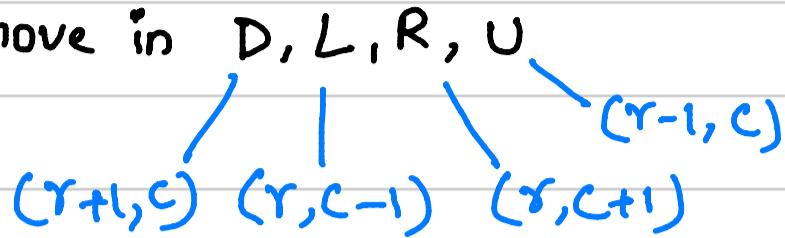
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> aux;
        totalWays(candidates, target, 0, res, aux);
        return res;
    }
};
```

D3

## ⑥ Rat in a maze

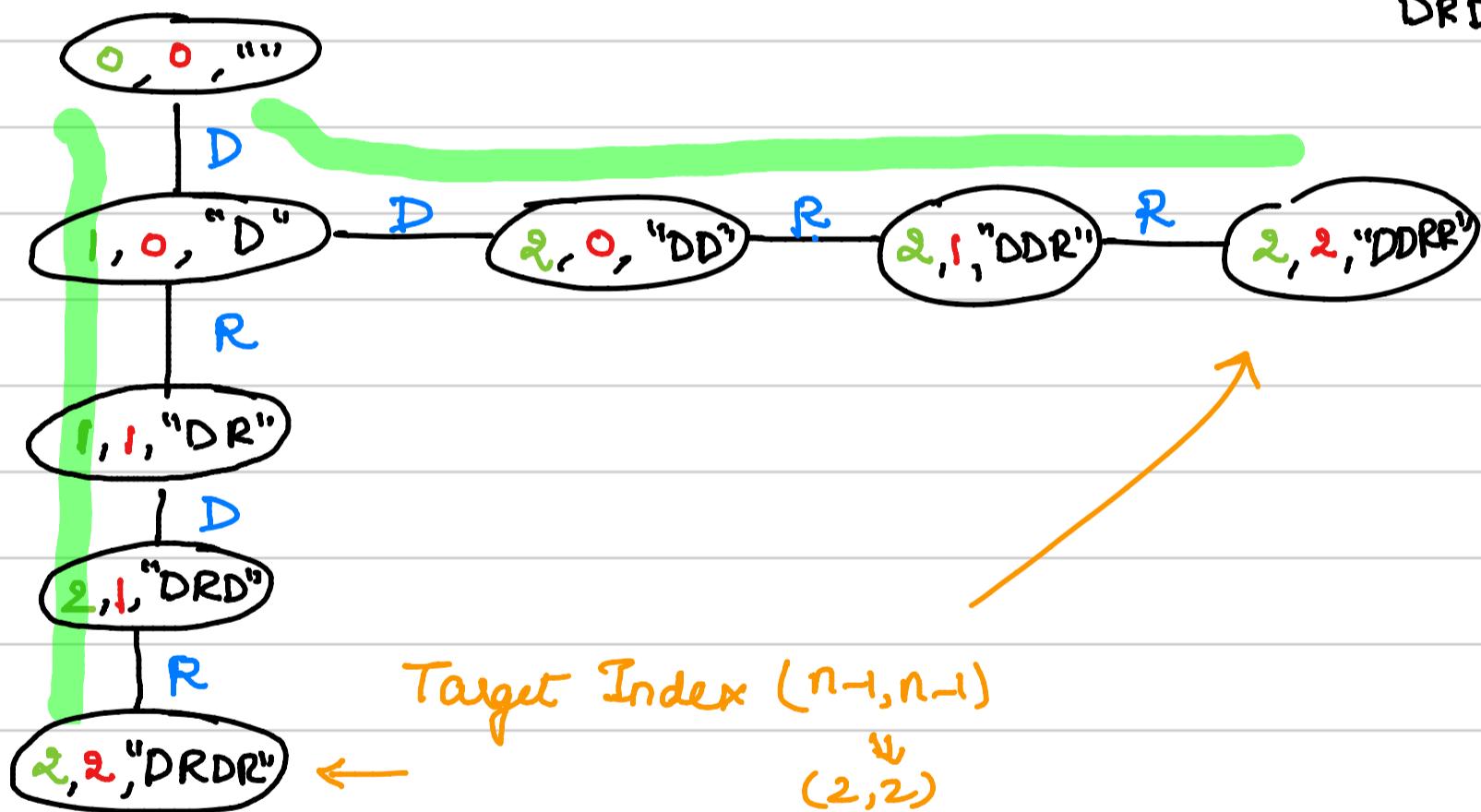
Generate all the ways to go from  $(0, 0)$  to  $(n-1, n-1)$

- \* At any cell we can move in D, L, R, U



Eg  $n=3$   $\begin{bmatrix} 0 & 1 & 2 \\ 1, 0, 0 \\ 1, 1, 0 \\ 2, 1, 1 \end{bmatrix} \Rightarrow DRDR, DDRR$

$N=4$   $\begin{bmatrix} 0 & 1, 0, 0, 0 \\ 1, 1, 0, 1 \\ 1, 1, 0, 0 \\ 0, 1, 1, 1 \end{bmatrix}$   
[DDRD, DRDR]



- \* Before making any call from cell change its state
- \* while returning, **UNDO** the changes made (Backtracking.)

Code →

```
class Solution{
public:
void allPaths(int row, int col, int n, vector<vector<int>>&m, string ans, vector<string>&res){

    if(row<0 || row>=n || col<0 || col>=n || m[row][col]==0){
        return;
    }

    if(row==n-1 && col==n-1){
        res.push_back(ans);
        return;
    }

    m[row][col]= 0;
    allPaths(row+1, col,n,m,ans+"D",res);
    allPaths(row, col-1,n,m,ans+"L",res);
    allPaths(row, col+1,n,m,ans+"R",res);
    allPaths(row-1, col,n,m,ans+"U",res);
    m[row][col] = 1;

    return;
}

vector<string> findPath(vector<vector<int>> &m, int n) {
    string ans = "";
    vector<string> res;
    allPaths(0,0,n,m,ans,res);
    sort(res.begin(), res.end());
    return res;
}
};
```

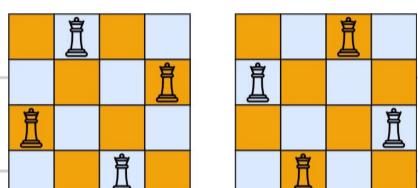
# N-queens

D4

return all configurations

- ⑦ If given  $n$ , then we should place  $n$ -queens in  $N \times N$  matrix, such that  $\text{No 2 queens} \rightarrow \text{share same row, column, diagonal}$

Eg  $n = 4$



Initially	0 1 2 3	$X_R \rightarrow \text{Bad Row}$
0	[., ., ., .]	$X_C \rightarrow \text{Bad column}$
1	[., ., ., .]	$X_D \rightarrow \text{Bad diagonal}$
2	[., ., ., .]	$X_N \rightarrow \text{Not possible}$
3	[., ., ., .]	

Step → Start from (0,0)

①

Column pos	0 1 2 3	0 1 2 3
0	[Q, ., ., .]	✓
1	[., ., ., .]	$X_R$ $X_D$ ✓ $X_D$
2	[., ., ., .]	
3	[., ., ., .]	

②

Column pos	0 1 2 3	0 1 2 3
0	[Q, ., ., .]	✓
1	[., ., Q, .]	$X_R$ $X_D$ ✓ $X_D$
2	[., ., ., .]	$X_R$ $X_D$ $X_C$ $X_D$
3	[., ., ., .]	↳ this says (1,2) is a bad config & so is (0,0) ∴ Backtrack.

③

Column pos	0 1 2 3	0 1 2 3
0	[., ., Q, .]	$X_N$ ✓
1	[., ., ., .]	$X_D$ $X_C$ $X_D$ ✓
2	[., ., ., .]	
3	[., ., ., .]	

④

Column pos	0 1 2 3	0 1 2 3
0	[., ., Q, .]	$X_N$ ✓
1	[., ., ., Q]	$X_D$ $X_C$ $X_D$ ✓
2	[., ., ., .]	✓ $X_C$ $X_D$ $X_C$
3	[., ., ., .]	

⑤

Column pos	0 1 2 3	0 1 2 3
0	[., ., Q, .]	$X_N$ ✓
1	[., ., ., Q]	$X_D$ $X_C$ $X_D$ ✓
2	[Q, ., ., .]	✓ $X_C$ $X_D$ $X_C$
3	[., ., ., .]	$X_C$ $X_C$ ✓ $X_C$

⑥

Column pos	0 1 2 3	0 1 2 3
0	[., ., Q, .]	$X_N$ ✓
1	[., ., ., Q]	$X_D$ $X_C$ $X_D$ ✓
2	[Q, ., ., .]	✓ $X_C$ $X_D$ $X_C$
3	[., ., ., Q]	$X_C$ $X_C$ ✓ $X_C$

↑ final result ∴ store & backtrack for other config.

## code →

```
● ○ ●
1 class Solution {
2 public:
3
4     bool valid_row(int curr_row, vector<vector<char>>&grid, int n){
5         for(int i = 0; i < n; i++){
6             if(grid[curr_row][i]=='Q')
7                 return false;
8         }
9         return true;
10    }
11
12    bool valid_col(int curr_col, vector<vector<char>>&grid, int n){
13        for(int i = 0; i < n; i++){
14            if(grid[i][curr_col]=='Q')
15                return false;
16        }
17        return true;
18    }
19
20    bool valid_diagonal(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
21        int i = curr_row;
22        int j = curr_col;
23        while(i>=0 && j>=0){           // Top-left diagonal
24            if(grid[i][j]=='Q')
25                return false;
26            i--; j--;
27        }
28
29        i = curr_row;
30        j = curr_col;
31        while(i>=0 && j<n){          // Top-right diagonal
32            if(grid[i][j]=='Q')
33                return false;
34            i--; j++;
35        }
36
37        i = curr_row;
38        j = curr_col;
39        while(i<n && j>=0){          // Bottom-left diagonal
40            if(grid[i][j]=='Q')
41                return false;
42            i++; j--;
43        }
44
45        i = curr_row;
46        j = curr_col;
47        while(i<n && j<n){          // Bottom-right diagonal
48            if(grid[i][j]=='Q')
49                return false;
50            i++; j++;
51        }
52
53        return true;
54    }
55}
```



```
1  bool isValid(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
2      return valid_row(curr_row, grid, n) && valid_col(curr_col, grid, n) && valid_diagonal(grid, curr_row, curr_col, n);
3  }
4
5  // Function to convert grid char to strings
6  vector<string> populate(vector<vector<char>>&grid, int n){
7      vector<string> result;
8      for(int i = 0; i<n; i++){
9          string temp = "";
10         for(int j=0; j<n; j++){
11             temp += grid[i][j];
12         }
13         result.push_back(temp);
14     }
15     return result;
16 }
17
18 void solve(vector<vector<char>>&grid, int curr_row, int n, vector<vector<string>>&ans){
19     if(curr_row==n){
20         vector<string> temp = populate(grid,n);
21         ans.push_back(temp);
22         return;
23     }
24     for(int curr_col=0; curr_col < n; curr_col++){
25         if(isValid(grid, curr_row, curr_col,n)){
26             grid[curr_row][curr_col] = 'Q';
27             solve(grid, curr_row+1, n, ans);
28             grid[curr_row][curr_col] = '.';
29         }
30     }
31 }
32
33 vector<vector<string>> solveNQueens(int n) {
34     vector<vector<string>> ans;
35     vector<vector<char>>grid(n, vector<char>(n, '.'));
36     solve(grid, 0, n, ans);
37     return ans;
38 }
39 };
```

### 13 N-Queens II

↳ need to find the total number of possibilities

\* everything is same as in N-Queens but return the no. of elements in the result.

D5

## Sudoku Solver

- ⑧ A sudoku solution must satisfy all of the following rules:

- 1 Each of the digits 1-9 must occur exactly once in each row.
- 2 Each of the digits 1-9 must occur exactly once in each column.
- 3 Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

Eg

5	3		7					
6		1	9	5				
9	8				6			
8			6					3
4		8	3					1
7		2						6
6				2	8			
		4	1	9				5
		8		7	9			



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

### Algorithm

① Let  $(i, j)$  be an empty cell

② for  $i$  from 1 to 9 :

if  $i$  is not in row, column,  $3 \times 3$  sub-grid :

a)  $\text{grid}(r, c) = i$

b) recursively fill remaining empty cells.

c) if recursion is successful :

return true

d)  $\text{grid}(r, c) = \cdot$  (backtracking)

③ return false

## Code



```
1 class Solution {
2 public:
3     bool valid_row(vector<vector<char>>&board, int currRow, int currVal){
4         for(int i=0; i<9; i++){
5             if(board[currRow][i]==currVal+'0'){
6                 return false;
7             }
8         }
9         return true;
10    }
11
12    bool valid_col(vector<vector<char>>&board, int currCol, int currVal){
13        for(int i=0; i<9; i++){
14            if(board[i][currCol]==currVal+'0'){
15                return false;
16            }
17        }
18        return true;
19    }
20
21    bool valid_grid(vector<vector<char>>&board, int currRow, int currCol, int currVal){
22        int x = 3*(currRow/3);
23        int y = 3*(currCol/3);
24        for(int i=0; i<3; i++){
25            for(int j=0; j<3; j++){
26                if(board[x+i][y+j]== currVal+'0'){
27                    return false;
28                }
29            }
30        }
31        return true;
32    }
33
34    bool isValidCell(vector<vector<char>>&board, int currRow, int currCol, int currVal){
35        return valid_row(board, currRow, currVal) && valid_col(board, currCol, currVal) &&
36        valid_grid(board, currRow, currCol, currVal);
37    }
38
39
```



```
1
2     bool sudokuSolver(vector<vector<char>>&board, int currRow, int currCol){
3         if(currRow==9)
4             return true;
5
6         int nextRow = 0;
7         int nextCol = 0;
8
9         // find next possible row n column
10        if(currCol==8){
11            nextRow = currRow+1;
12            nextCol = 0;
13        } else {
14            nextRow = currRow;
15            nextCol = currCol+1;
16        }
17
18        // if not filled then call
19        if(board[currRow][currCol]!='.'){
20            return sudokuSolver(board, nextRow, nextCol);
21        }
22
23        // try all possibilities from 1 to 9 numbers
24        for(int currVal=1; currVal<10; currVal++){
25
26            // if valid then make the change
27            if(isValidCell(board, currRow, currCol, currVal)){
28                board[currRow][currCol] = '0'+currVal;
29
30                // if already solved then return true directly
31                if(sudokuSolver(board, nextRow, nextCol)==true)
32                    return true;
33
34                // backtracking
35                board[currRow][currCol] = '.';
36            }
37        }
38
39        return false;
40    }
41    void solveSudoku(vector<vector<char>>& board) {
42        sudokuSolver(board, 0, 0);
43    }
44};
```

D6

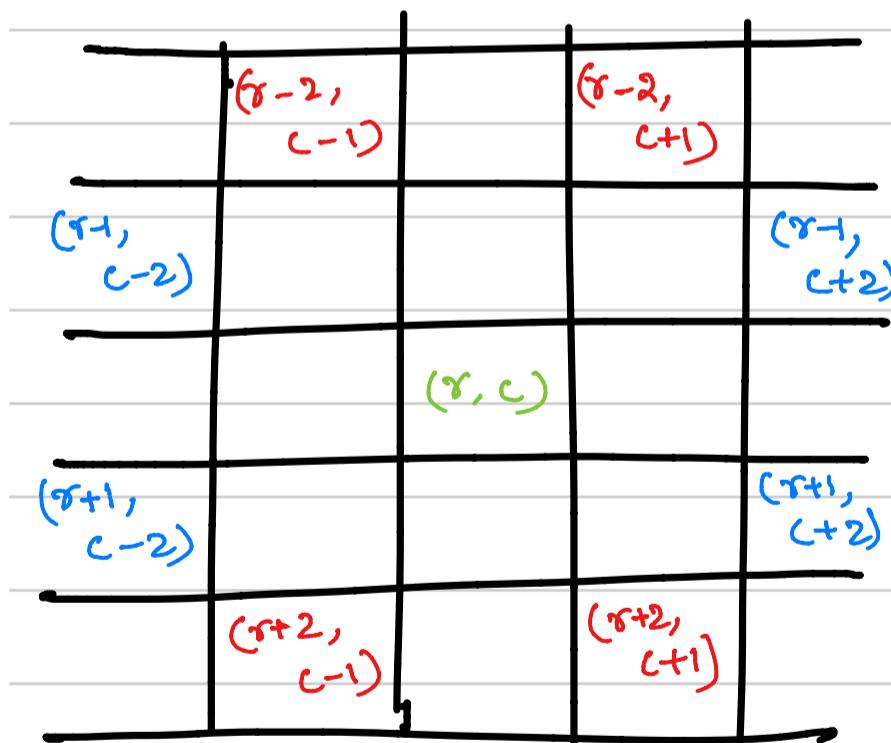
## Knight's tour problem.

- ⑨ Given an  $n \times n$  board, print the order of each cell in which they are visited. ( $n >= 8$ )

For  $n = 8$ , the result is

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Sol) For every cell  $(r, c)$  we have 8 possibilities,



- $(r-2, c-1)$
- $(r-2, c+1)$
- $(r+2, c-1)$
- $(r+2, c+1)$
- $(r-1, c-2)$
- $(r-1, c+2)$
- $(r+1, c-2)$
- $(r+1, c+2)$

- the rest is similar to rat-in-a-maze problem except that the value will be incremented by 1.

## Code →

```
● ● ●

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 void display(vector<vector<int>>&grid){
5     for(auto i: grid){
6         for(auto j:i){
7             cout<<j<<" ";
8         }
9         cout<<"\n";
10    }
11 }
12
13 void KnightTour(vector<vector<int>> &grid, int currRow, int currCol,
14                           int upcomingVal, int n){
15     if(upcomingVal==n*n){
16         display(grid);
17         cout<<"\n";
18         return;
19     }
20
21     if(currRow<0 || currRow>=n || currCol<0 || currCol>=n
22         || grid[currRow][currCol]!=0){
23         return;
24     }
25
26     grid[currRow][currCol] = upcomingVal;
27
28     KnightTour(grid, currRow-2, currCol-1, upcomingVal+1, n);
29     KnightTour(grid, currRow-2, currCol+1, upcomingVal+1, n);
30     KnightTour(grid, currRow+2, currCol-1, upcomingVal+1, n);
31     KnightTour(grid, currRow+2, currCol+1, upcomingVal+1, n);
32     KnightTour(grid, currRow-1, currCol-2, upcomingVal+1, n);
33     KnightTour(grid, currRow-1, currCol+2, upcomingVal+1, n);
34     KnightTour(grid, currRow+1, currCol-2, upcomingVal+1, n);
35     KnightTour(grid, currRow+1, currCol+2, upcomingVal+1, n);
36
37     grid[currRow][currCol] = 0;
38     return;
39 }
40
41 int main() {
42     int n;
43     cin>>n;
44     vector<vector<int>>grid(n, vector<int>(n, 0));
45     KnightTour(grid, 0, 0, 1, n);
46     return 0;
47 }
48
```

## 10 Letter combination of a phone number

Eg digits = "23" → abc  
 0 1 2 → def  
 0 1 2

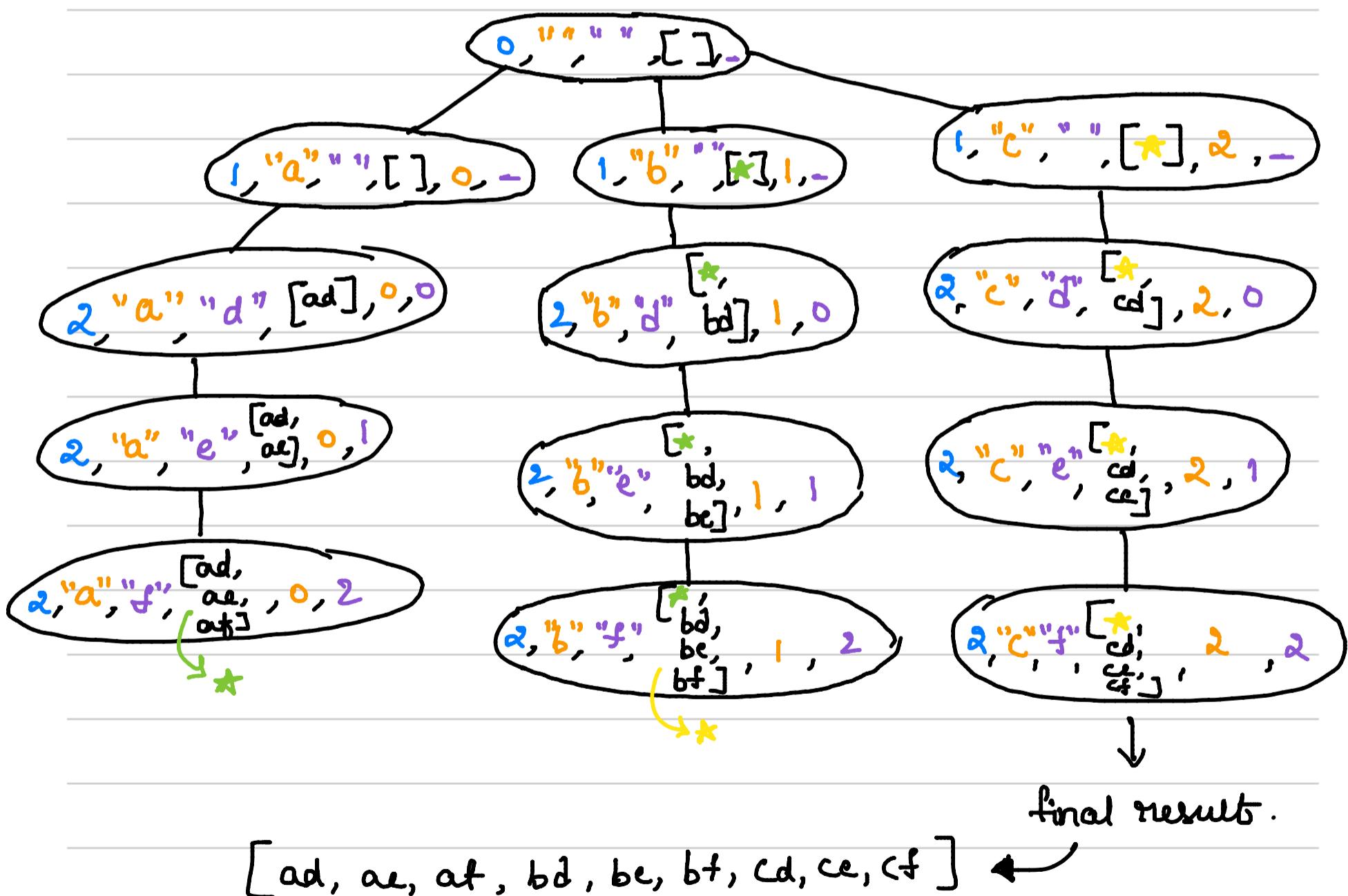
"2" → abc  
 0 1 2

"3" → def  
 0 1 2



- \* Initially create a map for numbers & their alphabets
- \* Then for each index in a string find all possibilities

//



## Code →

```
1 class Solution {
2 public:
3     void findAll( map<char,string> &mapper, string digits,
4                 vector<string> &ans, string &s, int currentIndex){
5
6         if(currentIndex>=digits.length()){
7             ans.push_back(s);
8             return;
9         }
10
11         char currNum = digits[currentIndex];
12         string alpha = mapper[currNum];
13
14         for(int i=0; i<alpha.size(); i++){
15             s.push_back(alpha[i]);
16             findAll(mapper, digits, ans, s, currentIndex+1);
17             s.pop_back();
18         }
19         return;
20     }
21
22     vector<string> letterCombinations(string digits) {
23
24         map<char,string> mapper{
25             {'1', ""}, // 1 has no letters
26             {'2', "abc"}, // 2 has 3 letters
27             {'3', "def"}, // 3 has 3 letters
28             {'4', "ghi"}, // 4 has 3 letters
29             {'5', "jkl"}, // 5 has 3 letters
30             {'6', "mno"}, // 6 has 3 letters
31             {'7', "pqrs"}, // 7 has 4 letters
32             {'8', "tuv"}, // 8 has 3 letters
33             {'9', "wxyz"} // 9 has 4 letters
34         };
35         string s = "";
36         vector<string> ans;
37
38         // edge case
39         if(digits.size()==0){
40             return ans;
41         }
42         // else generate all possibilities
43         findAll(mapper, digits, ans, s, 0);
44         return ans;
45
46     }
47 }
```

11) Subsets → same as subsets but no duplicates.

① using set<int>

Code →



```
1 class Solution {
2 public:
3     void allsubs(vector<int>& nums,int curr,
4                 vector<int>&ds, set<vector<int>>&ans)
5     {
6         if(curr>=nums.size()){
7             ans.insert(ds);
8             return;
9         }
10        int currval = nums[curr];
11        ds.push_back(currval);
12        allsubs(nums,curr+1,ds,ans);
13
14        // removing currentVal (not considering)
15        ds.pop_back();
16        allsubs(nums,curr+1,ds,ans);
17    }
18
19    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
20        set<vector<int>>ans;
21        vector<int>vec;
22        sort(nums.begin(),nums.end());
23        allsubs(nums,0,vec,ans);
24        vector<vector<int>> res{ans.begin(), ans.end()};
25        return res;
26    }
27};
```

② without using lets

code →

```
● ● ●

1 class Solution {
2 public:
3     void allsubs(vector<int> &nums, int curr, vector<int> &ds,
4                  vector<vector<int>>& res){
5         res.push_back(ds); // storing initial answers
6         for(int i=curr; i<nums.size(); i++){
7             if(i>curr && nums[i]==nums[i-1]) continue; // avoiding duplicates
8             ds.push_back(nums[i]);
9             allsubs(nums, i+1, ds, res);
10            ds.pop_back();
11        }
12        return;
13    }
14
15    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
16        vector<vector<int>> res;
17        vector<int> ds;
18        sort(nums.begin(), nums.end());
19        allsubs(nums, 0, ds, res);
20        return res;
21    }
22 };
23
24
```

12

## Combinational sum - II

→ same as Combinational Sum but no duplicates

Code →



```
1 class Solution {
2 public:
3     void findAll(vector<int>& candidates, int target, int idx,
4                  vector<vector<int>> &ans, vector<int> &ds){
5
6         if(target==0){
7             ans.push_back(ds);
8             return;
9         }
10
11        for(int i = idx; i<candidates.size(); i++){
12
13            // avoid duplicates
14            if(i>idx && candidates[i]==candidates[i-1]) continue;
15
16            if(candidates[idx]<=target){
17                ds.push_back(candidates[i]);
18                findAll(candidates, target-candidates[i], i+1, ans, ds);
19                ds.pop_back();
20            }
21        }
22    }
23
24    vector<vector<int>> combinationSum2(vector<int>& candidates,
25                                         int target){
26        vector<vector<int>> ans;
27        sort(candidates.begin(), candidates.end());
28        vector<int> ds;
29        findAll(candidates, target, 0, ans, ds);
30        return ans;
31    }
32};
```

### ⑬ N-Queens II

↳ need to find the total number of possibilities

- \* everything is same as in N-Queens but return the no. of elements in the result.

# Linked List

- Karun Karthik

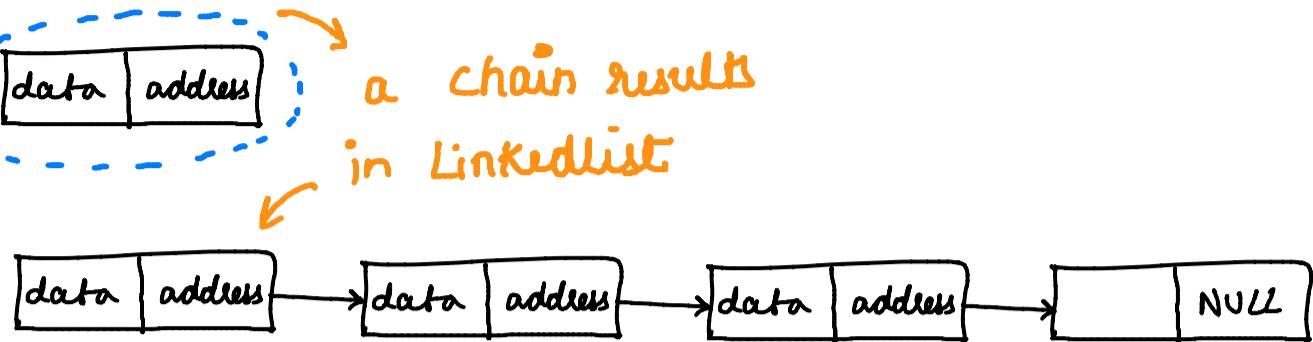
## Contents →

0. Introduction
1. Reverse a Linked List
2. Middle of Linked List
3. Delete node in a Linked List
4. Merge two sorted Lists
5. Add two numbers
6. Add two numbers II
7. Linked List Cycle
8. Linked List Cycle II
9. Remove Nth node from End of List
10. Palindrome Linked List
11. Remove duplicates from sorted List
12. Swapping nodes in Linked List
13. Odd Even Linked List
14. Swap Nodes in Pairs
15. Copy list with Random Pointer
16. Reverse Nodes in K-group
17. Design Linked List
18. Sort List

# Linked List

LinkedList is linear data structure, which consists of a group of nodes in a sequence.

Class Node  
→ data  
→ Node\*



## Advantages

1. Dynamic nature
2. Optimal insertion & deletion
3. Stacks and queues can be easily implemented
4. No memory wastage

## Disadvantages

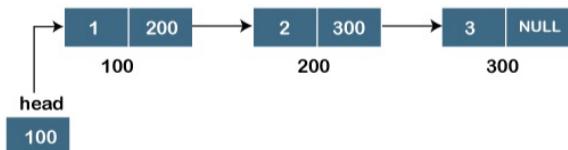
1. More memory usage due to address pointer.
2. Slow traversal compared to arrays.
3. No reverse traversal in singly linked list
4. No random access.

## Real life Applications

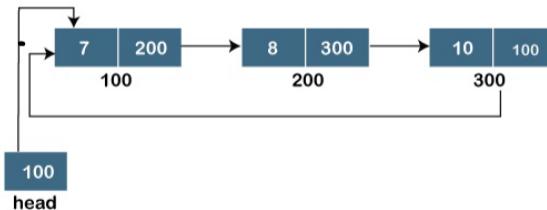
1. Previous & next page in browser
2. Image Viewer
3. Music player

## Type

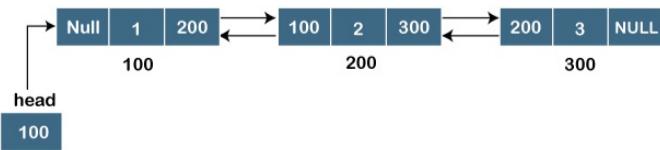
### 1. Singly linkedlist



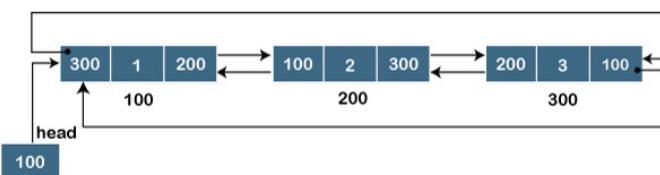
### 3. Circular Linkedlist



### 2. Doubly linkedlist



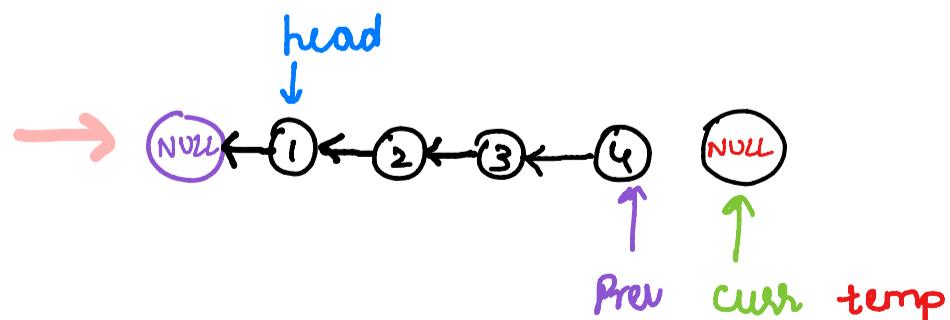
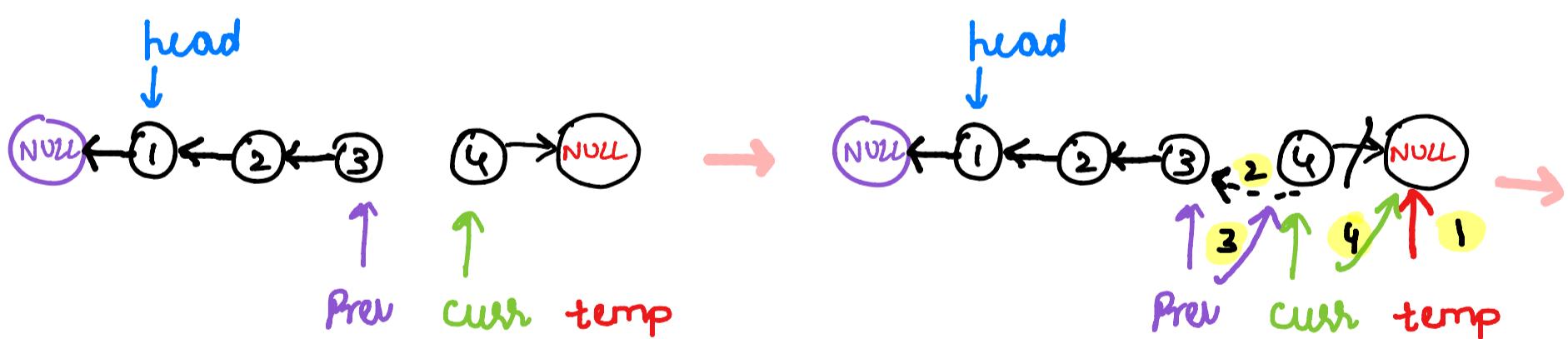
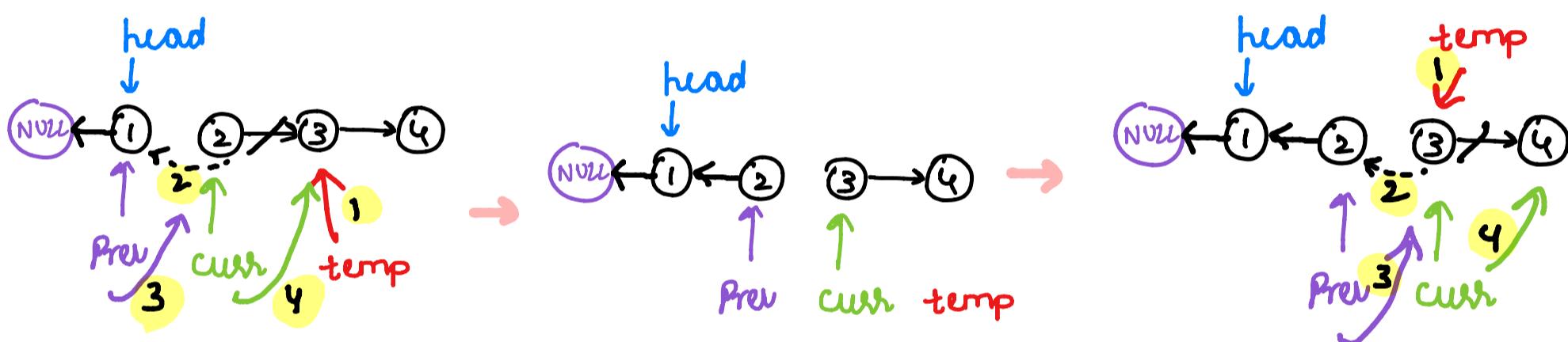
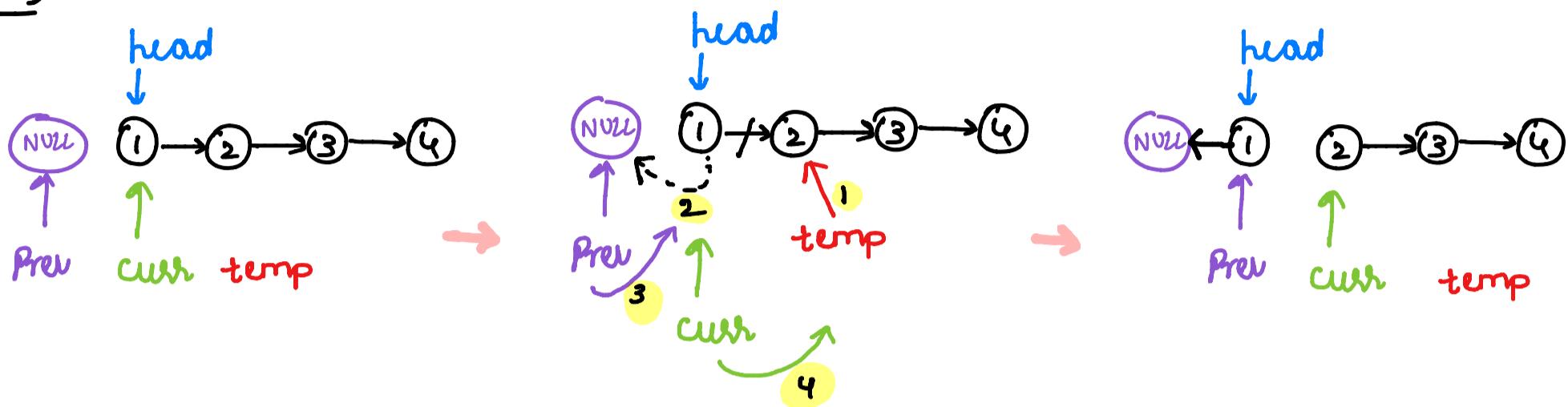
### 4. Doubly circular linkedlist



① Reverse a linkedlist → gives a linkedlist, returns reversed list.

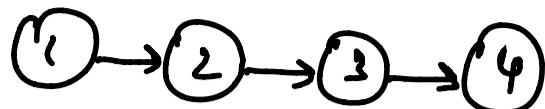
Eg  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \Rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Sol)



→ As curr points to NULL, we reached end of linkedlist  
→ Return Prev as it is the starting pointer of reversed list.

## Recursive →



curr, prev

(1, NULL):

newNode = 1 → next = 2

1 → next = null

call (2, 1)

curr, prev

(2, 1):

newNode = 2 → next = 3

2 → next = 1

call (3, 2)

curr, prev

(3, 2):

newNode = 3 → next = 4

3 → next = 2

call (4, 3)

curr, prev

(4, 3):

newNode = 4 → next = null

4 → next = 3

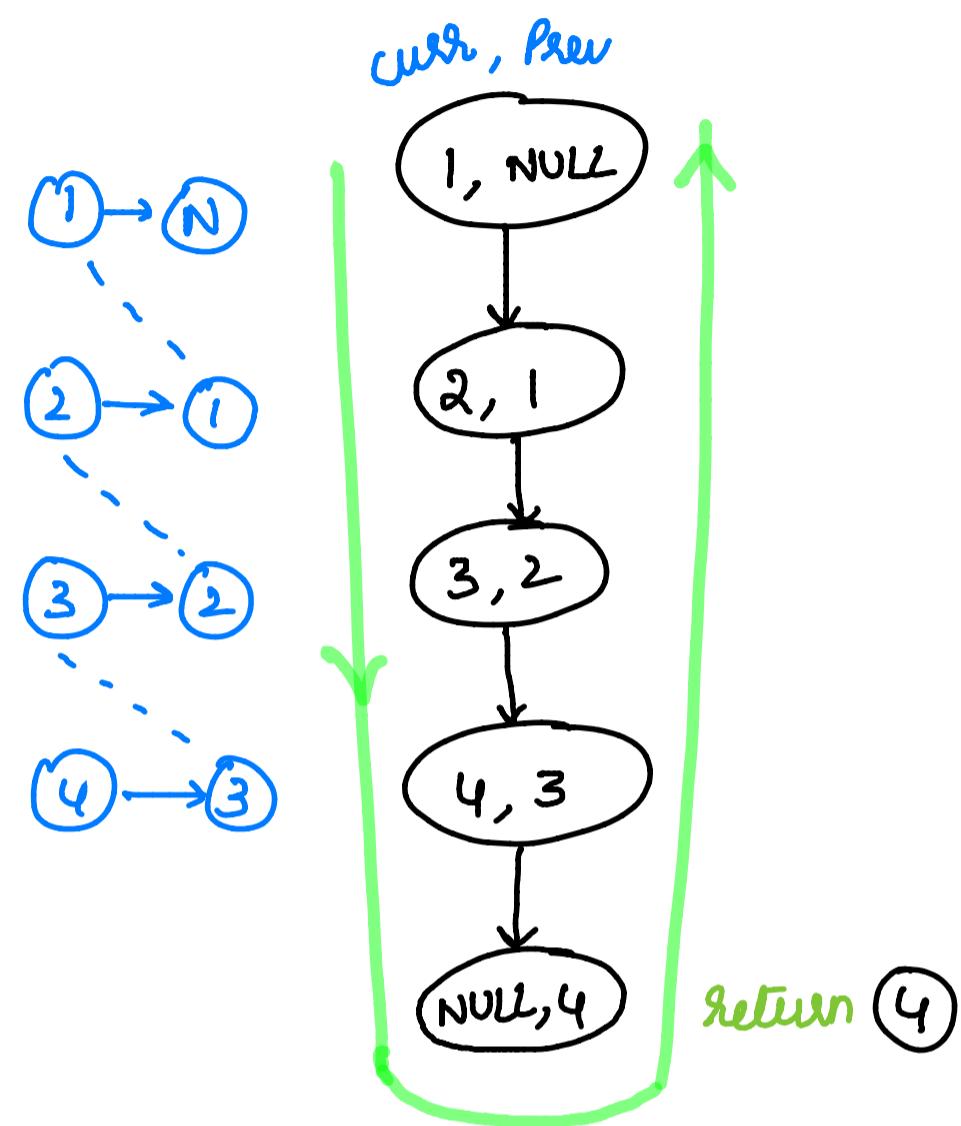
call (NULL, 4)

curr, prev

(NULL, 4):

as curr == NULL

return prev



func (curr, prev):

if curr == NULL

return prev

newNode = curr → next

curr → next = prev

recursively call newNode & curr as  
as curr → prev

Code →

$Tc \rightarrow O(n)$

$Sc \rightarrow O(1)$

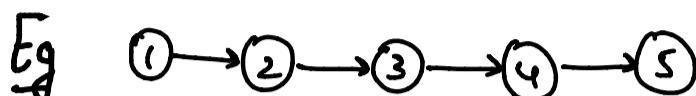


```
1
2 // Iterative ->
3 class Solution {
4 public:
5     ListNode* reverseList(ListNode* head) {
6         ListNode *prev = NULL, *curr=head, *temp;
7         while(curr){
8             temp = curr->next;
9             curr->next = prev;
10            prev = curr;
11            curr = temp;
12        }
13        return prev;
14    }
15 };
16
17
18 // Recursive ->
19 class Solution {
20 public:
21     ListNode* reverseLinker(ListNode* curr, ListNode* prev) {
22         if(curr==NULL)
23             return prev;
24         ListNode* newNode = curr->next;
25         curr->next = prev;
26         return helper(newNode, curr);
27     }
28
29     ListNode* reverseLinker(ListNode* head) {
30         return helper (head, NULL);
31     }
32 }
```

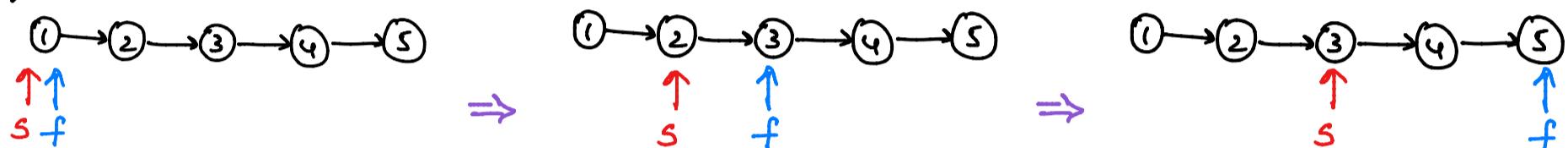
② Middle of Linkedlist → Given the head, return middle node.

Approach 1 → traverse the list & find no of nodes & return mid

Approach 2 → use 2 pointers → slow (moves by 1) } By time  
fast (moves by 2) } fast reaches end, slow points to the mid.



Res ⇒ 



code

as fast reached end  
return slow.



```
1 class Solution {
2 public:
3     ListNode* middleNode(ListNode* head) {
4         if(head == NULL)
5             return head;
6         ListNode* slow = head, *fast = head;
7
8         // Traverse the LinkedList
9         while(fast != NULL && fast -> next != NULL)
10        {
11            slow = slow -> next;
12            fast = fast -> next -> next;
13        }
14
15        return slow;
16    }
17};
```

TC → O(n)

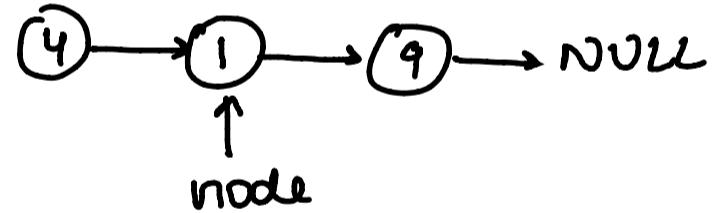
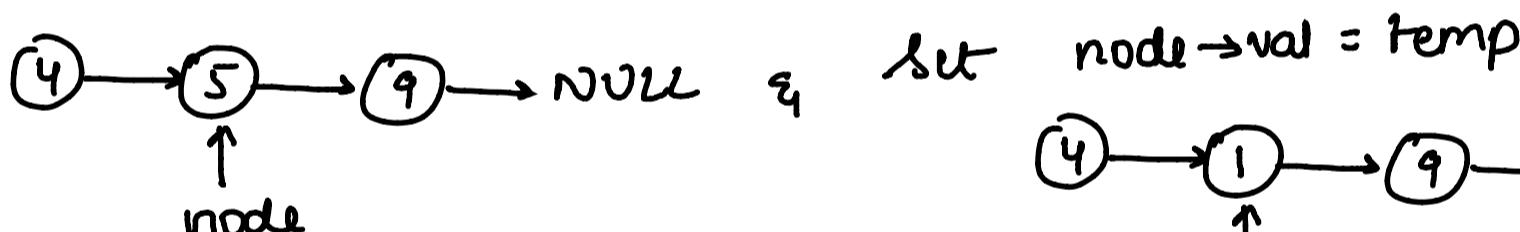
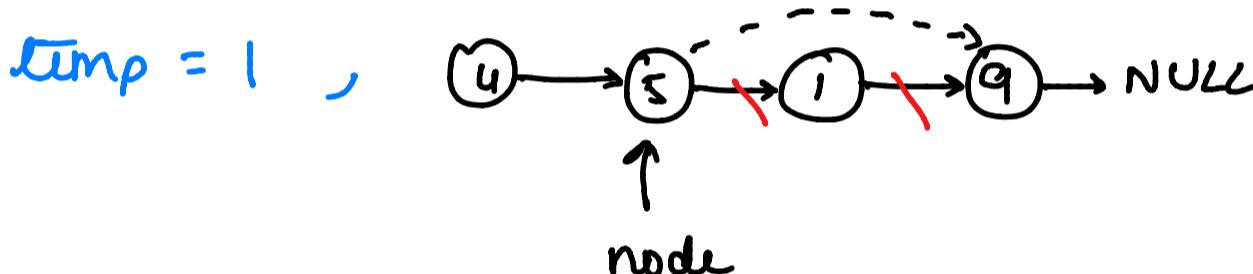
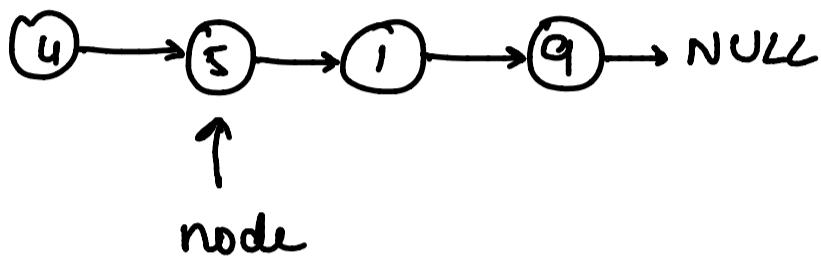
SC → O(1)

### ③ Delete node in a linkedlist →

given a linkedlist's node, delete it.

- copy node's next node's val into a temp variable
- skip the node→next node
- copy the temp variable's value into the node.

Eg

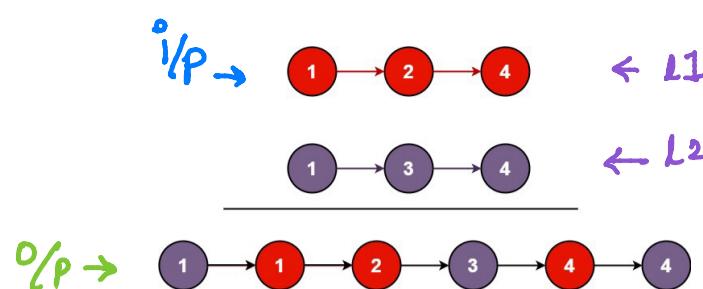


code →

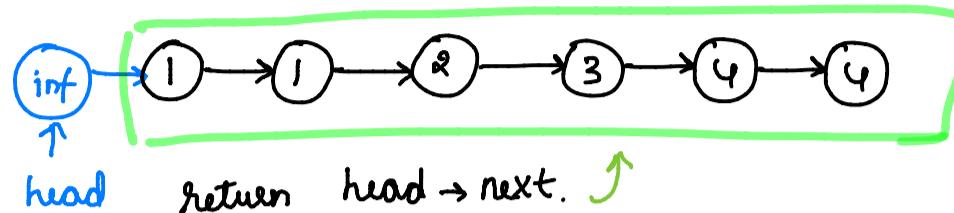
```
● ● ●  
1 class Solution {  
2 public:  
3     void deleteNode(ListNode* node) {  
4         int val = node->next->val;  
5         node->next = node->next->next;  
6         node->val = val;  
7     }  
8 }
```

Tc → O(1)  
Sc → O(1)

#### ④ Merge two sorted lists



→ Take a dummyNode & chain the next node which contains smaller value of  $L_1$  &  $L_2$



Code →

```

1  class Solution {
2  public:
3      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4
5          if( l1 == NULL ) return l2;
6          if( l2 == NULL ) return l1;
7
8          ListNode* dummy = new ListNode(-101);
9          ListNode* head = dummy;
10
11         // Traverse the lists
12         while( l1 != NULL && l2 != NULL )
13     {
14             if( l1->val < l2->val )
15             {
16                 ListNode* newnode = new ListNode(l1->val);
17                 dummy->next = newnode;
18                 l1 = l1->next;
19             }
20             else
21             {
22                 ListNode* newnode = new ListNode(l2->val);
23                 dummy->next = newnode;
24                 l2 = l2->next;
25             }
26             dummy = dummy->next;
27         }
28
29         /* If a particular list is NULL, then directly chain
30         the other */
31         if(l1!=NULL) dummy->next = l1;
32         if(l2!=NULL) dummy->next = l2;
33
34         return head->next;
35     }
36 };

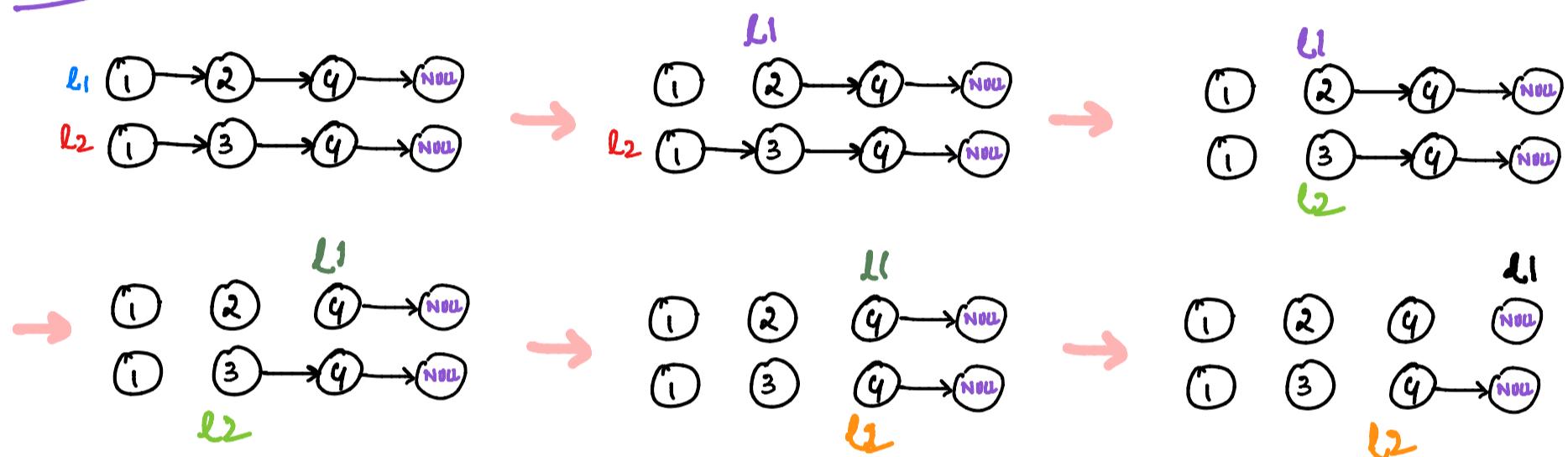
```

## Recursive Code →

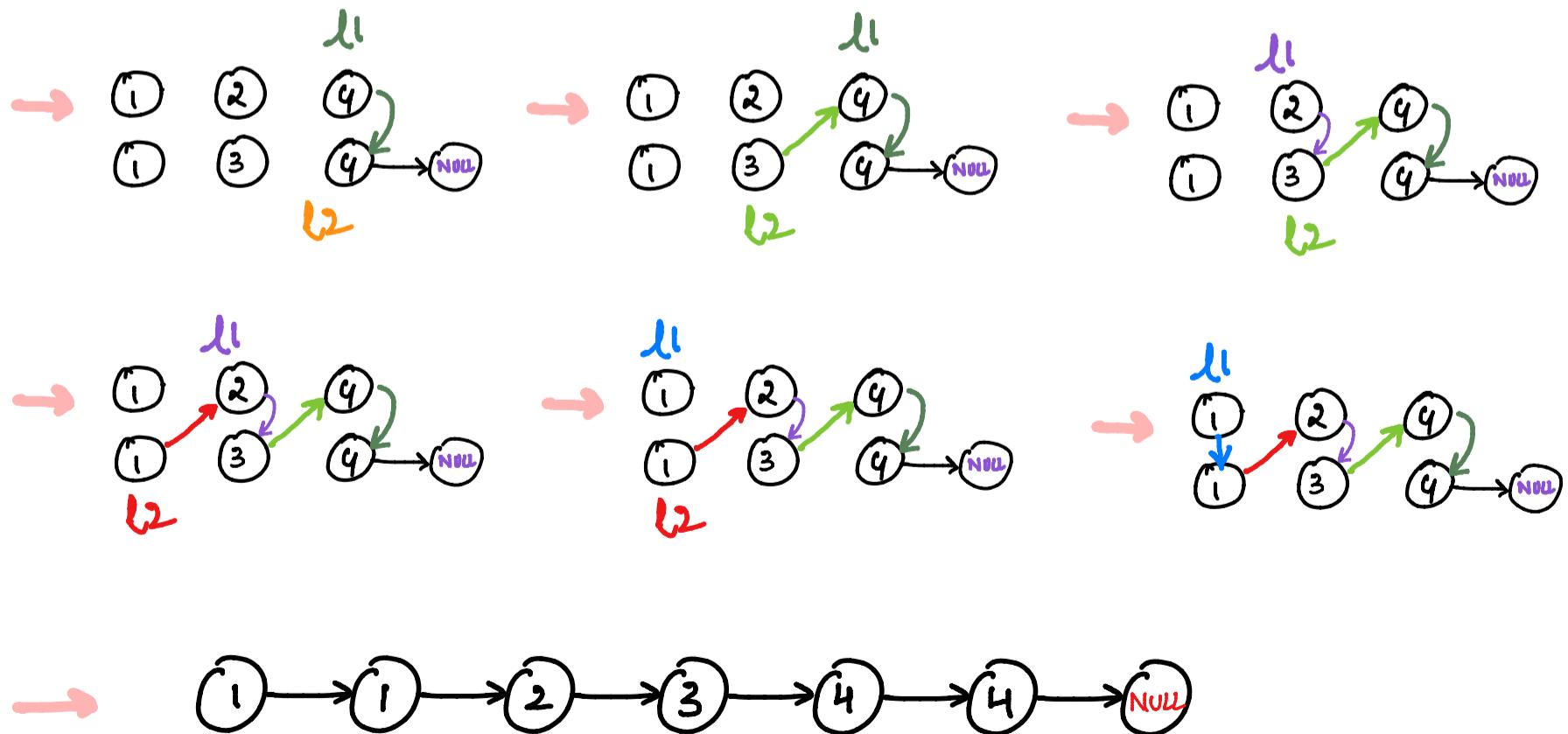
```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;

        // compare the starting values and link the nodes
        if (l1->val <= l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

## Dry Run

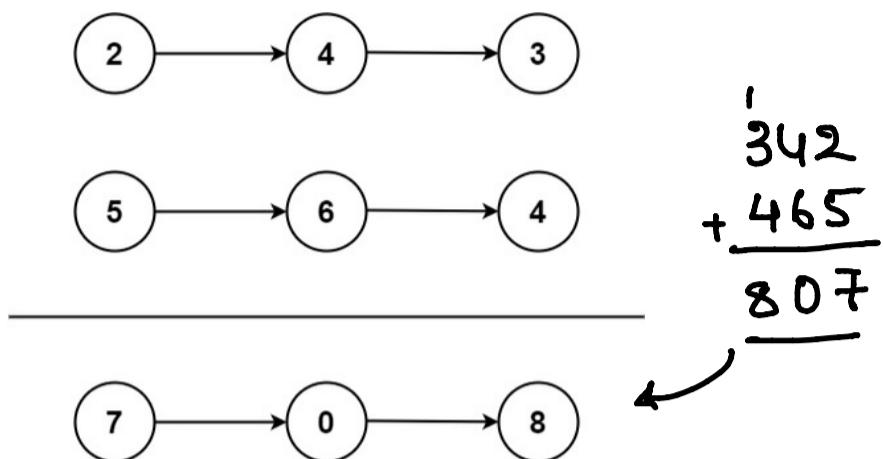


As  $l_1$  is null, return  $l_2$



## 5 Add two Numbers

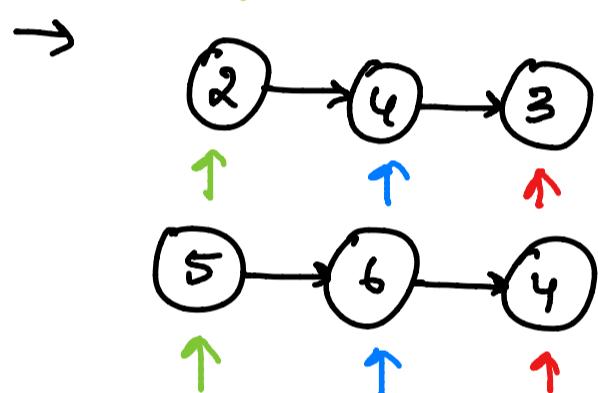
Given 2 lists in  
reverse order, add them  
and return the sum.



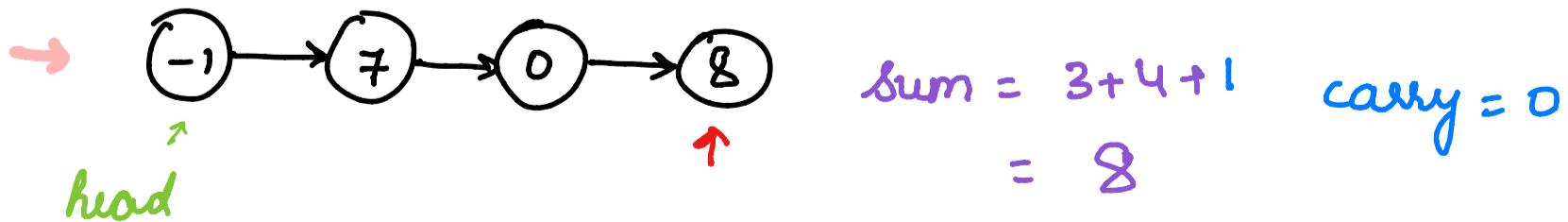
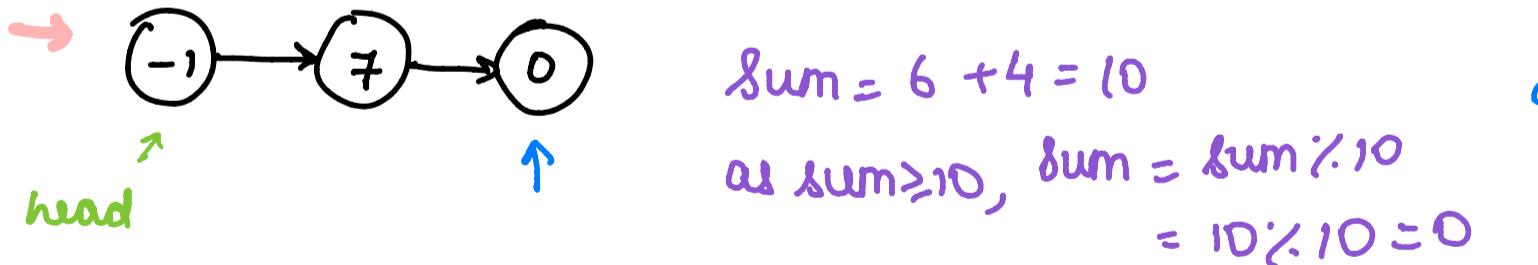
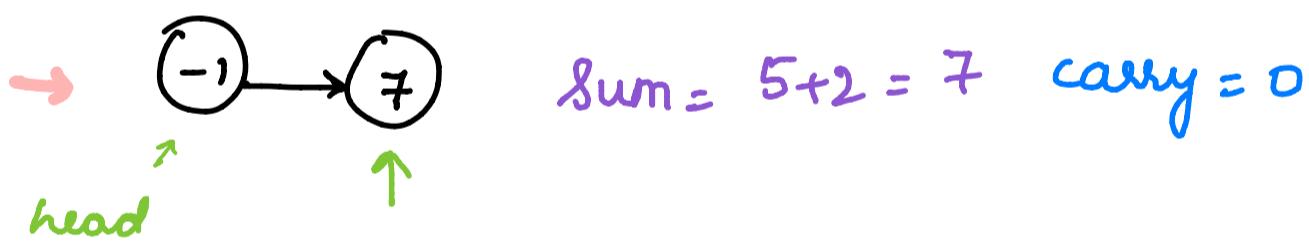
$$Tc \rightarrow O(m+n)$$

$$Sc \rightarrow O(\max(m,n))$$

Initially,



- 1) traverse both list simultaneously  
if sum  $\geq 10$ , then set carry=1
  - 2) add both values + carry
  - 3) Create newNode with this value



## code

```
1 class Solution {
2 public:
3
4     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
5
6         ListNode* dummyNode;
7         ListNode* head;
8         dummyNode = head = new ListNode(-1);
9         if(!l1)
10             return l2;
11         if(!l2)
12             return l1;
13
14         int carry = 0;
15
16         while(l1 || l2){
17             int firstVal = l1 ? l1->val : 0;
18             int secondVal = l2 ? l2->val : 0;
19
20             int total = firstVal + secondVal + carry;
21             carry = total / 10;
22             total = total % 10;
23
24             ListNode* newNode = new ListNode(total);
25             dummyNode->next = newNode;
26
27             dummyNode = dummyNode->next;
28
29             l1 = l1 ? l1->next : l1;
30             l2 = l2 ? l2->next : l2;
31         }
32
33         if(carry)
34             dummyNode->next = new ListNode(1);
35
36         return head->next;
37     }
38 }
```

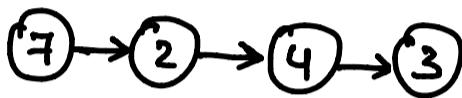
## ⑥ Add two numbers !!

→ Problem solving approach is same as previous problem

→ Points to note :

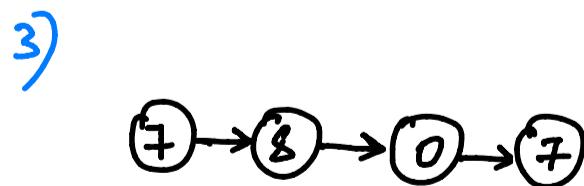
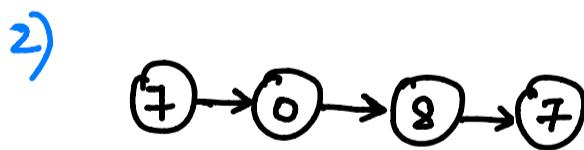
1. Reverse both the lists
2. Add them
3. Reverse the result

Eg →



Code →

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode* prev = NULL;
5         ListNode* curr = head;
6         ListNode* temp = NULL;
7         while(curr!=NULL)
8         {
9             temp = curr->next;
10            curr->next = prev;
11            prev = curr;
12            curr = temp;
13        }
14        return prev;
15    }
16
17    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
18        l1=reverseList(l1); // O(n)
19        l2=reverseList(l2); //O(n)
20        ListNode* dummyNode;
21        ListNode* head;
22        dummyNode = head = new ListNode(-1);
23        if(!l1)
24            return l2;
25        if(!l2)
26            return l1;
27
28        int carry = 0;
29
30        while(l1 || l2){
31            int firstVal = l1 ? l1->val : 0;
32            int secondVal = l2 ? l2->val : 0;
33
34            int total = firstVal + secondVal + carry;
35            carry = total / 10;
36            total = total % 10;
37
38            ListNode* newNode = new ListNode(total);
39            dummyNode->next = newNode;
40
41            dummyNode = dummyNode->next;
42
43            l1 = l1 ? l1->next : l1;
44            l2 = l2 ? l2->next : l2;
45        }
46
47        if(carry)
48            dummyNode->next = new ListNode(1);
49
50        return reverseList(head->next); //O(max(m,n))
51    }
52};
```



Result ↑

## ⑦ Linked List Cycle

$Tc \rightarrow O(n)$   
 $Sc \rightarrow O(1)$

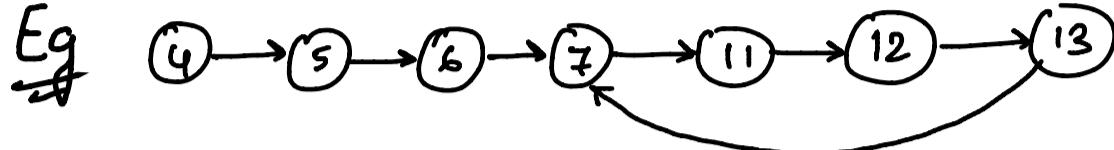
### Approach - 1

Create a set of nodes & insert every node into it,  
if already exist then return true else false.

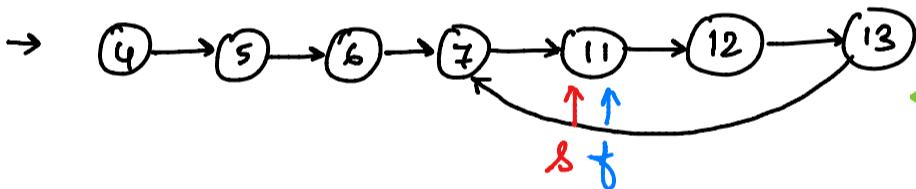
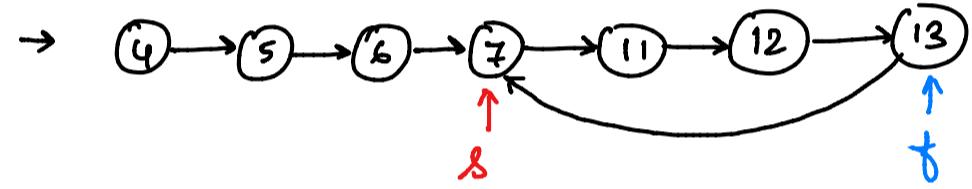
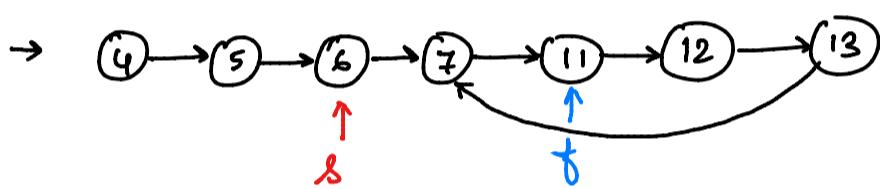
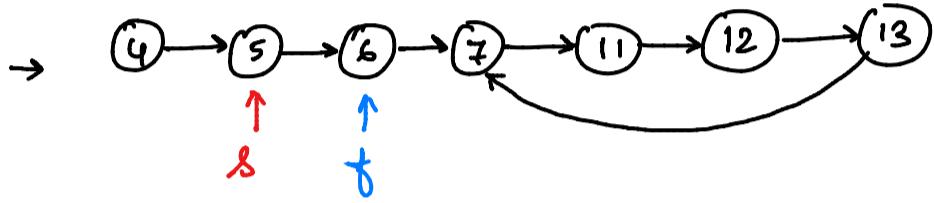
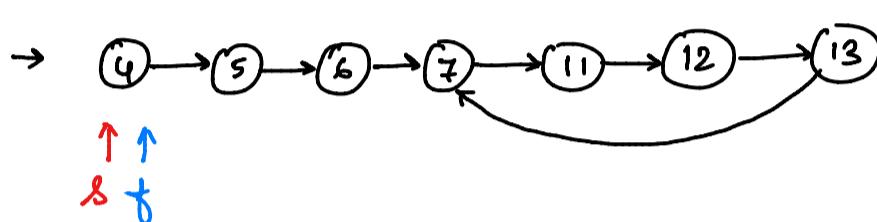
### Approach - 2

using fast & slow pointer  $Tc \rightarrow O(n)$   $Sc \rightarrow O(1)$

Eg



keep iterating till  
 $fast \rightarrow next \&& fast \rightarrow next \rightarrow next \neq null$ .

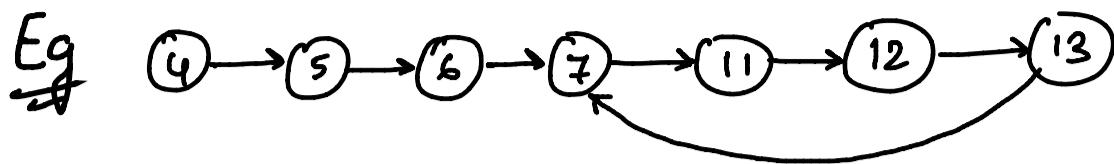


return true

Code →

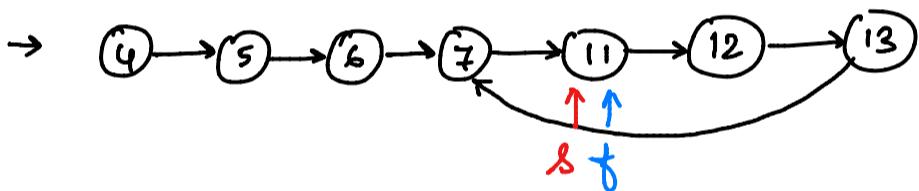
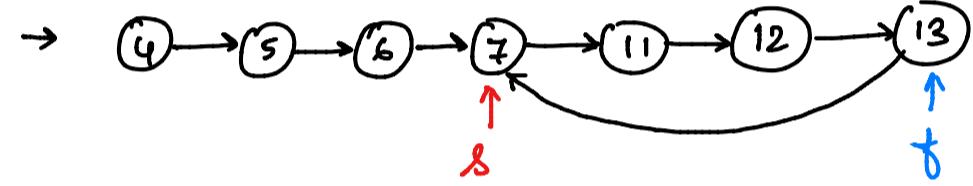
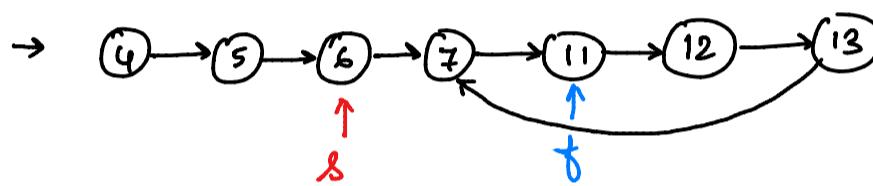
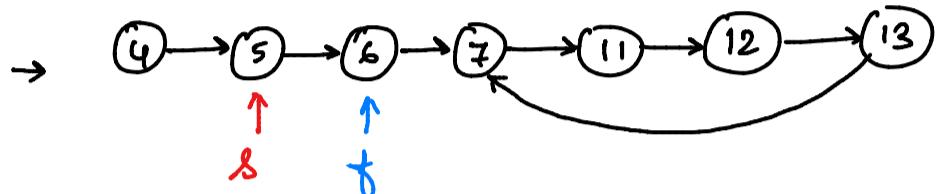
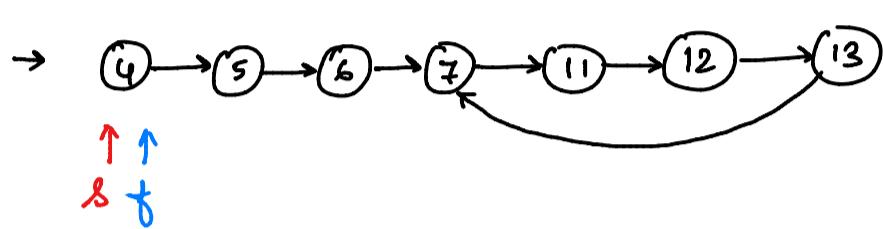
```
1 class Solution {
2 public:
3     bool hasCycle(ListNode *head) {
4         if(head==NULL) return false;
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL && fast->next->next!=NULL)
7         {
8             fast=fast->next->next;
9             slow=slow->next;
10            if(fast==slow) return true;
11        }
12        return false;
13    }
14};
```

⑧ Linked list cycle  $\Rightarrow$  returns the node where cycle begins.

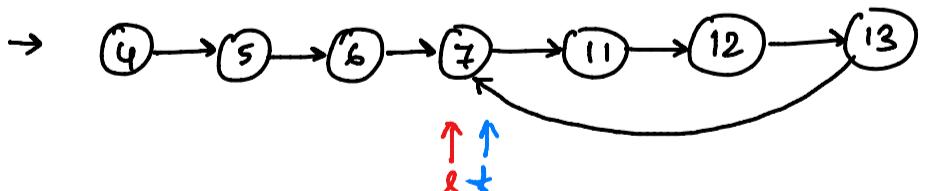
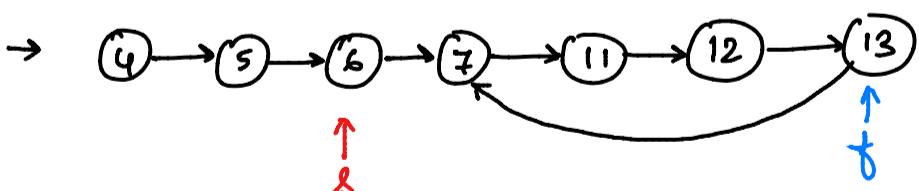
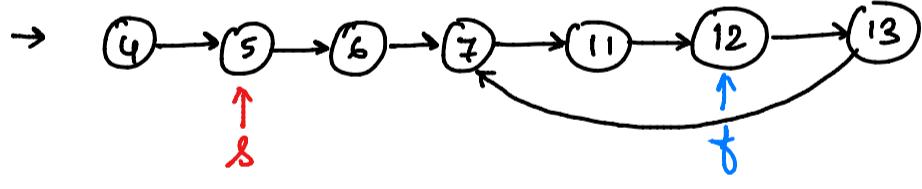
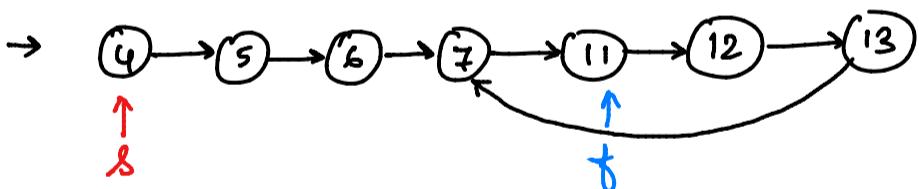


result = 7

keep iterating while  $\text{fast} \rightarrow \text{next}$  &  $\text{fast} \rightarrow \text{next} \rightarrow \text{next}$  exist.



← once  $\text{slow} == \text{fast}$ , then set  
slow = head, & move pointers  
by 1 unit



$\rightarrow$  when  $\text{slow} == \text{fast}$ , it denotes the node where cycle begins.

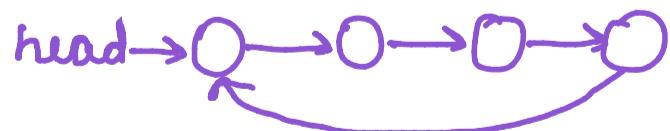
return slow;  $\Rightarrow$  7

code →

```
● ● ●  
1 class Solution {  
2 public:  
3     ListNode *detectCycle(ListNode *head) {  
4         if(head==NULL) return NULL;  
5         ListNode *fast = head, *slow = head;  
6         while(fast->next!=NULL && fast->next->next!=NULL)  
7         {  
8             fast = fast->next->next;  
9             slow = slow->next;  
10            if(fast == slow)  
11            {  
12                slow = head;  
13                while (slow != fast)  
14                {  
15                    slow = slow->next;  
16                    fast = fast->next;  
17                }  
18                return slow;  
19            }  
20        }  
21        return NULL;  
22    }  
23};
```

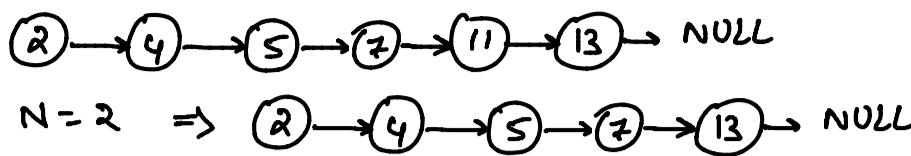
$$TC \rightarrow O(n) + O(n) = O(2n) = \underline{\underline{O(n)}}$$

\* worst case when its a loop



$$SC \rightarrow O(1)$$

## ⑨ Remove $N^{\text{th}}$ node from End of list



Approach - 1 → find length of list  $L$ , delete  $l-n+1^{\text{th}}$  node

Approach - 2 → a) Reverse b) Delete  $N^{\text{th}}$  node c) Reverse

Code →

$T_C \rightarrow O(n)$

$S_C \rightarrow O(1)$

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode *prev = NULL, *curr = head, *temp;
5         while(curr){
6             temp = curr->next;
7             curr->next = prev;
8             prev = curr;
9             curr = temp;
10        }
11        return prev;
12    }
13
14    ListNode* removeNthFromEnd(ListNode* head, int n) {
15        ListNode *dummy = new ListNode(-1);
16        dummy->next = reverseList(head);
17        head = dummy;
18        ListNode *curr = head;
19        ListNode *prev = NULL;
20        // Iteration
21        for(int i=0; i<n; i++)
22        {
23            prev = curr;
24            curr = curr->next;
25        }
26        // Deletion
27        prev->next = curr->next;
28        return reverseList(head->next);
29    }
30}
```

## ⑩ Pallindrome Linked List

Approach - 1 → Create a copy of list & reverse it. Compare value by value.  
If all are equal true else false.

Approach - 2 → Reach middle node & return the remaining list as new list. Reverse the newList & compare its value by value.

Code →

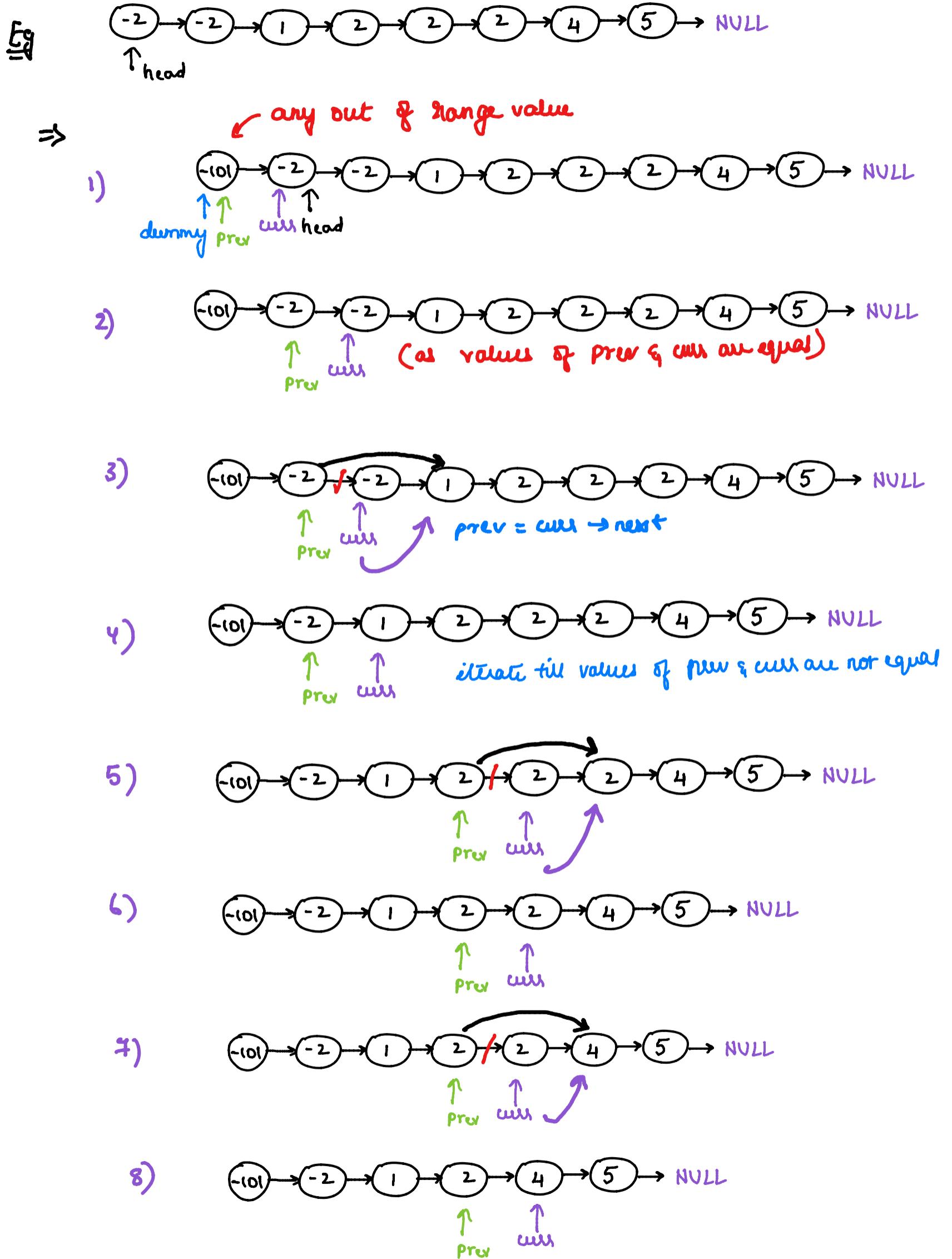
TC → O(n)

SC → O(1)

```
● ● ●
1 class Solution {
2 public:
3     ListNode* midNode(ListNode*head)
4     {
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL and fast->next->next!=NULL){
7             fast=fast->next->next;
8             slow = slow->next;
9         }
10        return slow;
11    }
12    ListNode* reverseList(ListNode* head) {
13        ListNode *prev = NULL, *curr = head, *temp;
14        while(curr!=NULL)
15        {
16            temp=curr->next;
17            curr->next = prev;
18            prev = curr;
19            curr = temp;
20        }
21        return prev;
22    }
23    bool compare(ListNode*l1,ListNode* l2)
24    {
25        while(l1!=NULL && l2!=NULL)
26        {
27            if(l1->val!=l2->val)    return false;
28            l1 = l1->next;
29            l2 = l2->next;
30        }
31        return true;
32    }
33    bool isPalindrome(ListNode* head) {
34        if(head==NULL) return false;
35        if(head->next == NULL) return true;
36        ListNode *mid = midNode(head);
37        ListNode *l2 = mid->next;
38        mid->next = NULL;
39        l2 = reverseList(l2);
40        return compare(head,l2);
41    }
42};
```

# (1) Remove duplicates from sorted list →

Given a linkedlist, return linkedlist without duplicates.



code →

```
● ● ●

1  class Solution {
2  public:
3      ListNode* deleteDuplicates(ListNode* head) {
4          ListNode* dummy = new ListNode(101);
5          dummy->next = head;
6          ListNode* curr = head;
7          ListNode* prev = dummy;
8          while(curr!=NULL)
9          {
10              if(curr->val==prev->val){
11                  prev->next = curr->next;
12                  curr = curr->next;
13              } else {
14                  prev = curr;
15                  curr = curr->next;
16              }
17          }
18          return dummy->next;
19      }
20  };
21
22
23 // Another approach
24
25 ListNode* deleteDuplicates(ListNode* head) {
26     if(head==NULL || head->next==NULL) return head;
27     ListNode *curr = head;
28     while(curr->next!=NULL){
29         if(curr->val == curr->next->val){
30             curr->next = curr->next->next;
31         } else {
32             curr = curr->next;
33         }
34     }
35     return head;
36 }
```

$$T_C \rightarrow O(n)$$

$$S_C \rightarrow O(1)$$