Stack & Queues

Stack → Linear data structure

- follows LIFO, last in first out.
- operations → push : insert into top of stack
  pop : delete from top of stack.

Applications →

- by compilers to check for parenthesis
- to evaluate postfix expression
- to convert infix to postfix / prefix form.
- to store values during recursion & context during function call.
- to implement DFS of graph

Queue → Linear data structure

- follows FIFO, first in first out.
- operations → enqueue : insert element at end of queue
  dequeue : delete element at start of queue

Applications →

- Schedule jobs by CPU.
- to carry out FIFO basis like printing jobs.
- to implement BFS of graph

Types →

- Queue
- Circular Queue
- Doubly ended Queue
- Priority Queue.

① Implement a Stack using Linkedlist →

code →

```
#include <bits/stdc++.h>
using namespace std;

struct Node{
    int data;
    Node* next;
};

Node* top;

void push(int data){
    Node* temp = new Node();
    if (!temp){
        cout << "\nStack Overflow";
        exit(1);
    }
    // add at the top and change top as new node
    temp->data = data;
    temp->next = top;
    top = temp;
}

int isEmpty(){
    // if top is null then empty
    return top == NULL;
}

int peek(){
    // if stack is not empty then return top node's data
    if (!isEmpty())
        return top->data;
    else
        exit(1);
}

void pop(){
    Node* temp;
    if(top == NULL){
        cout << "\nStack Underflow" << endl;
        exit(1);
    } else {
        temp = top;
        top = top->next;
        free(temp);
    }
}
```

(2) Implement a Queue using Linkedlist →

code →

```
class Node {
    int data;
    Node* next;
    Node(int d){
        data = d;
        next = NULL;
    }
};

class Queue {
    Node *front, *rear;

    Queue(){
        front = rear = NULL;
    }

    void enQueue(int x)
    {
        Node* temp = new Node(x);
        // if empty then node is both front and rear
        if (rear == NULL) {
            front = rear = temp;
            return;
        }
        // else add at end
        rear->next = temp;
        rear = temp;
    }

    void deQueue()
    {
        // if empty then return NULL
        if (front == NULL)
            return;
        // store front node
        Node* temp = front;
        front = front->next;

        // if front is NULL => no Nodes, change rear to NULL
        if (front == NULL)
            rear = NULL;
        // free node
        delete (temp);
    }
};
```

③ Implement a Stack using Queue →

If push, push into queue from rear end & pop & push all elements

If pop, pop from queue from front end.

Code →

```
class Stack {
  queue <int> q;

  public:

    // push operation
    void Push(int x) {
        int n = q.size();
        q.push(x);
        for (int i = 0; i < n; i++)
        {
            int value = q.front();
            q.pop();
            q.push(value);
        }
    }

    // pop operation
    int Pop() {
        int value = q.front();
        q.pop();
        return value;
    }

    // accessing top value
    int Top() {
        return q.front();
    }

    // finding size of stack
    int Size() {
        return q.size();
    }
};
```

④ Implement a Queue using Stack →

→ use 2 stacks.

→ while pop(), shift all elements in 1 stack to another. & return top value.

code →

```
class Queue {
  public:
    stack <int> in;
    stack <int> out;

    // push operation
    void Push(int x) {
        in.push(x);
    }

    // pop operation
    int Pop() {
        // shift in to out
        if (out.empty()){
            while (in.size()){
                out.push(in.top());
                in.pop();
            }
        }
        int x = out.top();
        out.pop();
        return x;
    }

    // peek operation
    int Top() {
        if (out.empty()){
            while (in.size()){
                out.push(in.top());
                in.pop();
            }
        }
        return out.top();
    }

    int Size() {
        return in.size()+out.size();
    }
};
```

⑤ Valid parenthesis

s = "{}" → T s = "(){}" → T

s = "{[]}" → T s = ")[" → F

Eg s = "{[]()}()[]([])" → True.

→ if match found then pop, else push.

stack : [ ] s = "{[]()}()[]([])"

stack : [{] s = "{[]()}()[]([])"

stack : [{[ ] s = "{[]()}()[]([])"

stack : [{~~[~~] s = "{[]()}()[]([])"

stack : [{( ] s = "{[]()}()[]([])"

stack : [{~~(~~] s = "{[]()}()[]([])"

stack : [~~{~~] s = "{[]()}()[]([])"

stack : [( ] s = "{[]()}()[]([])"

stack : [~~(~~] s = "{[]()}()[]([])"

stack : [[ ] s = "{[]()}()[]([])"

stack : [~~[~~] s = "{[]()}()[]([])"

stack : [( ] s = "{[]()}()[]([])"

stack : [([ ] s = "{[]()}()[]([])"

stack : [(~~[~~] s = "{[]()}()[]([])"

stack : [~~(~~] s = "{[]()}()[]([])"

∴ As the stack is empty & string is completely traversed the string is valid ∴ return true.

Code →

```cpp
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        for(auto i : s)
        {
            if (st.empty() || i == '(' || i == '{' || i == '[')
            {
                st.push(i);
            }
            else
            {
                if ((i == ')' && st.top() != '(') ||
                    (i == ']' && st.top() != '[') ||
                    (i == '}' && st.top() != '{')){
                    return false;
                }
                st.pop();
            }
        }
        return st.empty();
    }
};
```

TC → O(n)

SC → O(n)

6 Asteroid Collision →

only consider magnitude

+ve sign => right direction
-ve sign => left direction

If x & y collide then min(x,y) will be removed
If x == y then both will be removed.

Eg [5, 10, -5]

5,10 will not collide
10,-5 will collide & -5 will be removed

result = [5,10]

Eg [10, 6, -8, -8, 8, 9]

stack		[10, 6, -8, -8, 8, 9]	
stack	10,	[10, 6, -8, -8, 8, 9]	
stack	10, 6	[10, 6, -8, -8, 8, 9]	as 6 is +ve push
stack	10, 6	[10, 6, -8, -8, 8, 9]	as 6 & 8 will collide (opp directions), 6 will be removed
stack	10	[10, 6, -8, -8, 8, 9]	as 10 & 8 will collide (opp directions), 8 will be removed
stack	10	[10, 6, -8, -8, 8, 9]	as 10 & 8 will collide (opp directions), 8 will be removed
stack	10, 8	[10, 6, -8, -8, 8, 9]	as 8 is +ve push
stack	10, 8, 9	[10, 6, -8, -8, 8, 9]	as 9 is +ve push

result = [10, 8, 9]

TC → O(2n) ≃ O(n) SC → O(n)

worstcase

code →

```cpp
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& asteroids) {

        vector<int> res;

        for(int i=0; i< asteroids.size(); i++){

            if(res.empty() || asteroids[i]>0)
                res.push_back(asteroids[i]);
            else {

                while(!res.empty() && res.back()>0 && res.back()<abs(asteroids[i])) {
                    res.pop_back();
                }

                if(!res.empty() && res.back()+asteroids[i]==0)
                    res.pop_back();
                else if(res.empty() || res.back()<0)
                    res.push_back(asteroids[i]);
            }
        }
        return res;
    }
};
```

7 Next greater element → [2, 4, 1, 3, 1, 6]

2 → 4 3 → 6
4 → 6 1 → 6
1 → 3 6 → -1

Eg [4, 5, 2, 25]

4 → 5 2 → 25
5 → 25 25 → -1

→ Iterate from last & compare its value with top of stack
→ If stack is greater then its the next greater element
→ else keep popping till the next greater element is found.

Eg [11, 13, 3, 10, 7, 21, 26]

←

Stack = [ ]	[11, 13, 3, 10, 7, 21, 26]	
Stack = [26]	[11, 13, 3, 10, 7, 21, 26]	26 → -1
Stack = [26, 21]	[11, 13, 3, 10, 7, 21, 26]	21 → 26
Stack = [26, 21, 7]	[11, 13, 3, 10, 7, 21, 26]	7 → 21
Stack = [26, 21, ~~7~~, 10]	[11, 13, 3, 10, 7, 21, 26]	pop 7, push 10 10 → 21
Stack = [26, 21, 10]	[11, 13, 3, 10, 7, 21, 26]	3 → 10
Stack = [26, 21, ~~10~~, ~~3~~, 13]	[11, 13, 3, 10, 7, 21, 26]	pop 3,10 push 13 13 → 21
Stack = [26, 21, 13]	[11, 13, 3, 10, 7, 21, 26]	11 → 13

res = [13, 21, 10, 21, 21, 26, -1]

Code →

```cpp
class Solution
{
    public:
    //Function to find the next greater element for each element of the array.
    vector<long long> nextLargerElement(vector<long long> arr, int n){

        stack<long long> st;
        vector<long long> res(n);

        for(int i=n-1; i>=0 ; i--){
            long long currVal = arr[i];

            while(!st.empty() && st.top()<=currVal)
                st.pop();

            res[i] = st.empty()?-1:st.top();
            st.push(currVal);
        }
        return res;
    }
};
```

TC → O(n)

SC → O(n)

(8) Next Smaller element →

→ entire approach is similar to next greater element except for comparision.

code →

Tc → O(n)

Sc → O(n)

```cpp
vector<int> nextSmallerElement(vector<int> &arr, int n)
{
    stack<int> st;
    vector<int> res(n);
    for(int i=n-1; i>=0 ; i--){

        long long currVal = arr[i];

        while(!st.empty() && st.top()>=currVal)
            st.pop();

        res[i] = st.empty()?-1:st.top();
        st.push(currVal);
    }
    return res;
}
```

(9) Stock Span Problem → given price quotes of stock for n days.

we need to find span of stock on any particular day.

max no. of consecutive days for which price <= curr day's price

Eg [100, 80, 60, 70, 60, 75, 85]
  0, 1, 2, 3, 4, 5, 6

stack = [stores indexes]

span = [0 | 0 | 0 | 0 | 0 | 0 | 0]
  0 1 2 3 4 5 6

if currentElement > stack.top
  pop stack
else:
  span = currentIndex - stack.top

→ push index into stack after processing →

0 1 2 3 4 5 6		stack	span
[100, 80, 60, 70, 60, 75, 85]	span of 1st element = 1	[0]	[~~0~~ 1 | 0 | 0 | 0 | 0 | 0 | 0]
[100, 80, 60, 70, 60, 75, 85]	80 > 100 ⇒ false ∴ span = 1−0 = 1	[0,1]	[1 | ~~0~~ 1 | 0 | 0 | 0 | 0 | 0]
[100, 80, 60, 70, 60, 75, 85]	60 > 100 ⇒ false ∴ span = 2−1 = 1	[0,1,2]	[1 | 1 | ~~0~~ 1 | 0 | 0 | 0 | 0]
[100, 80, 60, 70, 60, 75, 85]	70 > 60 ⇒ true ∴ pop 70 > 80 ⇒ false ∴ span = 3−1 = 2	[0,1,3]	[1 | 1 | 1 | ~~0~~ 2 | 0 | 0 | 0]
[100, 80, 60, 70, 60, 75, 85]	60 > 70 ⇒ false ∴ span = 4−3 = 1	[0,1,3,4]	[1 | 1 | 1 | 2 | ~~0~~ 1 | 0 | 0]
[100, 80, 60, 70, 60, 75, 85]	75 > 60 ⇒ true ∴ pop 75 > 70 ⇒ true ∴ pop 75 > 80 ⇒ false span = 5 − 1 = 4	[0,1,5]	[1 | 1 | 1 | 2 | 1 | ~~0~~ 4 | 0]
[100, 80, 60, 70, 60, 75, 85]	85 > 75 ⇒ true ∴ pop 85 > 80 ⇒ true ∴ pop 85 > 100 ⇒ false span = 6 − 0 = 6	[0,6]	[1 | 1 | 1 | 2 | 1 | 4 | ~~0~~ 6]

$\boxed{\text{span} = [1 \mid 1 \mid 1 \mid 2 \mid 1 \mid 4 \mid 6]}$
  0 1 2 3 4 5 6

Code →

Tc → O(n)

Sc → O(n)

```cpp
class Solution
{
    public:
    //Function to calculate the span of stocks price for all n days.
    vector <int> calculateSpan(int price[], int n)
    {
        vector<int> span(n);
        stack<int> st;

        st.push(0);
        span[0] = 1;

        for(int i=1; i<n; i++){

            int currPrice = price[i];

            while(!st.empty() && currPrice >= price[st.top()])
                st.pop();

            if(st.empty()){
                span[i] = i+1;
            } else {
                span[i] = i-st.top();
            }

            st.push(i);
        }
        return span;
    }
};
```

(10) Celebrity Problem →

A Celebrity is a person, who is known to everyone & knows none.
Given a square matrix M & if i^{th} person knows j^{th} person then $M[i][j] = 1$, else 0.

Eg →

$$M = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \left[\begin{array}{l} [0,1,0], \\ [0,0,0], \\ [0,1,0] \end{array} \right] \end{array} \qquad n = 3.$$

① create stack & push values from 0 to n-1.
② do the following till stack has more than 1 value.
- pop 1st element & set it to A
- pop again & set it to B
- if A knows B then push B else A.

stack → [ ] ⇒ stack [0, 1, 2]

⇒ stack $[0, \cancel{1}, \cancel{2}]$ A = 2, B = 1 & $M[2][1] == 1$ (true) ∴ push 1 ⇒ stack [0,1]

stack $[\cancel{0}, \cancel{1}]$ A = 1, B = 0 & $M[1][0] == 1$ (false) ∴ push 1 ⇒ stack [1]

∴ as stack has only 1 element, stop.

⇒ Now pop the stack & consider it as celebrity & check for
- anyone doesn't know celeb (!M[i][celeb])
- if celeb knows anyone (M[celeb][i])

} return -1.

∴ from i=0 to 2 & celeb=1

i=0 (!M[0][1] or M[1][0]) = 0
i=1 skip as celeb is i
i=2 (!M[2][1] or M[1][2]) = 0

all are failed i.e no violation of conditions.

∴ return celeb i.e 1

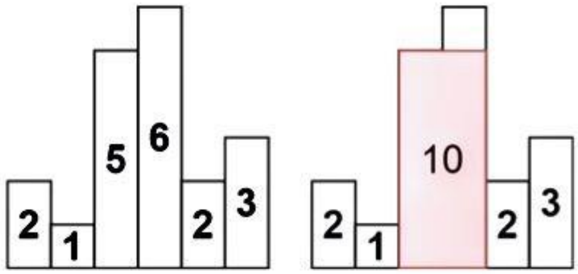
code →

TC = O(n)

SC = O(n)

```cpp
class Solution
{
    public:
    //Function to find if there is a celebrity in the party or not.
    int celebrity(vector<vector<int> >& M, int n) {

        stack<int> s;

        for(int i=0;i<n;i++)    s.push(i);

        // check and if is a celebrity then push into stack
        while(s.size()>1)
        {
            int a=s.top();
            s.pop();
            int b=s.top();
            s.pop();

            if(M[a][b]==1)
                s.push(b);
            else
               s.push(a);
        }

        int celeb = s.top();

        for (int i = 0; i < n; i++){
            // if i person doesn't know celeb or celeb knows anyone else
            // then return -1
            if ( (i!=celeb) && (!M[i][celeb]) || M[celeb][i] ))
                return -1;
        }

       return celeb;
    }
};
```

(11) Largest Rectangle in Histogram →



⇒ given an array of heights, return area of largest rectangle

Ans = 10.

0 1 2 3 4 5   Stack.

arr = [2, 1, 5, 6, 2, 3]   [ ]   area = 0   maxArea = 0

i=0   [2, 1, 5, 6, 2, 3]   [0]   area = 0   maxArea = 0

→ i=1   [2, 1, 5, 6, 2, 3]   [0]   area = 0   maxArea = 0

now arr[st.top()] > currElement ⇒ ht = arr[st.top()] & st.pop()   [~~0~~]

(2   1   2)

& as stack is empty now, width = i & push(i)

(1   1)

∴ ht = 2 & width = 1 ∴ area = 2 & maxArea = ~~0~~ 2.

→ i=2   [2, 1, 5, 6, 2, 3]   [1]   area = 0   maxArea = 2

now arr[st.top()] > currElement ⇒ false ∴ push(i)

(1   5   2)

→ i=3   [2, 1, 5, 6, 2, 3]   [1, 2]   area = 0   maxArea = 2

now arr[st.top()] > currElement ⇒ false ∴ push(i)

(5   6   3)

→ i=4   [2, 1, 5, 6, 2, 3]   [1, 2, 3]   area = 0   maxArea = 2

now arr[st.top()] > currElement ⇒ ht = arr[st.top()] & st.pop()   [1, 2, ~~3~~]

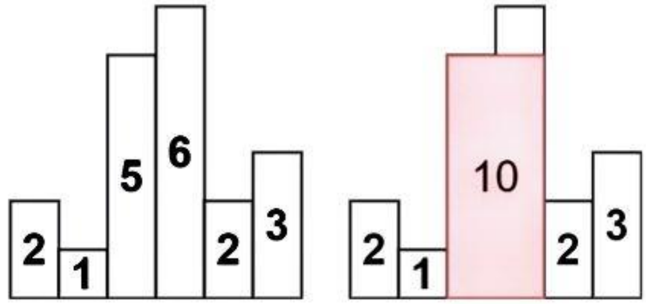
(6   2   6)

width = i − st.top() − 1 = 1 ∴ area = 6 * 1 = 6   maxArea = ~~2~~ 6.

(4   2   1)

& push(i)

(4)



0 1 2 3 4 5

→ [2, 1, 5, 6, 2, 3]   [1, 2]   area = 10   maxArea = 10

i=4   [1, ~~2~~]

now $\overset{5}{arr[st.top()]}$ > $\overset{2}{currElement}$ ⇒ ht = $\overset{5}{arr[st.top()]}$ & st.pop()

width = $\overset{4}{i}$ - $\overset{1}{st.top()}$ - 1 = 2   ∴ area = 5*2 = 10   maxArea = ~~2~~ 10.

& push(i)^{4}

0 1 2 3 4 5

→ [2, 1, 5, 6, 2, 3]   [1, 4]   area = 0   maxArea = 10

i=5

now $\overset{2}{arr[st.top()]}$ > $\overset{3}{currentElement}$ ⇒ false ∴ push(i)^{5}

⇒ Last iteration to pop stack. ⇒ i = 6

0 1 2 3 4 5

→ [2, 1, 5, 6, 2, 3]   [1, 4, 5]   area = 3   maxArea = 10

ht = $\overset{3}{arr[st.top()]}$ & pop() & as stack is not empty   [1, 4, ~~5~~]

width = $\overset{6}{i}$ - $\overset{4}{st.top()}$ - 1 = 1   ∴ area = 3*1 = 3   maxArea = 10

0 1 2 3 4 5

→ [2, 1, 5, 6, 2, 3]   [1, 4]   area = 3   maxArea = 10

ht = $\overset{2}{arr[st.top()]}$ & pop() & as stack is not empty   [1, ~~4~~]

width = $\overset{6}{i}$ - $\overset{1}{st.top()}$ - 1 = 4   ∴ area = 2*4 = 8   maxArea = 10

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ \end{array}$$

→ [2, 1, 5, 6, 2, 3] [1,] area = 6 maxArea = 10

[~~1~~,]

ht = arr[st.top()] & pop() & as stack is empty

width = i = 6 ⇒ ∴ area = 1 * 6 = 6 maxArea = 10

∴ As stack is empty return maxArea = 10.

Code →

TC → O(n)

SC → O(n)

```cpp
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        stack < int > st;
        int maxArea = 0;
        int n = heights.size();

        for (int i = 0; i <= n; i++) {

            while (!st.empty() && (i == n || heights[st.top()] >= heights[i])) {

                int height = heights[st.top()];
                st.pop();
                int width;
                if (st.empty()){
                    width = i;
                } else {
                    width = i - st.top() - 1;
                }

                int area = width*height;
                maxArea = max(maxArea, area);
            }
            st.push(i);
        }
        return maxArea;
    }
};
```

(12) Sliding Window Maximum →

→ process first 'k' elements before pushing into result arr.

→ if dq.front() == i−k then pop-front (out of boundary case)

→ if nums[dq.back()] < nums[i] then pop-back
(meaningless to store smaller elements in window)

→ if i >= k−1 then push nums[dq.front()]

Eg nums = [1, 3, −1, −3, 5, 3, 6, 7] k = 3 res = [3, 3, 5, 5, 6, 7]

⇒ nums | deque | res

[1, 3, −1, −3, 5, 3, 6, 7]
0 1 2 3 4 5 6 7
deque: (empty) res: [ ]

i=0 [1, 3, −1, −3, 5, 3, 6, 7]
deque: 0 res: [ ]

i=1 [1, 3, −1, −3, 5, 3, 6, 7]
deque: 0 res: [ ]
→ dq.front == i−k → false
nums[0] < nums[i] (1 < 3)
∴ pop back & push i
deque: ~~0~~ 1

i=2 [1, 3, −1, −3, 5, 3, 6, 7]
deque: 1, 2 res: [3]
→ dq.front == i−k → false
nums[1] < nums[i] (3, −1)
∴ false & push i

→ as i >= k−1
push num[dq.front()] i.e. 3
into res

i=3 $[\overset{0}{1}, \overset{1}{3}, \overset{2}{-1}, \overset{3}{-3}, \overset{4}{5}, \overset{5}{3}, \overset{6}{6}, \overset{7}{7}]$

→ dq.front == i-K → false

$\text{num}[\overset{-1}{2}] < \overset{-3}{\text{num}}[i]$

∴ false & push i

1, 2, 3

→ as i >= k-1

push num[dq.front()] ie 3

into res

[3, 3]

i=4 $[\overset{0}{1}, \overset{1}{3}, \overset{2}{-1}, \overset{3}{-3}, \overset{4}{5}, \overset{5}{3}, \overset{6}{6}, \overset{7}{7}]$

→ dq.front == i-K   true ∴ popfront

$\overset{-3}{\text{num}[3]} < \overset{5}{\text{num}[i]}$ ∴ pop_back

$\overset{-1}{\text{num}[2]} < \overset{5}{\text{num}[i]}$ ∴ pop_back

& push(i)

order of pop

~~1~~, ~~2~~, ~~3~~, 4

→ as i >= k-1

push num[dq.front()] ie 5

into res

[3, 3, 5]

i=5 $[\overset{0}{1}, \overset{1}{3}, \overset{2}{-1}, \overset{3}{-3}, \overset{4}{5}, \overset{5}{3}, \overset{6}{6}, \overset{7}{7}]$

→ dq.front == i-K → false

$\overset{5}{\text{num}[4]} < \overset{3}{\text{num}[i]}$

∴ false & push(i)

order of pop

4, 5

→ as i >= k-1

push num[dq.front()] ie 5

into res

[3, 3, 5, 5]

i=6 $[\overset{0}{1}, \overset{1}{3}, \overset{2}{-1}, \overset{3}{-3}, \overset{4}{5}, \overset{5}{3}, \overset{6}{6}, \overset{7}{7}]$

→ dq.front == i-K → false

$\overset{3}{\text{num}[5]} < \overset{6}{\text{num}[i]}$ ∴ pop_back

$\overset{5}{\text{num}[4]} < \overset{6}{\text{num}[i]}$ ∴ pop_back

& push

order of pop

~~4~~, ~~5~~, 6

→ as i >= k-1

push num[dq.front()] ie 6

into res

[3, 3, 5, 5, 6]

0 1 2 3 4 5 6 7
i=7 [1, 3, -1, -3, 5, 3, 6, 7]

order of pop
~~6~~, 7

[3, 3, 5, 5, 6, 7]

→ dq.front == i-K → false

6 7
nums[6] < nums[i] ∴ pop_back

& push(i)

→ as i >= k-1
push nums[dq.front()] ie 7
into res

code →

TC → O(N)
SC → O(K)

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque <int> dq;
        vector <int> ans;
        for (int i = 0; i < nums.size(); i++) {

            if (!dq.empty() && dq.front() == i - k)
                dq.pop_front();

            while (!dq.empty() && nums[dq.back()] < nums[i])
                dq.pop_back();

            dq.push_back(i);

            if (i >= k - 1)
                ans.push_back(nums[dq.front()]);
        }
        return ans;
    }
};
```

Linked List

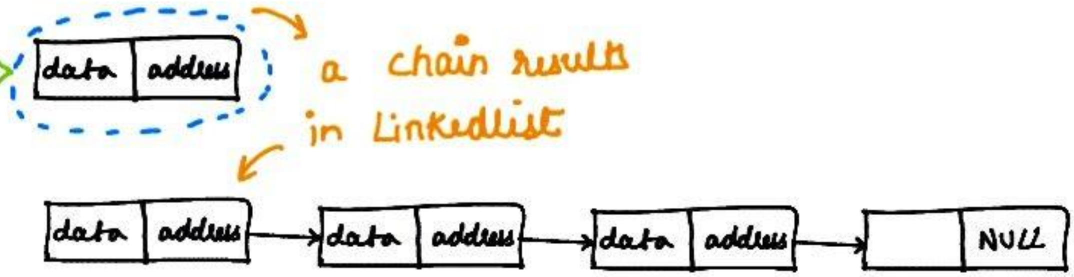
– Karun Karthik

Contents →

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

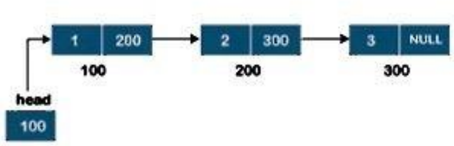
1. More memory usage due to address pointer.
2. Slow traversal compared to arrays.
3. No reverse traversal in singly linked list
4. No random access.

Real life Applications

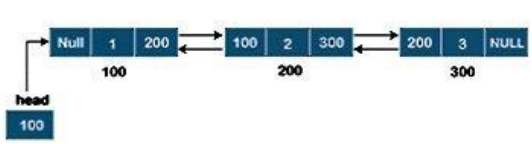
1. Previous & next page in browser
2. Image Viewer
3. Music player

Types

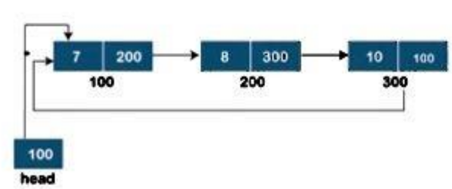
1. Singly Linkedlist



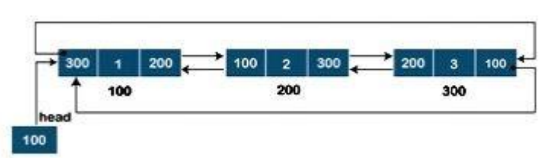
2. Doubly Linkedlist



3. Circular Linkedlist



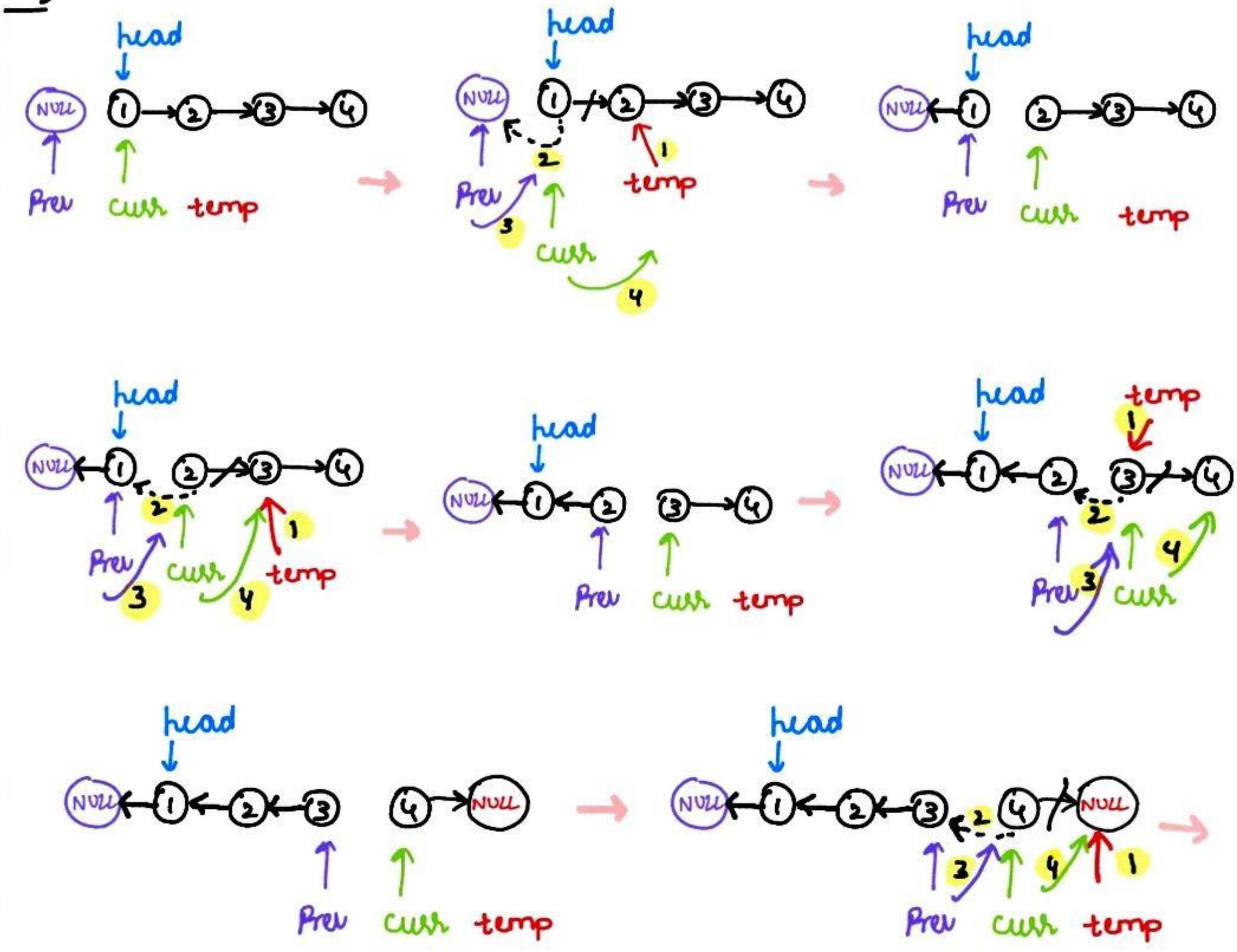
4. Doubly circular linkedlist

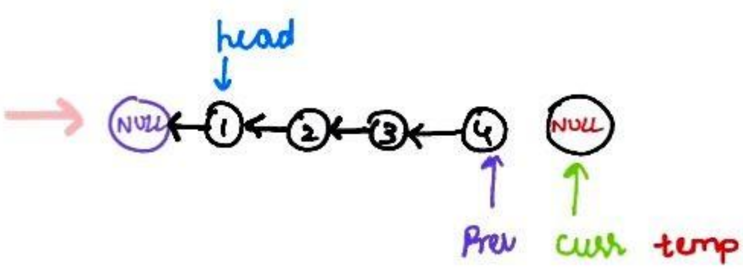


① Reverse a linkedlist → given a linkedlist, return reversed list.

Eg ① → ② → ③ → ④ ⇒ ④ → ③ → ② → ①

Sol)





→ As curr points to NULL, we reached end of Linkedlist

→ Return Prev as it is the starting pointer of reversed list.

Recursive →

1 → 2 → 3 → 4

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

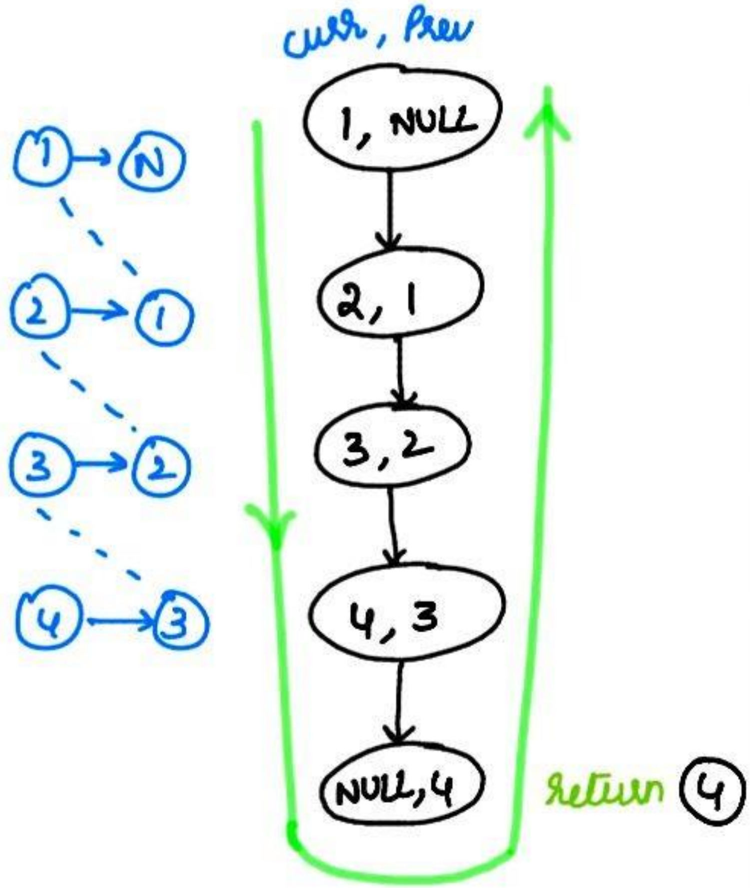
4 → next = 3

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 as curr & curr as prev

Code →

Tc → O(n)

Sc → O(1)

```cpp

// Iterative ->
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr=head, *temp;
        while(curr){
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
};


// Recursive ->
class Solution {
public:
    ListNode* reverseLinker(ListNode* curr, ListNode* prev) {
        if(curr==NULL)
            return prev;
        ListNode* newNode = curr->next;
        curr->next = prev;
        return helper(newNode, curr);
    }

    ListNode* reverseLinker(ListNode* head) {
        return helper (head, NULL);
    }
};
```

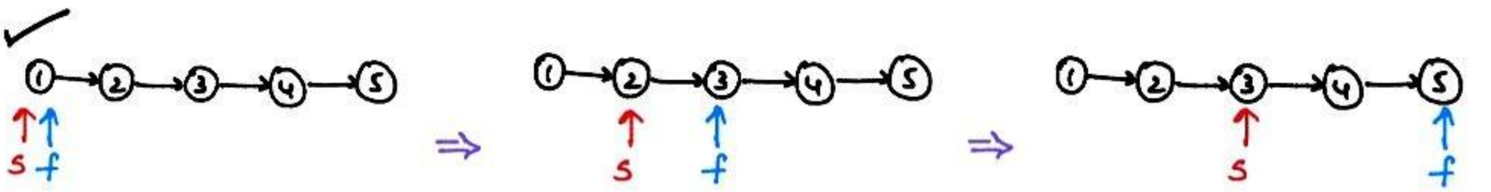
2. Middle of Linkedlist → given the head, return middle node.

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

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

Eg 1 → 2 → 3 → 4 → 5

Res ⇒ 3 → 4 → 5



as fast reached end return slow.

3 → 4 → 5

Code

```cpp
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        if(head == NULL)
            return head;
        ListNode* slow = head, *fast = head;

        // Traverse the LinkedList
        while(fast != NULL && fast -> next != NULL)
        {
            slow = slow -> next;
            fast = fast -> next -> next;
        }

        return slow;
    }
};
```

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 →

4 → 5 → 1 → 9 → NULL
↑
node

temp = 1 ,

4 → 5 → ~~→~~ 1 ~~→~~ 9 → NULL
↑
node

4 → 5 → 9 → NULL
↑
node

& set node → val = temp

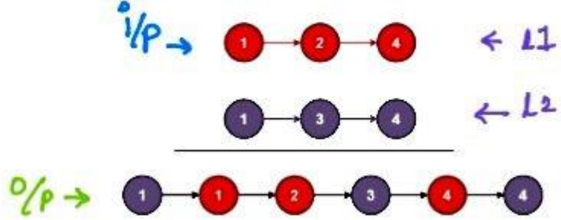
4 → 1 → 9 → NULL
↑
node

code →

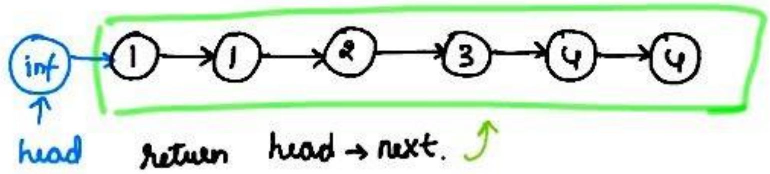
```cpp
class Solution {
public:
    void deleteNode(ListNode* node) {
        int val = node->next->val;
        node->next = node->next->next;
        node->val = val;
    }
};
```

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 →

Tc → O(m+n)

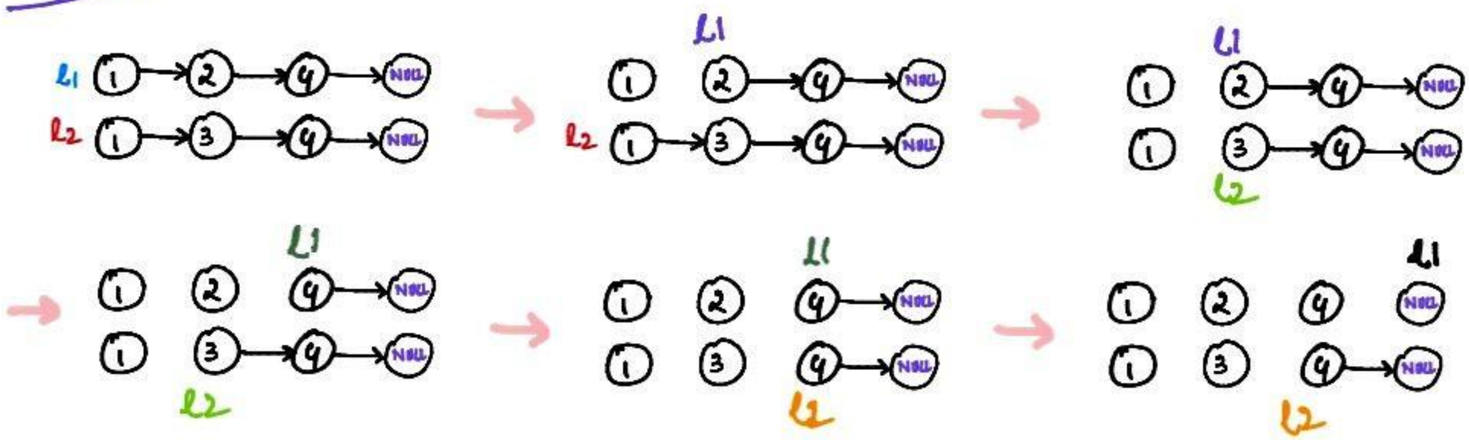
Sc → O(m+n)

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

        if( l1 == NULL ) return l2;
        if( l2 == NULL ) return l1;

        ListNode* dummy = new ListNode(-101);
        ListNode* head = dummy;

        // Traverse the lists
        while( l1 != NULL && l2 != NULL )
        {
            if( l1->val < l2->val )
            {
                ListNode* newnode = new ListNode(l1->val);
                dummy->next = newnode;
                l1 = l1->next;
            }
            else
            {
                ListNode* newnode = new ListNode(l2->val);
                dummy->next = newnode;
                l2 = l2->next;
            }
            dummy = dummy->next;
        }

        /* If a particular list is NULL, the directly chain
        the other */
        if(l1!=NULL) dummy->next = l1;
        if(l2!=NULL) dummy->next = l2;

        return head->next;
    }
};
```

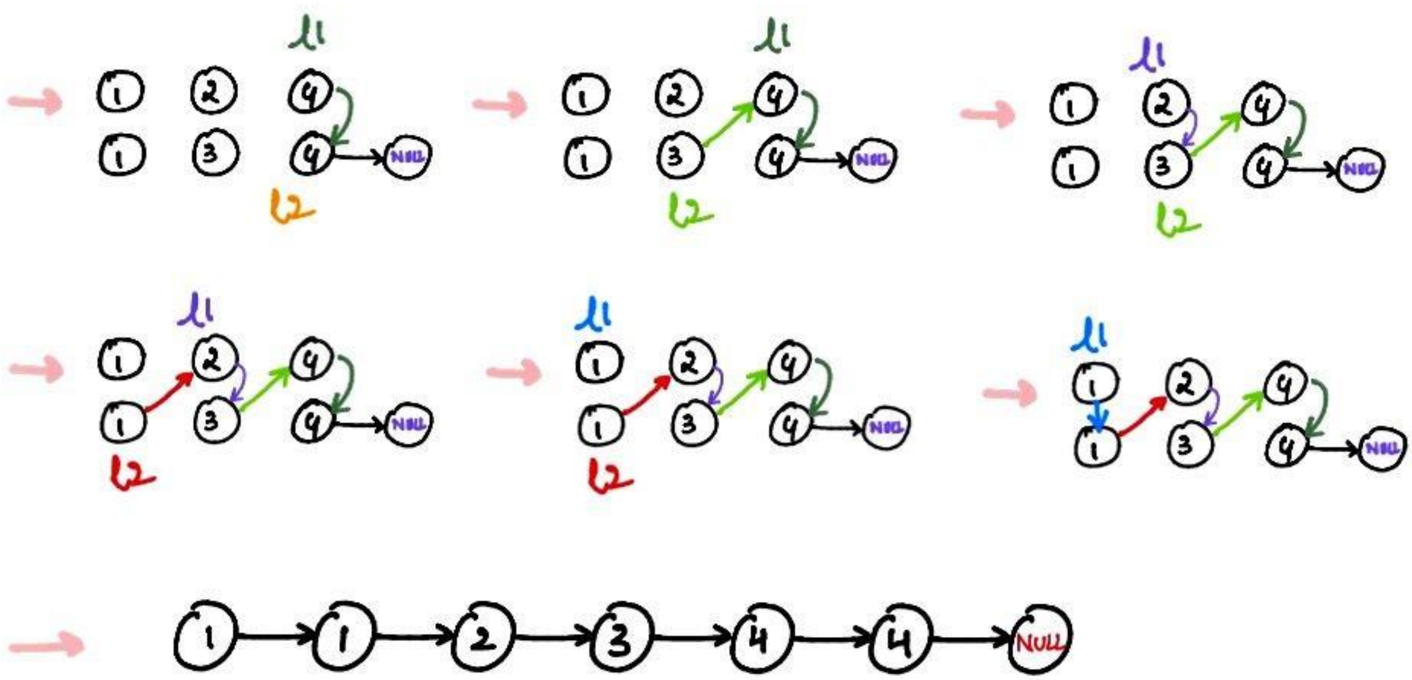
Recursive Code →

```cpp
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 l1 is null, return l2



(5) Add two Numbers

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

2 → 4 → 3

5 → 6 → 4

7 → 0 → 8

$$\begin{array}{r} \overset{1}{3}42 \\ +\,465 \\ \hline 807 \end{array}$$

TC → $O(m+n)$

SC → $O(\max(m,n))$

1) traverse both list simultaneously & if sum >= 10, then set carry = 1
2) add both values + carry
3) create newNode with this value

Initially,

→ 2 → 4 → 3

5 → 6 → 4

→ (-1) → 7

head

sum = 5+2 = 7 carry = 0

→ (-1) → 7 → 0

head

sum = 6 + 4 = 10

as sum ≥ 10, sum = sum % 10
= 10 % 10 = 0

carry = 1

→ (-1) → 7 → 0 → 8

head

sum = 3+4+1
= 8

carry = 0

code

```cpp
class Solution {
public:

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {

        ListNode* dummyNode;
        ListNode* head;
        dummyNode = head = new ListNode(-1);
        if(!l1)
            return l2;
        if(!l2)
            return l1;

        int carry = 0;

        while(l1 || l2){
            int firstVal = l1 ? l1->val : 0;
            int secondVal = l2 ? l2->val : 0;

            int total = firstVal + secondVal + carry;
            carry = total / 10;
            total = total % 10;

            ListNode* newNode = new ListNode(total);
            dummyNode->next = newNode;

            dummyNode = dummyNode->next;

            l1 = l1 ? l1->next : l1;
            l2 = l2 ? l2->next : l2;
        }

        if(carry)
            dummyNode->next = new ListNode(1);

        return head->next;
    }
};
```

6 Add two numbers II

→ Problem solving approach is same as previous problem

→ Points to note :

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

Eg →

7 → 2 → 4 → 3

5 → 6 → 4

1) 3 → 4 → 2 → 7

4 → 6 → 5

2) 7 → 0 → 8 → 7

3) 7 → 8 → 0 → 7

Result ↑

Code →

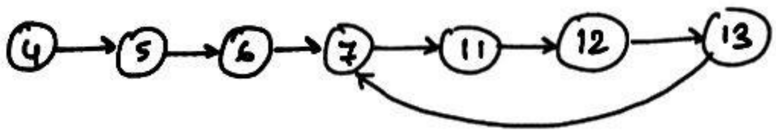
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* curr = head;
        ListNode* temp = NULL;
        while(curr!=NULL)
        {
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        l1=reverseList(l1); // O(n)
        l2=reverseList(l2); //O(n)
        ListNode* dummyNode;
        ListNode* head;
        dummyNode = head = new ListNode(-1);
        if(!l1)
            return l2;
        if(!l2)
            return l1;

        int carry = 0;

        while(l1 || l2){
            int firstVal = l1 ? l1->val : 0;
            int secondVal = l2 ? l2->val : 0;

            int total = firstVal + secondVal + carry;
            carry = total / 10;
            total = total % 10;

            ListNode* newNode = new ListNode(total);
            dummyNode->next = newNode;

            dummyNode = dummyNode->next;

            l1 = l1 ? l1->next : l1;
            l2 = l2 ? l2->next : l2;
        }

        if(carry)
            dummyNode->next = new ListNode(1);

        return reverseList(head->next); //o(max(m,n))
    }
};
```

(7) Linked List Cycle →

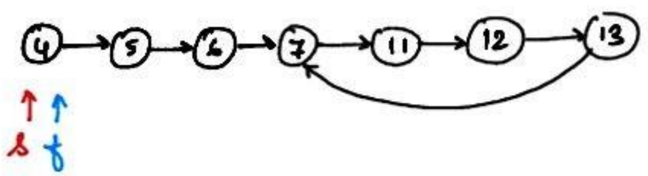
↗ TC → O(n)
SC → O(n)

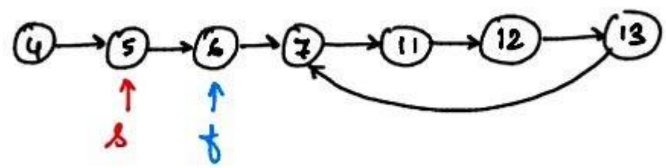
Approach - 1 → Create a set of nodes & insert every Node into it, if already exist then return True else false.

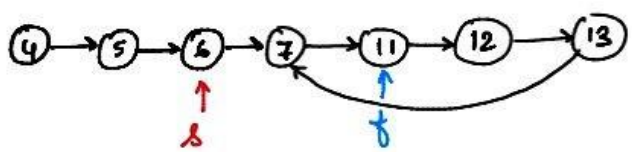
Approach - 2 → using fast & slow pointer TC → O(n) SC → O(1)

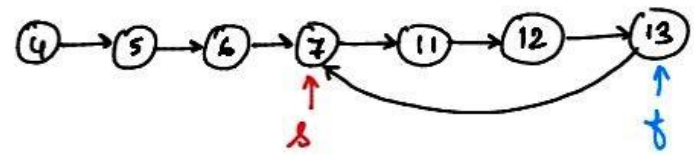
Eg 

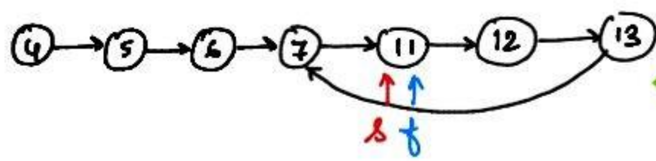
keep iterating till fast → next && fast → next → next exist.

→ 

→ 

→ 

→ 

→ 

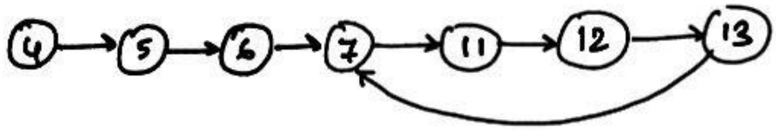
← return True

Code →

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head==NULL) return false;
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL && fast->next->next!=NULL)
        {
            fast=fast->next->next;
            slow=slow->next;
            if(fast==slow)  return true;
        }
        return false;
    }
};
```

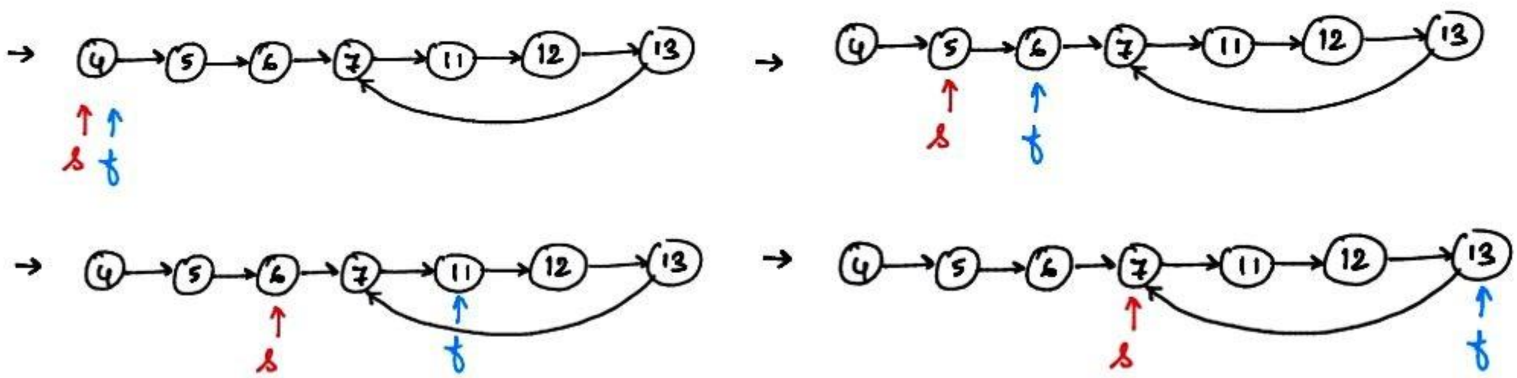
(8) Linked list cycle II → return the node where cycle begins.

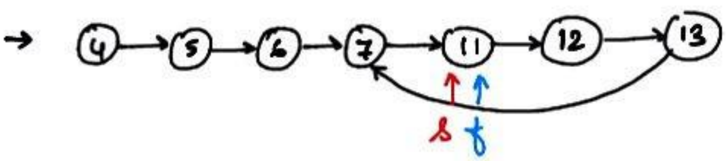
Eg



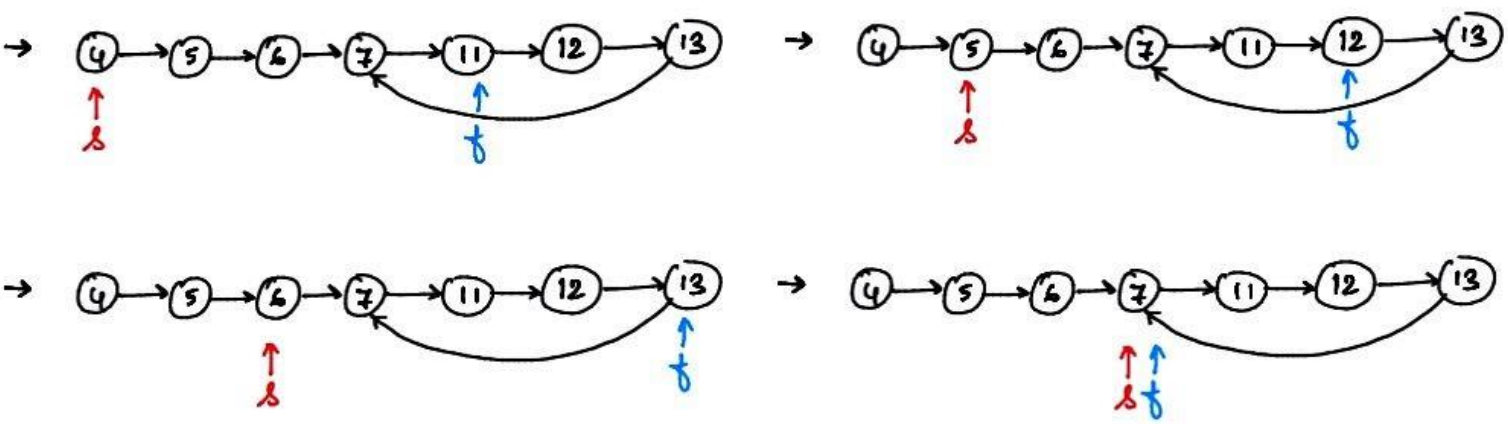
result = 7

Keep iterating while fast→next && fast→next→next exist.





← once slow == fast, then set slow = head, & move pointers by 1 unit



→ when slow == fast, it denotes the node where cycle begins.

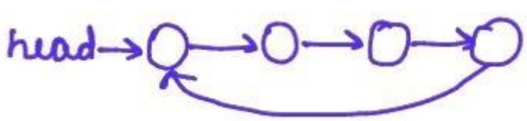
return slow; => 7

code →

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if(head==NULL) return NULL;
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL && fast->next->next!=NULL)
        {
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow)
            {
                slow = head;
                while (slow != fast)
                {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
        return NULL;
    }
};
```

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

* worst case when its a loop



$SC \rightarrow O(1)$

(9) Remove Nth node from End of list

2 → 4 → 5 → 7 → 11 → 13 → NULL

N = 2 ⇒ 2 → 4 → 5 → 7 → 13 → NULL

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

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

code →

TC → O(n)

SC → O(1)

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr = head, *temp;
        while(curr){
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *dummy = new ListNode(-1);
        dummy->next = reverseList(head);
        head = dummy;
        ListNode *curr = head;
        ListNode *prev = NULL;
        // Iteration
        for(int i=0; i<n; i++)
        {
            prev = curr;
            curr = curr->next;
        }
        // Deletion
        prev->next = curr->next;
        return reverseList(head->next);
    }
};
```

(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 it value by value.

code →

```cpp
class Solution {
public:
    ListNode* midNode(ListNode*head)
    {
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL and fast->next->next!=NULL){
            fast=fast->next->next;
            slow = slow->next;
        }
        return slow;
    }
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr = head, *temp;
        while(curr!=NULL)
        {
            temp=curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
    bool compare(ListNode*l1,ListNode* l2)
    {
        while(l1!=NULL && l2!=NULL)
        {
            if(l1->val!=l2->val)    return false;
            l1 = l1->next;
            l2 = l2->next;
        }
        return true;
    }
    bool isPalindrome(ListNode* head) {
        if(head==NULL) return false;
        if(head->next == NULL) return true;
        ListNode *mid = midNode(head);
        ListNode *l2 = mid->next;
        mid->next = NULL;
        l2 = reverseList(l2);
        return compare(head,l2);
    }
};
```

TC → O(n)

SC → O(1)

11) Remove duplicates from sorted list →

Given a linkedlist, return linkedlist without duplicates.

Eg (-2) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑head

⇒

any out of range value

1) (-101) → (-2) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑dummy ↑prev ↑curr ↑head

2) (-101) → (-2) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr (as values of prev & curr are equal)

3) (-101) → (-2) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr

prev = curr → next

4) (-101) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr

iterate till values of prev & curr are not equal

5) (-101) → (-2) → (1) → (2) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr

6) (-101) → (-2) → (1) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr

7) (-101) → (-2) → (1) → (2) → (2) → (4) → (5) → NULL

↑prev ↑curr

8) (-101) → (-2) → (1) → (2) → (4) → (5) → NULL

↑prev ↑curr

code →

```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* dummy = new ListNode(101);
        dummy->next = head;
        ListNode* curr = head;
        ListNode* prev = dummy;
        while(curr!=NULL)
        {
            if(curr->val==prev->val){
                prev->next = curr->next;
                curr = curr->next;
            } else {
                prev = curr;
                curr = curr->next;
            }
        }
        return dummy->next;
    }
};


// Another approach

ListNode* deleteDuplicates(ListNode* head) {
    if(head==NULL || head->next==NULL) return head;
    ListNode *curr = head;
    while(curr->next!=NULL){
        if(curr->val == curr->next->val){
            curr->next = curr->next->next;
        } else {
            curr = curr->next;
        }
    }
    return head;
}
```

Tc → $O(n)$

Sc → $O(1)$

(12) Swapping Nodes in Linked List →

Eg:

1 → 2 → 3 → 4 → 5

1 → 4 → 3 → 2 → 5 $\quad k=2$

given a linkedlist swap k^{th} node from both ends.

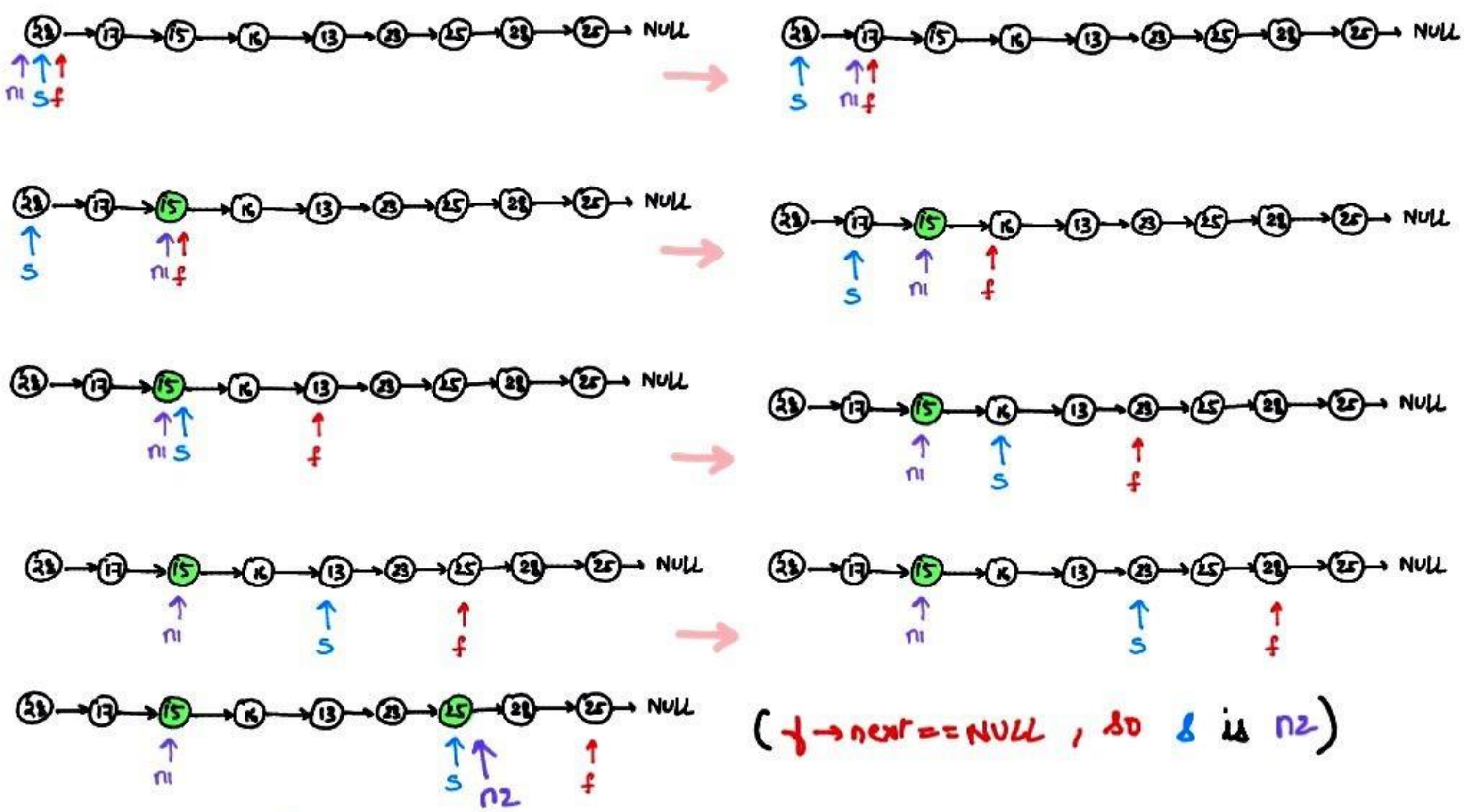
Eg

* for K-1 times iterate f & mark n1, then iterate s & f till f is not NULL, once null mark S as n2.

(as it is 1 indexed)

swap (n1, n2)

28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL $\quad k = 3$



(f → next == NULL, so s is n2)

swap (15, 25)

Result ⇒ 28 → 17 → 25 → 16 → 13 → 23 → 15 → 28 → 25 → NULL

code →

```cpp
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        ListNode *slow = head, *fast = head, *n1 = head;
        // finding n1
        for(int i=0; i<k-1; i++){
            fast = fast->next;
            n1 = fast;
        }
        // finding n2 (i.e slow)
        while(fast->next!=NULL){
            fast = fast->next;
            slow = slow->next;
        }
        // swapping
        int n1_val = n1->val;
        n1->val = slow->val;
        slow->val = n1_val;
        return head;
    }
};
```

TC → O(n)

SC → O(1)

(13) Odd Even Linked List →

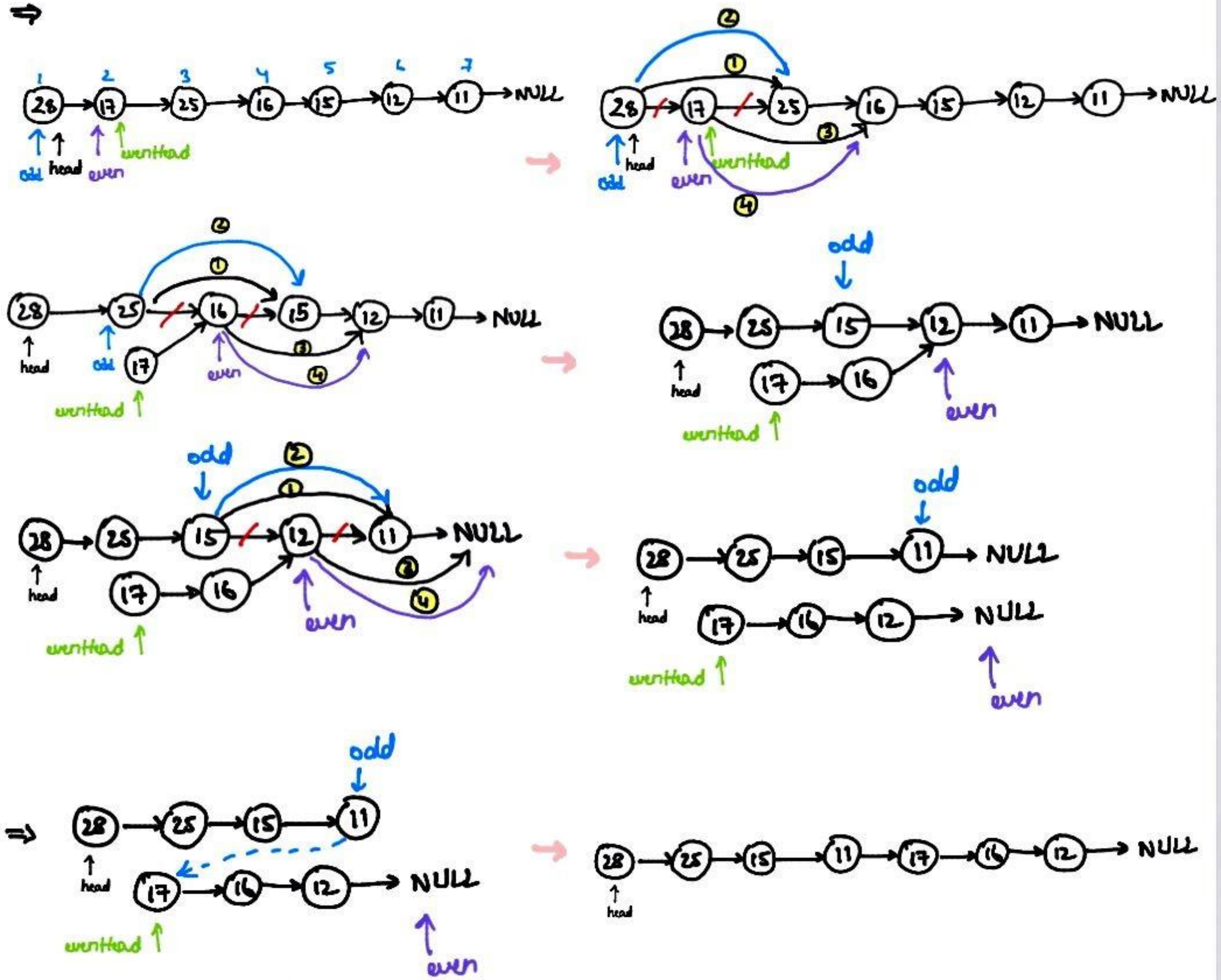
given a linkedlist group all odd indices nodes followed by even nodes

Eg

28 → 17 → 25 → 16 → 15 → 12 → 11 → NULL
1   2   3   4   5   6   7

28 → 25 → 15 → 11 → 17 → 16 → 12 → NULL

⇒



odd → next = evenHead

code →

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head) return NULL;

        ListNode *even =  head->next;
        ListNode *odd = head;
        ListNode *evenHead = even;

        while(even && even->next){
            odd->next=even->next;
            odd=odd->next;
            even->next=odd->next;
            even=even->next;
        }

        // like odd and even lists
        odd->next = evenHead;
        return head;
    }
};
```

TC → O(n)
SC → O(1)

(14) Swap Nodes in Pairs → given a linkedlist swap adjacent nodes.

Eg

Even length

28 → 17 → 15 → 15 → 86 → 14 → 63 → 22 → NULL

Ans ⇒ 17 → 28 → 15 → 15 → 14 → 86 → 22 → 63 → NULL

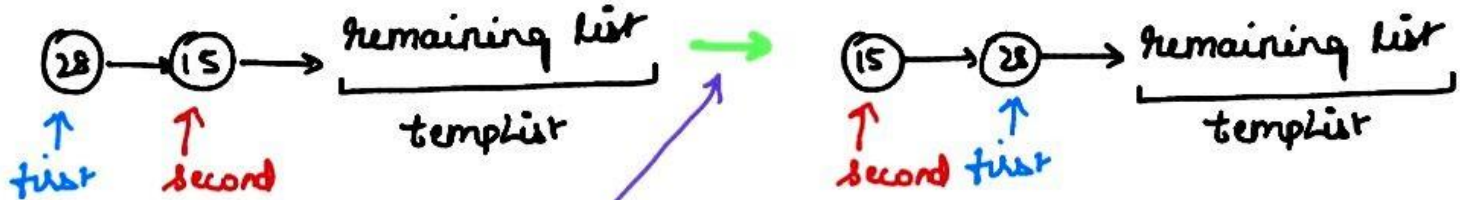
Odd length

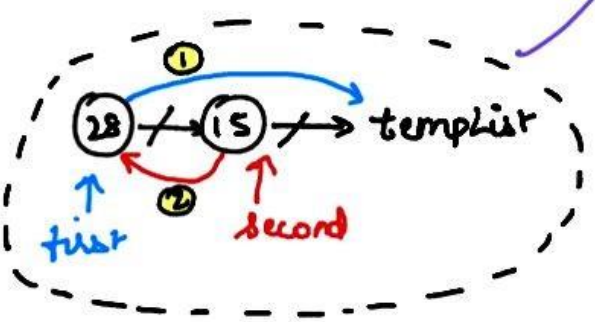
28 → 17 → 15 → 15 → 86 → 14 → 63 → NULL

Ans ⇒ 17 → 28 → 15 → 15 → 14 → 86 → 63 → NULL

Eg → 28 → 15 → 63 → 42 → 78 → 96 → 15 → 68 → NULL

Consider for 1st pair,





Templist = second → next
first → next = templist
second → next = first

} Logic

Solve recursively for all pairs.

Code →

```
class Solution {
public:
    ListNode* SwapAdjacentNodes(ListNode* head)
    {
        if(head==NULL || head->next==NULL)  return head;
        ListNode *first = head;
        ListNode *second = head->next;
        // start logic
        ListNode *tempList = SwapAdjacentNodes(second->next);
        first->next = temp;
        second->next = first;
        return second;
    }
    ListNode* swapPairs(ListNode* head) {
        return SwapAdjacentNodes(head);
    }
};
```

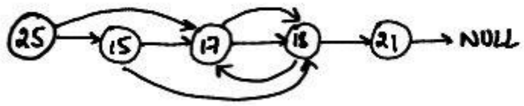
TC → O(N)

SC → O(1)

Recursive stack → O(N/2) ≃ O(N)

(15) Copy list with random pointer → given a list, clone & return.

Eg 1.



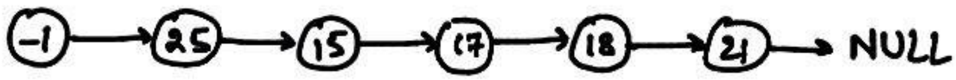
mp

Old	New
25	25
15	15
17	17
18	18
21	21
NULL	NULL

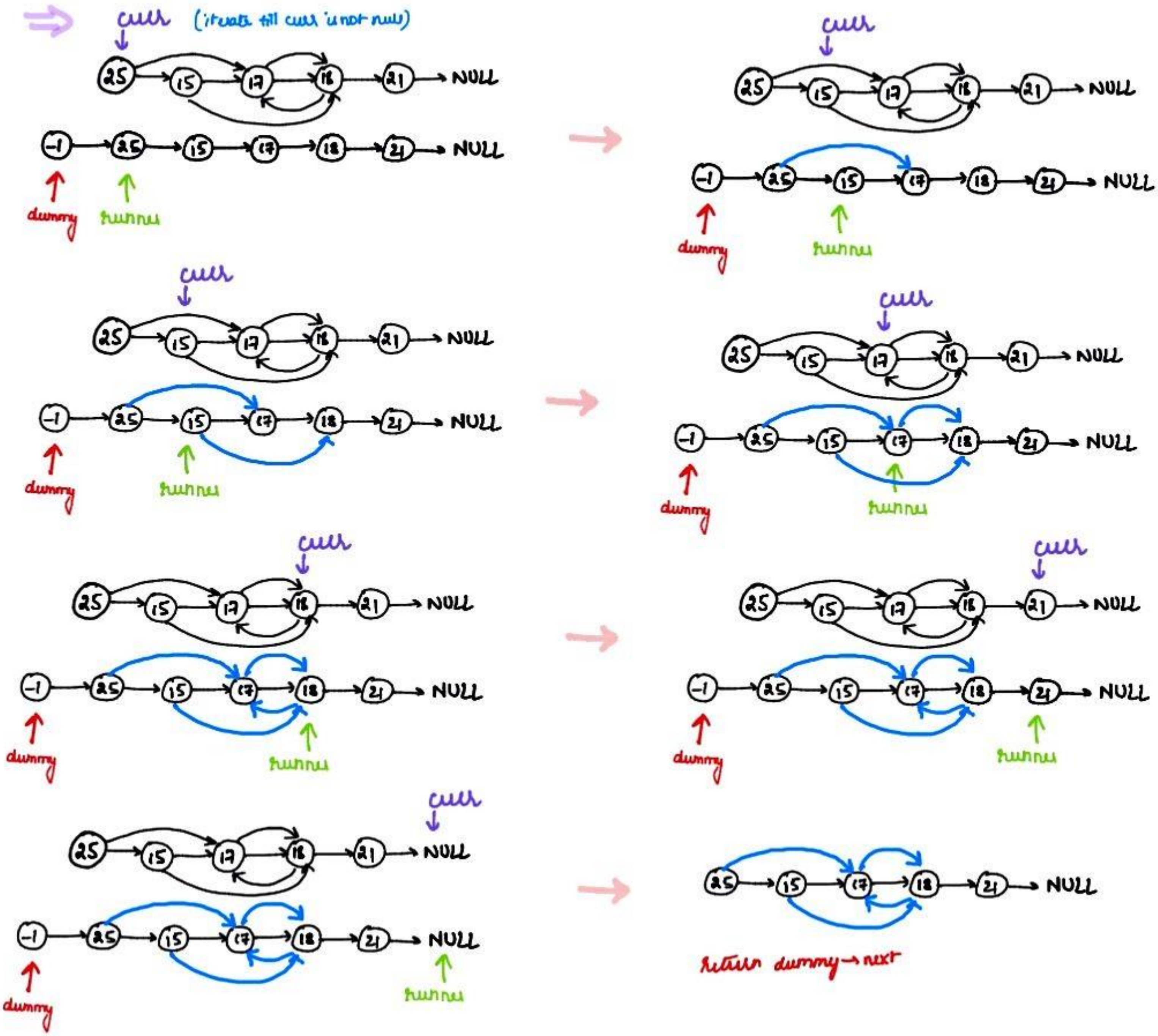
-> In 1st iteration create the list without random pointers & also maintain hashmap for mapping nodes pointed by random pointers

-> In 2nd iteration use map to link nodes pointed by random pointers

After 1st iteration →



⇒



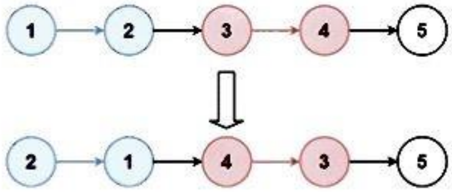
Code →

TC → O(n)

SC → O(n)

```cpp
class Solution {
public:
    Node* copyRandomList(Node* head) {

        unordered_map<Node*, Node*> mp;
        Node *dummy = new Node(100001);
        Node *runner = dummy, *curr = head;

        // initial iteration
        while(curr != NULL){
            Node *newNode = new Node(curr->val);
            runner->next = newNode;
            mp[curr] = newNode;
            curr = curr->next;
            runner = runner->next;
        }

        // setting starting points in both lists
        curr = head;
        runner = dummy->next;

        // setting the random pointers
        while(curr != NULL){
            if(curr->random != NULL)
                runner->random = mp[curr->random];
            runner = runner->next;
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

(16) Reverse Nodes in K-Group



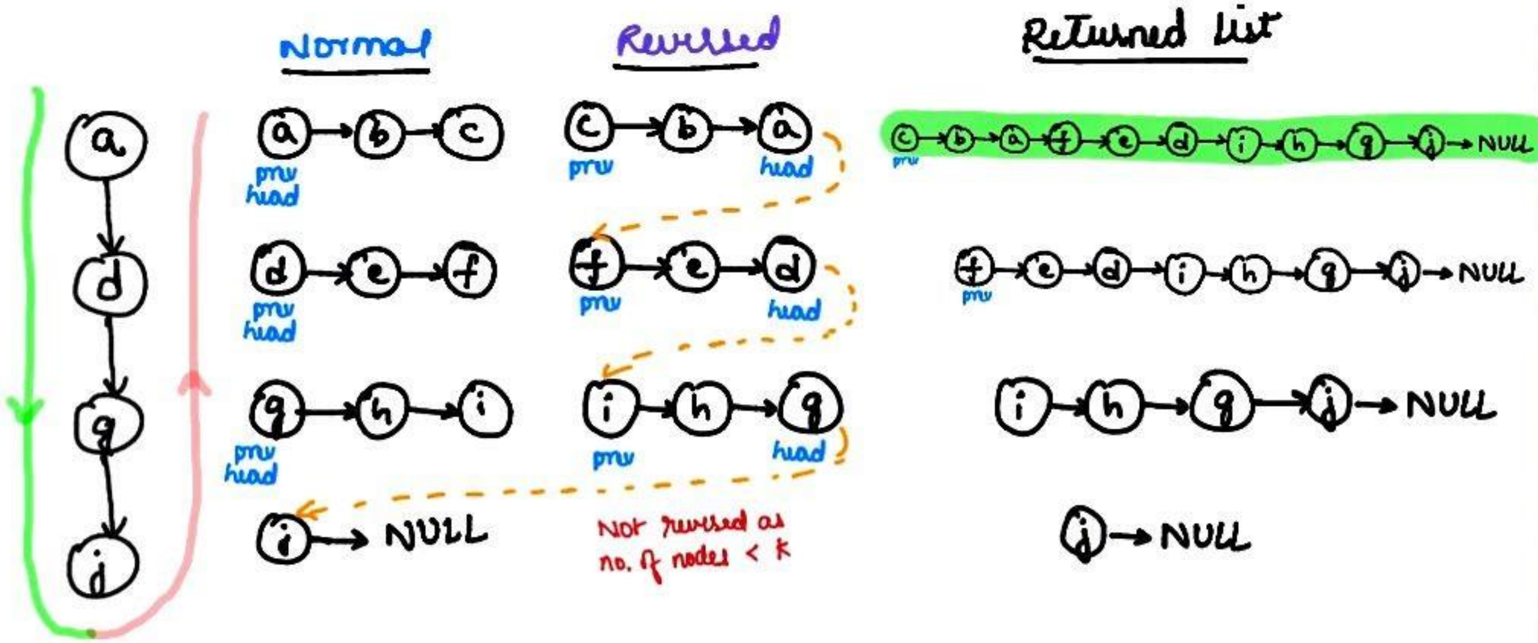
↳ K = 2

given a linkedlist & K, return a list with reversed nodes by K-groups.

Eg. a → b → c → d → e → f → g → h → i → j → NULL

k = 3

Res ⇒ c → b → a → f → e → d → i → h → g → j → NULL



* consider the case of f → e → d (prev, head), then tempList = i → h → g

linking would happen as head → next = tempList & this list would become tempList to c → b → a

code →

TC → O(n)
SC → O(1)

```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head)
    {
        ListNode *prev = NULL, *curr = head, *temp;
        while (curr!=NULL)
        {
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* reverseInGroups(ListNode* head, int k)
    {
        ListNode *curr = head;
        int currlen = 1;
        if(head == NULL) return head;
        while(curr->next!=NULL && currlen<k ){
            curr=curr->next;
            currlen+=1;
        }
        if(currlen<k)   return head;
        ListNode *tempNode = curr->next;
        curr->next = NULL;

        // start linking
        ListNode *tempList = reverseInGroups(tempNode,k);
        ListNode *prev = reverseList(head);
        head->next = tempList;
        return prev;
    }

    ListNode* reverseKGroup(ListNode* head, int k) {
        return reverseInGroups(head,k);
    }
};
```

(17) Design LinkedList → Implementation of Doubly LinkedList

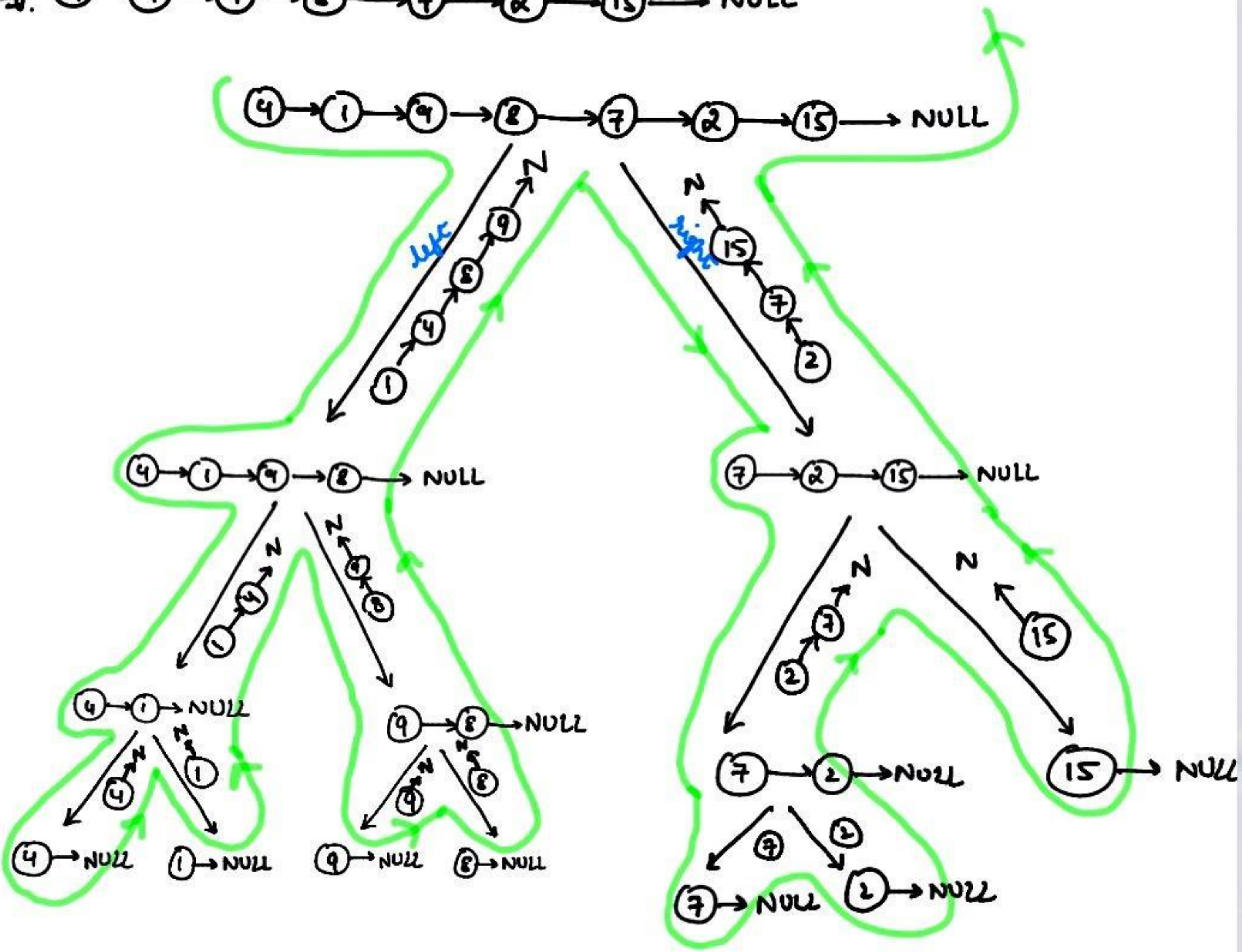
Code →

```cpp
class Node{
    public:
    int val;
    Node* prev;
    Node* next;
    Node(int val){
        this->val=val;
        prev = nullptr;
        next = nullptr;
    }
};
class MyLinkedList {

public:
    Node *head;
    Node *tail;
    MyLinkedList(){
        head = nullptr;
        tail = nullptr;
    }

    int get(int index){
        if(head == NULL)     return -1;
        Node *temp = head;
        int count = 0;
        while(temp!=NULL){
            temp=temp->next;
            count++;
        }
        if(index>=count)     return -1;
        temp = head;
        while(temp != NULL && index>0){
            temp=temp->next;
            index--;
        }
        return temp->val;
    }

    void addAtHead(int val){
        Node *newNode = new Node(val);
        if(head == NULL){
            head = newNode;
            tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }

    void addAtTail(int val){
        Node *temp = head;
        if(head == NULL){
            Node *newNode = new Node(val);
            head = newNode;
            tail = newNode;
            return;
        }
        while(temp->next!=NULL){
            temp = temp->next;
        }
        Node *newNode = new Node(val);
        temp->next = newNode;
        newNode->prev = temp;
        tail = newNode;
    }
```

```cpp
void addAtIndex(int index, int val){
        Node *temp = head;
        int count = 0;
        while(temp != NULL){
            temp = temp->next;
            count++;
        }
        if(index>count) return ;
        if(index==0){
            addAtHead(val);
            return;
        } else if(count == index){
            addAtTail(val);
            return;
        } else {
            temp = head;
            while(temp != NULL && index>0){
                temp = temp->next;
                index--;
            }
            Node* newNode = new Node(val);
            Node* temp2 = temp->prev;
            temp->prev->next = newNode;
            temp->prev = newNode;
            newNode->prev = temp2;
            newNode->next = temp;
        }
    }

     void deleteAtIndex(int index) {
        Node* temp = head;
        int count = 0;
        while(temp != NULL){
            temp=temp->next;
            count++;
        }
        if(index>=count) return;
        if(count==1 && index==0){
            head = NULL;
            return;
        } else if(count-1 == index){
            tail = tail->prev;
            tail->next = NULL;
            return;
        } else {
            if(index==0){
                head->next->prev = NULL;
                head = head->next;
                return;
            }
            temp=head;
            while(temp!=NULL && index>0){
                temp = temp->next;
                index--;
            }
            Node* temp2 = temp->next;
            temp->prev->next = temp2;
            temp->next->prev = temp->prev;
        }
    }
};
```

(18) Sort List → By following Merge Sort.

Eg. 4 → 1 → 9 → 8 → 7 → 2 → 15 → NULL



In the last step while returning from both branches we have,

left = 1 → 4 → 8 → 9 → NULL & right = 2 → 7 → 15 → NULL

so create dummy node & merge, i.e. -1 → 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

return dummy → next, 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

* The same happens at every intermediate merge.

code →

Tc → O(m+n)

Sc → O(n)

```cpp
class Solution {
public:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1);
        ListNode *curr = dummy;
        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 = l1;
        if(l2) curr->next = l2;

        return dummy->next;
    }

    ListNode* sortList(ListNode* head) {
        if(!head || !head->next) return head;

        ListNode *slow = head;
        ListNode *fast = head->next;
        while(fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        // dividing the lists into 2 parts
        fast = slow->next;
        slow->next = NULL;

        // sort & merge
        head = sortList(head);
        fast = sortList(fast)
        return merge(head, fast);
    }
};
```

Recursion & Backtracking

- Karun Karthik

Contents →

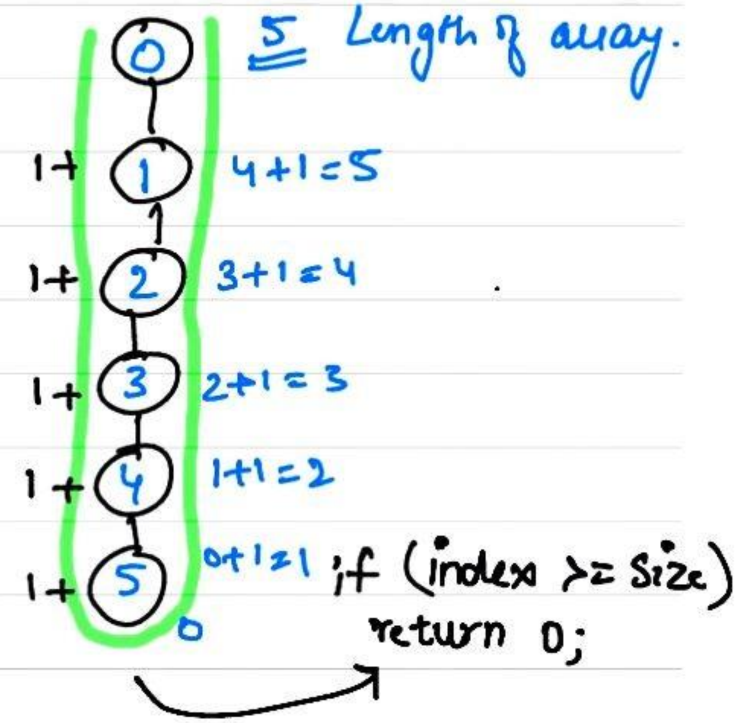
Recursion

(0) Length of an array

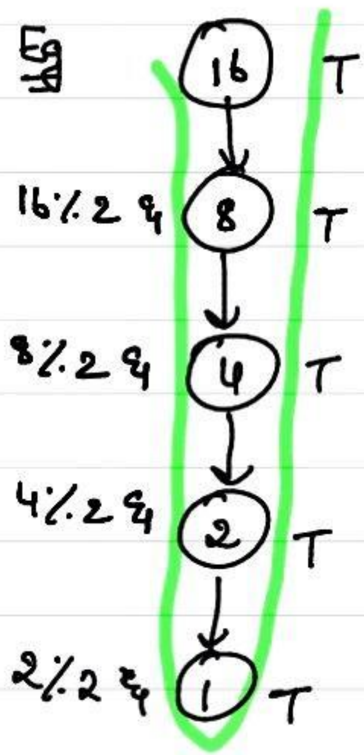
[20, 10, 40, 50, 30]
 0   1   2   3   4

$TC = O(n)$

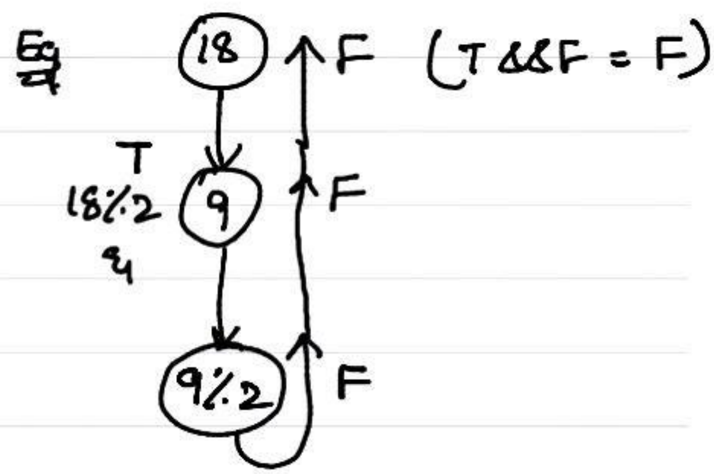
$SC = O(n)$.



(1) Power of 2 → $2^x = 2^0 \cdot 2^1 \cdot 2^2 \cdots 2^n$.



if 1 then
return true.



$TC = O(\log_2 n)$

Power of 2

C++

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

2 Power of 3

C++

```cpp
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++

```cpp
class Solution {
public:
    bool isPowerOfFour(int n) {
        if(n==1) return true; //need to write it first else it might
        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 nums = [1,2,3]

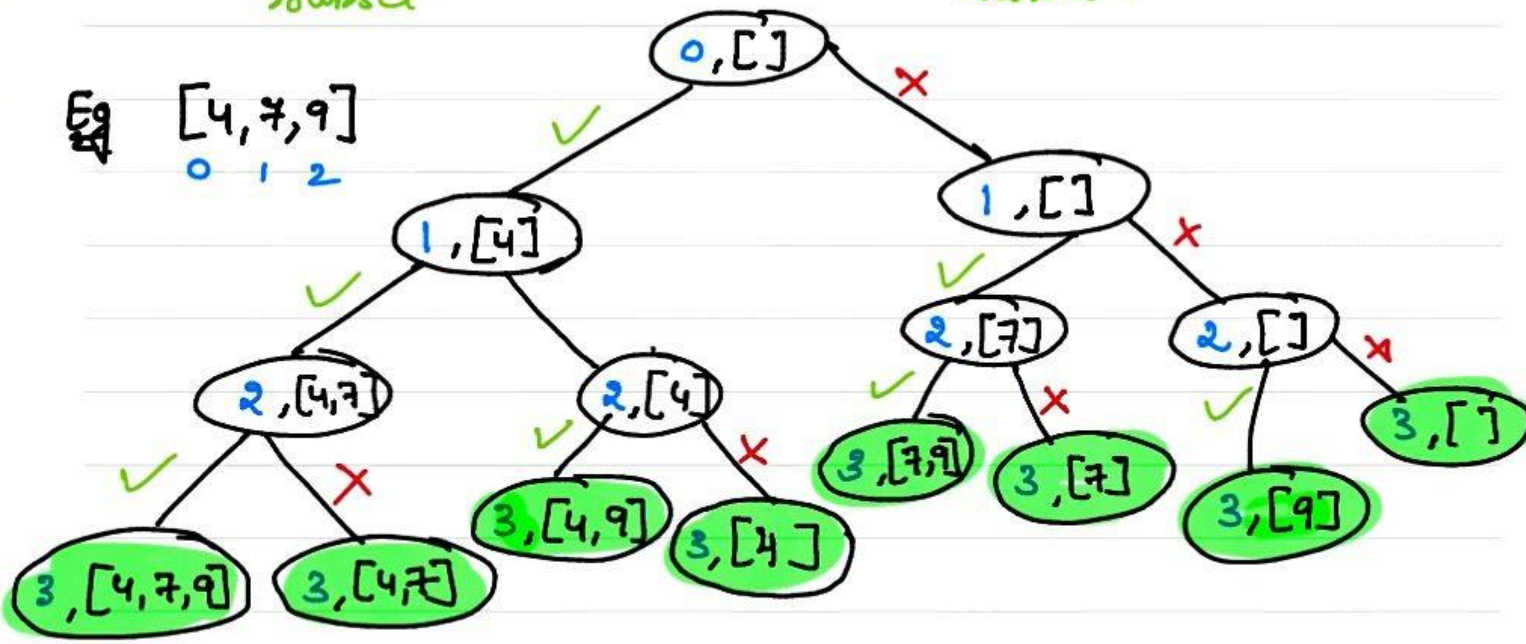
Sol = [[ ],[1],[1,2],[1,3],[1,2,3],[2],[2,3],[3]]

For every, element

can be a part of subset

cannot be a part of subset.

Eg [4,7,9]
0 1 2



⇒ [[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 = $O(2^n)$ → as there are 2 possibilities at every element.

SC = $O(2^n)$

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 :- nums = [2,3,5] target = 8

0 1 2

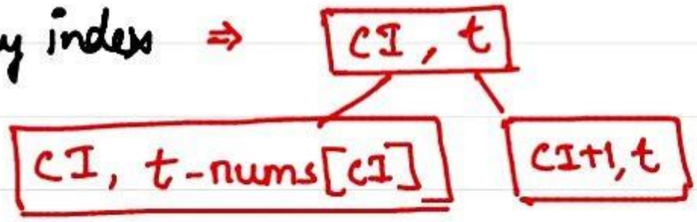
Sol. [[2,2,2,2], [2,3,3], [3,5]]

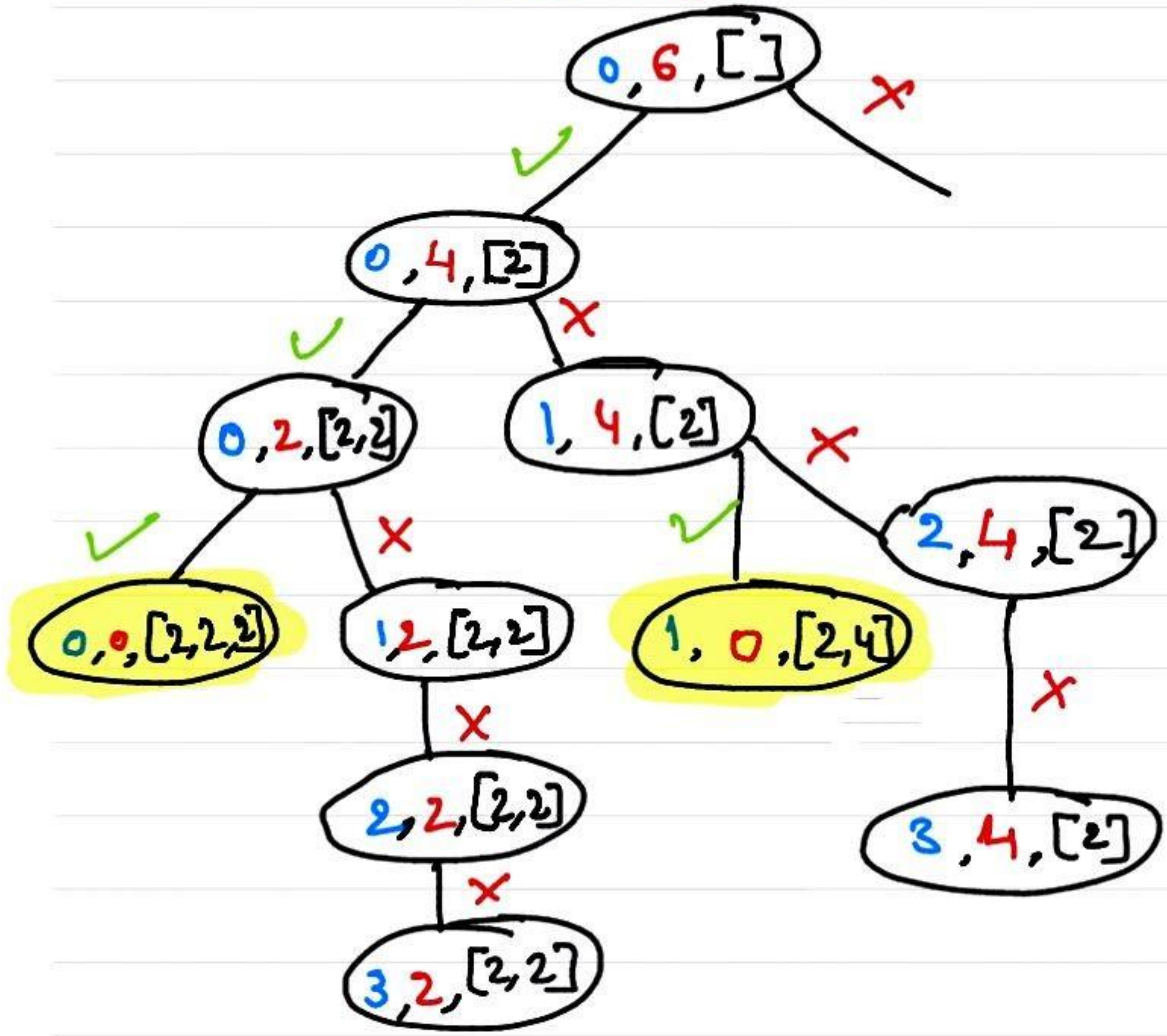
Eg [2,4,5]

0,1,2

target = 6

For every index ⇒





⇒ [[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,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 (r,c) we can move in D, L, R, U

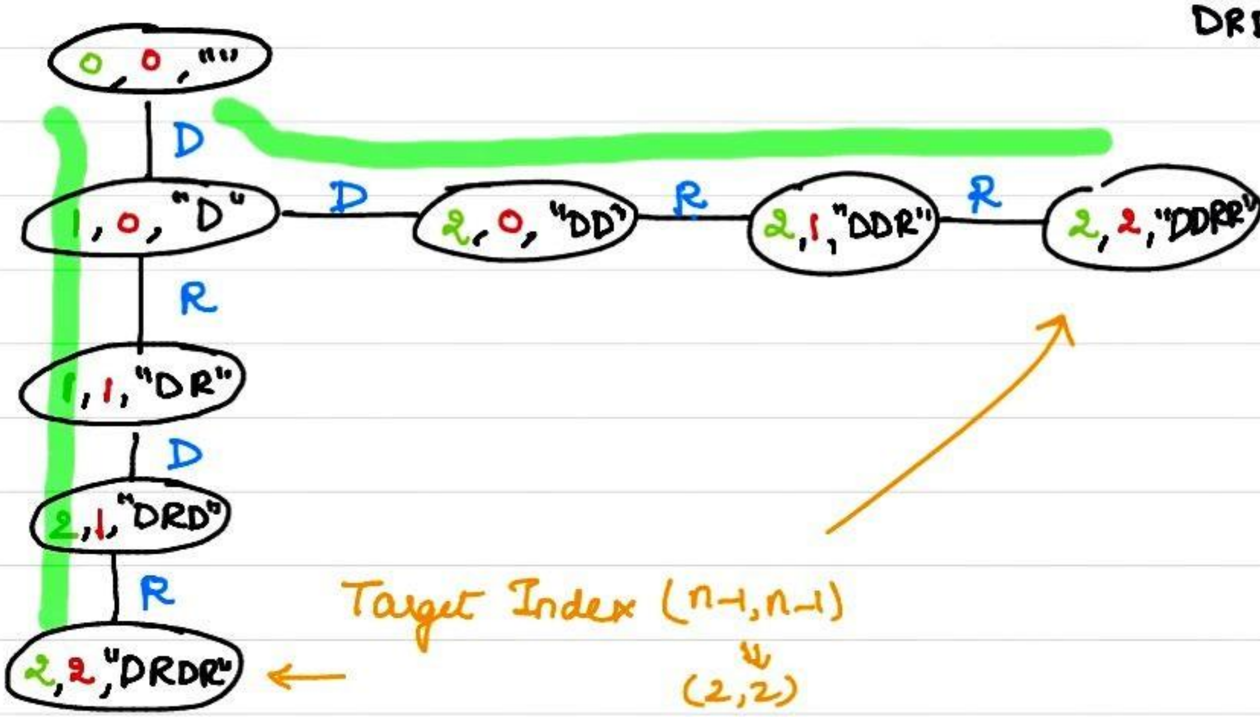
D → (r+1,c), L → (r,c-1), R → (r,c+1), U → (r-1,c)

Eg

$$n = 3 \quad \begin{array}{c|c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \end{array} \Rightarrow \text{DRDR, DDRR}$$

N = 4 {{1,0,0,0},
{1,1,0,1},
{1,1,0,0},
{0,1,1,1}}

[DDRDRR,
DRDDRR]



* Before making any call from cell change its state

* while returning, UNDO the changes made (Backtracking.)

Code →

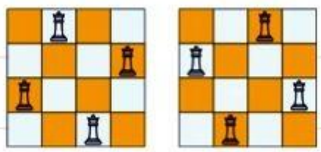
```
class Solution{
    public:
    void allPaths(int row, int col, int n, vector<vector<int>>&m, string ans, vector<string>&res){

        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;
    }
};
```

D4 N-queens

⑦ If given n, then we should place n-queens in N×N matrix, such that no 2 queens → share same row, column, diagonal

↳ return all configurations

Eg $n = 4$



Initially
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\cdot,\cdot,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\cdot] \\ 2 & [\cdot,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

X_R → Bad Row
X_C → Bad column
X_D → Bad diagonal
X_N → Not possible

Step → Start from (0, 0)

①
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\varphi,\cdot,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\cdot] \\ 2 & [\cdot,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

column pos

0	1	2	3
✓			
X_R	X_D	✓	X_D

②
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\varphi,\cdot,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\varphi,\cdot] \\ 2 & [\cdot,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

column pos

0	1	2	3
✓			
X_R	X_D	✓	X_D
X_R	X_D	X_C	X_D

↳ this says (1,2) is a bad config & so is (0,0) ∴ Backtrack.

③
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\cdot,\varphi,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\cdot] \\ 2 & [\cdot,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

column pos

0	1	2	3
X_N	✓		
X_D	X_C	X_D	✓

④
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\cdot,\varphi,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\varphi] \\ 2 & [\cdot,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

column pos

0	1	2	3
X_N	✓		
X_D	X_C	X_D	✓
✓	X_C	X_D	X_C

⑤
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\cdot,\varphi,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\varphi] \\ 2 & [\varphi,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\cdot,\cdot] \end{array}$$

column pos

0	1	2	3
X_N	✓		
X_D	X_C	X_D	✓
✓	X_C	X_D	X_C
X_C	X_C	✓	X_C

⑥
$$\begin{array}{c|c} & 0\;\;1\;\;2\;\;3 \\ 0 & [\cdot,\varphi,\cdot,\cdot] \\ 1 & [\cdot,\cdot,\cdot,\varphi] \\ 2 & [\varphi,\cdot,\cdot,\cdot] \\ 3 & [\cdot,\cdot,\varphi,\cdot] \end{array}$$

column pos

0	1	2	3
X_N	✓		
X_D	X_C	X_D	✓
✓	X_C	X_D	X_C
X_C	X_C	✓	X_C

↑ final result ∴ store & backtrack for other config.

code →

```cpp
class Solution {
public:

    bool valid_row(int curr_row, vector<vector<char>>&grid, int n){
        for(int i = 0; i < n; i++){
            if(grid[curr_row][i]=='Q')
                return false;
        }
        return true;
    }

    bool valid_col(int curr_col, vector<vector<char>>&grid, int n){
        for(int i = 0; i < n; i++){
            if(grid[i][curr_col]=='Q')
                return false;
        }
        return true;
    }

    bool valid_diagonal(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
        int i = curr_row;
        int j = curr_col;
        while(i>=0 && j>=0){        // Top-left diagonal
            if(grid[i][j]=='Q')
                return false;
            i--; j--;
        }

        i = curr_row;
        j = curr_col;
        while(i>=0 && j<n){        // Top-right diagonal
            if(grid[i][j]=='Q')
                return false;
            i--; j++;
        }

        i = curr_row;
        j = curr_col;
        while(i<n && j>=0){        // Bottom-left diagonal
            if(grid[i][j]=='Q')
                return false;
            i++; j--;
        }

        i = curr_row;
        j = curr_col;
        while(i<n && j<n){        // Bottom-right diagonal
            if(grid[i][j]=='Q')
                return false;
            i++; j++;
        }

        return true;
    }

```

```
    bool isValid(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
        return valid_row(curr_row, grid, n) && valid_col(curr_col, grid, n) && valid_diagonal(grid, curr_row, curr_col, n);
    }

    // Function to convert grid char to strings
    vector<string> populate(vector<vector<char>>&grid, int n){
        vector<string> result;
        for(int i = 0; i<n; i++){
            string temp = "";
            for(int j=0; j<n; j++){
                temp += grid[i][j];
            }
            result.push_back(temp);
        }
        return result;
    }

    void solve(vector<vector<char>>&grid, int curr_row, int n, vector<vector<string>>&ans){
        if(curr_row==n){
            vector<string> temp = populate(grid,n);
            ans.push_back(temp);
            return;
        }
        for(int curr_col=0; curr_col < n; curr_col++){
            if(isValid(grid, curr_row, curr_col,n)){
                grid[curr_row][curr_col] = 'Q';
                solve(grid, curr_row+1, n, ans);
                grid[curr_row][curr_col] = '.';
            }
        }
    }

    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> ans;
        vector<vector<char>>grid(n, vector<char>(n,'.'));
        solve(grid, 0, n, ans);
        return ans;
    }
};
```

(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, 3x3 sub-grid:

(a) grid(r, c) = i

(b) recursively fill remaining empty cells.

(c) if recursion is successful:
return true

(d) grid(r,c) = '.' (backtracking)

(3) return false

Code

```cpp
class Solution {
public:
    bool valid_row(vector<vector<char>>&board, int currRow, int currVal){
        for(int i=0; i<9; i++){
            if(board[currRow][i]==currVal+'0'){
                return false;
            }
        }
        return true;
    }

    bool valid_col(vector<vector<char>>&board, int currCol, int currVal){
        for(int i=0; i<9; i++){
            if(board[i][currCol]==currVal+'0'){
                return false;
            }
        }
        return true;
    }

    bool valid_grid(vector<vector<char>>&board, int currRow, int currCol, int currVal){
        int x = 3*(currRow/3);
        int y = 3*(currCol/3);
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                if(board[x+i][y+j]== currVal+'0'){
                    return false;
                }
            }
        }
        return true;
    }

    bool isValidCell(vector<vector<char>>&board, int currRow, int currCol, int currVal){
        return valid_row(board, currRow, currVal) && valid_col(board, currCol, currVal) &&
        valid_grid(board, currRow, currCol, currVal);
    }


```

```cpp

    bool sudokuSolver(vector<vector<char>>&board, int currRow, int currCol){
        if(currRow==9)
            return true;

        int nextRow = 0;
        int nextCol = 0;

        // find next possible row n column
        if(currCol==8){
            nextRow = currRow+1;
            nextCol = 0;
        } else {
            nextRow = currRow;
            nextCol = currCol+1;
        }

        // if not filled then call
        if(board[currRow][currCol]!='.'){
            return sudokuSolver(board, nextRow, nextCol);
        }

        // try all possibilities from 1 to 9 numbers
        for(int currVal=1; currVal<10; currVal++){

            // if valid then make the change
            if(isValidCell(board, currRow, currCol, currVal)){
                board[currRow][currCol] = '0'+currVal;

                // if already solved then return true directly
                if(sudokuSolver(board, nextRow, nextCol)==true)
                    return true;

                // backtracking
                board[currRow][currCol] = '.';
            }
        }

        return false;
    }
    void solveSudoku(vector<vector<char>>& board) {
        sudokuSolver(board, 0, 0);
    }
};
```

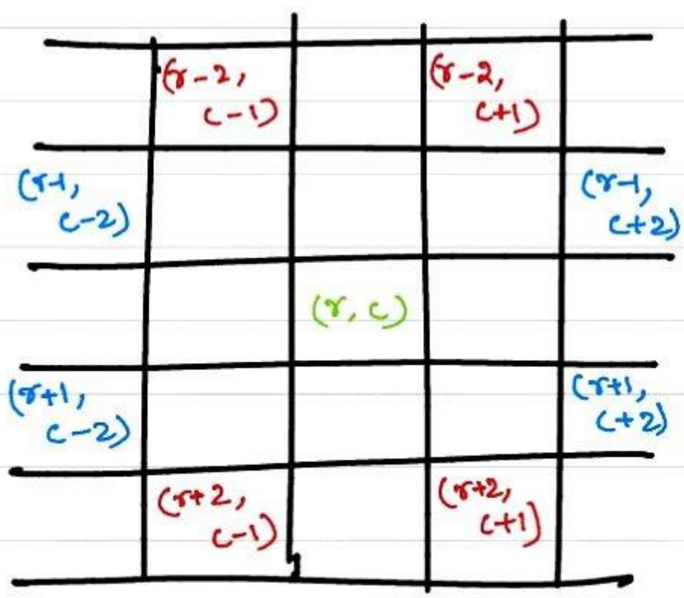
D6 Knight's tour problem.

(9) Given an n*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 →

```cpp
#include <bits/stdc++.h>
using namespace std;

void display(vector<vector<int>>&grid){
    for(auto i: grid){
        for(auto j:i){
            cout<<j<<" ";
        }
        cout<<"\n";
    }
}

void KnightTour(vector<vector<int>> &grid, int currRow, int currCol,
                                                    int upcomingVal, int n){
    if(upcomingVal==n*n){
        display(grid);
        cout<<"\n";
        return;
    }

    if(currRow<0 || currRow>=n || currCol<0 || currCol>=n
                              || grid[currRow][currCol]!=0){
        return;
    }

    grid[currRow][currCol] = upcomingVal;

    KnightTour(grid, currRow-2, currCol-1, upcomingVal+1, n);
    KnightTour(grid, currRow-2, currCol+1, upcomingVal+1, n);
    KnightTour(grid, currRow+2, currCol-1, upcomingVal+1, n);
    KnightTour(grid, currRow+2, currCol+1, upcomingVal+1, n);
    KnightTour(grid, currRow-1, currCol-2, upcomingVal+1, n);
    KnightTour(grid, currRow-1, currCol+2, upcomingVal+1, n);
    KnightTour(grid, currRow+1, currCol-2, upcomingVal+1, n);
    KnightTour(grid, currRow+1, currCol+2, upcomingVal+1, n);

    grid[currRow][currCol] = 0;
    return;
}

int main() {
    int n;
    cin>>n;
    vector<vector<int>>grid(n,vector<int>(n,0));
    KnightTour(grid, 0, 0, 1, n);
    return 0;
}
```

(10) Letter combination of a phone number

Eg digits = "23" → abc (0 1 2), def (0 1 2)

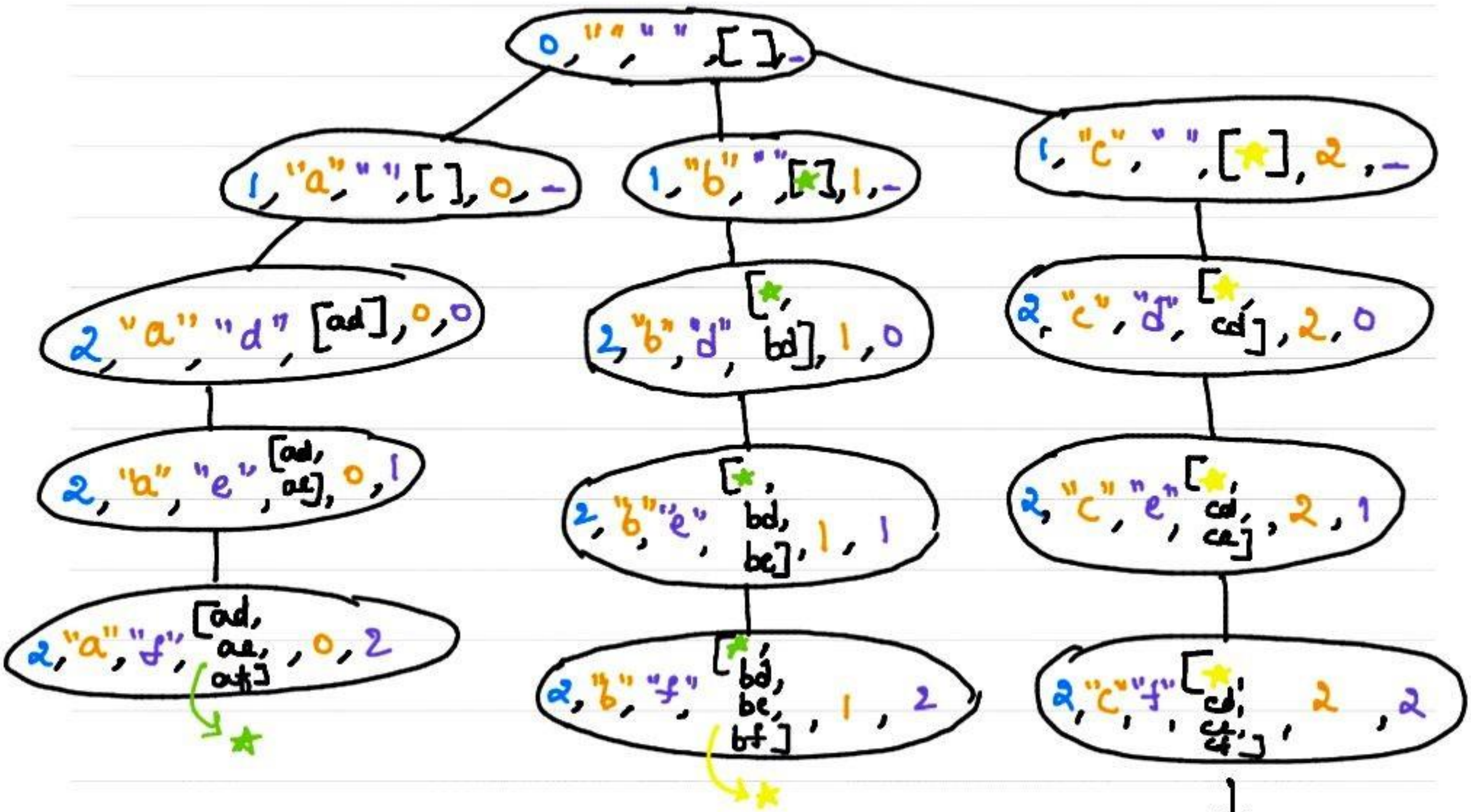
digits indexes: 0 1

"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

//



final result.

[ad, ae, af, bd, be, bf, cd, ce, cf]

Code →

```cpp
class Solution {
public:
    void findAll( map<char,string> &mapper, string digits,
    vector<string> &ans, string &s, int currentIndex){

        if(currentIndex>=digits.length()){
            ans.push_back(s);
            return;
        }

        char currNum = digits[currentIndex];
        string alpha = mapper[currNum];

        for(int i=0; i<alpha.size(); i++){
            s.push_back(alpha[i]);
            findAll(mapper, digits, ans, s, currentIndex+1);
            s.pop_back();
        }
        return;
    }

    vector<string> letterCombinations(string digits) {

        map<char,string>mapper{
            {'1', ""},
            {'2', "abc"},
            {'3', "def"},
            {'4', "ghi"},
            {'5', "jkl"},
            {'6', "mno"},
            {'7', "pqrs"},
            {'8', "tuv"},
            {'9', "wxyz"},
          };
        string s = "";
        vector<string> ans;

        // edge case
        if(digits.size()==0){
            return ans;
        }
        // else generate all possibilities
        findAll(mapper, digits, ans, s, 0);
        return ans;

    }
};
```

(11) Subsets II → Same as subsets but no duplicates.

(1) using set<int>

Code →

```cpp
class Solution {
public:
    void allsubs(vector<int>& nums,int curr,
    vector<int>&ds,set<vector<int>>&ans)
    {
        if(curr>=nums.size()){
            ans.insert(ds);
           return;
        }
        int currval = nums[curr];
        ds.push_back(currval);
        allsubs(nums,curr+1,ds,ans);

        // removing currentVal (not considering)
        ds.pop_back();
        allsubs(nums,curr+1,ds,ans);
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        set<vector<int>>ans;
        vector<int>vec;
        sort(nums.begin(),nums.end());
        allsubs(nums,0,vec,ans);
        vector<vector<int>> res{ans.begin(), ans.end()};
        return res;
    }
};
```

② without using sets

code →

```
class Solution {
public:
    void allsubs(vector<int> &nums, int curr, vector<int> &ds,
                                        vector<vector<int>>& res){
        res.push_back(ds); // storing initial answers
        for(int i=curr; i<nums.size(); i++){
            if(i>curr && nums[i]==nums[i-1]) continue; // avoiding duplicates
            ds.push_back(nums[i]);
            allsubs(nums, i+1, ds, res);
            ds.pop_back();
        }
        return;
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<vector<int>> res;
        vector<int> ds;
        sort(nums.begin(), nums.end());
        allsubs(nums, 0, ds, res);
        return res;
    }
};
```

(12) Combinational Sum - II

→ Same as combinational sum but no duplicates

Code →

```
class Solution {
public:
    void findAll(vector<int>&candidates, int target, int idx,
                    vector<vector<int>> &ans, vector<int> &ds){

        if(target==0){
            ans.push_back(ds);
            return;
        }

        for(int i = idx; i<candidates.size(); i++){

            // avoid duplicates
            if(i>idx && candidates[i]==candidates[i-1]) continue;

            if(candidates[idx]<=target){
                ds.push_back(candidates[i]);
                findAll(candidates, target-candidates[i], i+1, ans, ds);
                ds.pop_back();
            }
        }
    }

    vector<vector<int>> combinationSum2(vector<int>& candidates,
                                        int target){
        vector<vector<int>> ans;
        sort(candidates.begin(), candidates.end());
        vector<int> ds;
        findAll(candidates, target, 0, ans, ds);
        return ans;
    }
};
```

(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.

Trees - Part 1

- Karun Karthik

Contents

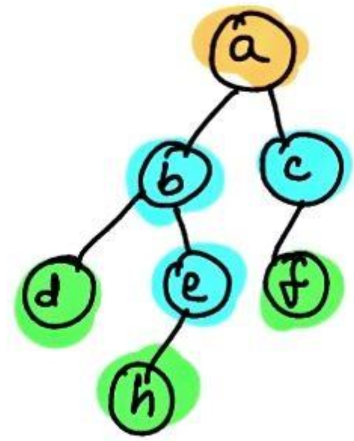
Trees

why trees?

1. Hirarchy
2. Computer system.
   (UNIX)

Tree - collection of tree-nodes

① Class Treenode
- data
- list <Treenode> children



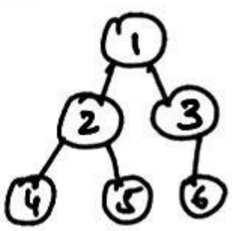
Root Leaf Internal

② Binary Tree → atmost 2 children (0,1,2)
- data
- leftchild
- rightchild

③ Types →

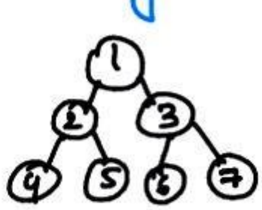
Ⓐ Complete Binary Tree
↳ all levels are completely filled except last one

Eg



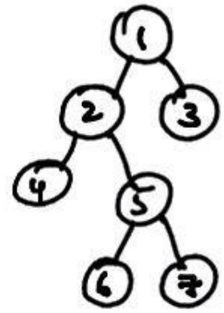
Ⓑ Perfect Binary Tree
↳ every internal node has exactly 2 children

Eg



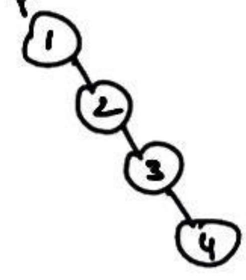
Ⓒ Full Binary tree
↳ if every node has 0 or 2 children

Eg



Ⓓ Skewed Binary Tree
(* used for finding complexity)
↳ all nodes have either one or no child.

Eg

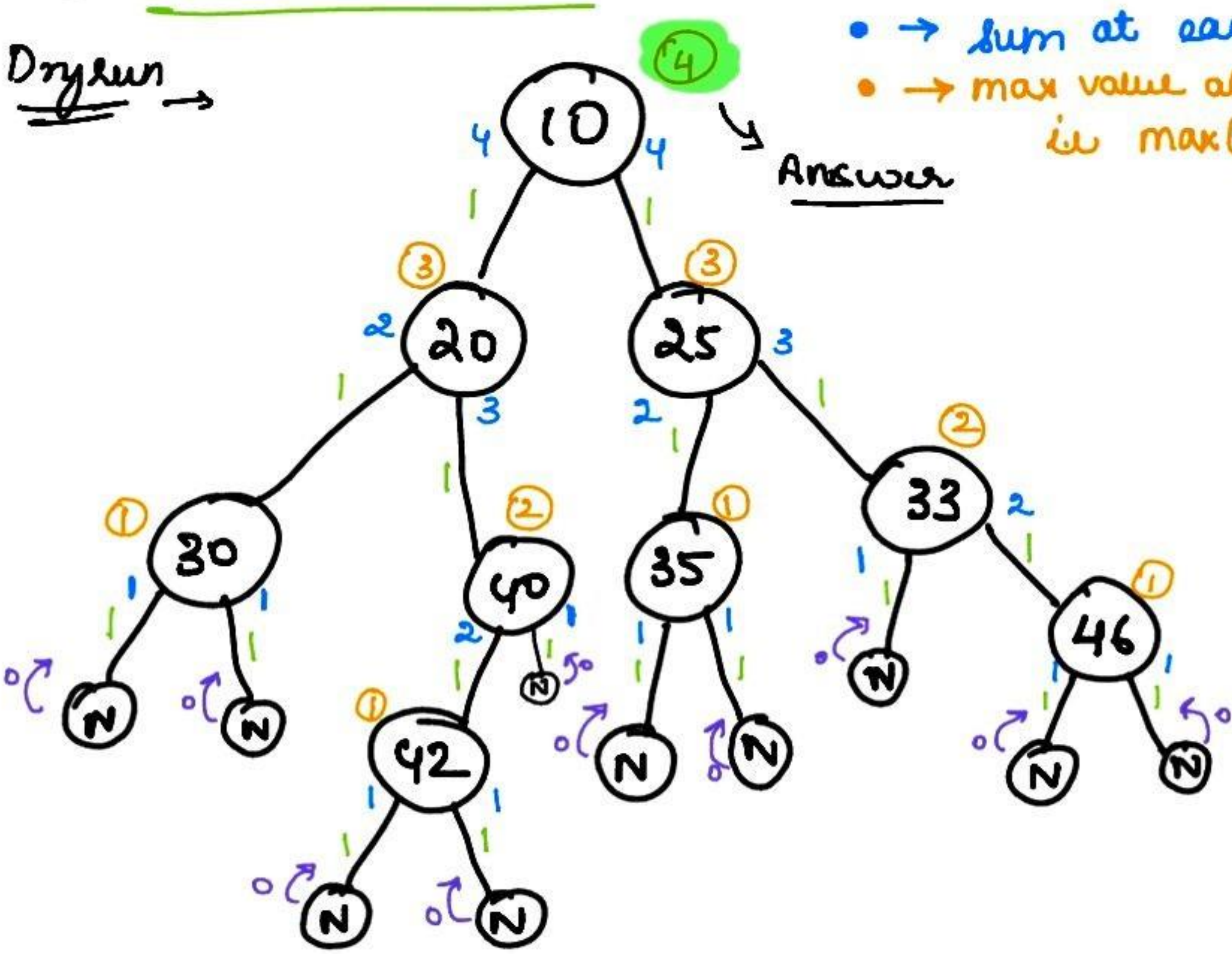


D1

① Depth of a binary tree (Max depth)

- → 1 added while returning.
- → sum at each node.
- → max value at node is max(left, right).

Dryrun →



if null then ht = 0

- consider max at a node i.e either left or right

TC → O(n)
SC → O(1)
AUX → O(h)
h → height

Code →

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(n
 *     TreeNode(int x) : val(x), left(nullptr), ri
 *     TreeNode(int x, TreeNode *left, TreeNode *r
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;

        int lefth= 1+ maxDepth(root->left);
        int righth = 1+maxDepth(root->right);
        return max(lefth,righth);
    }
};
```

② Maximum depth of n-ary tree

Idea is same as previous problem, only implementation changes

Code →

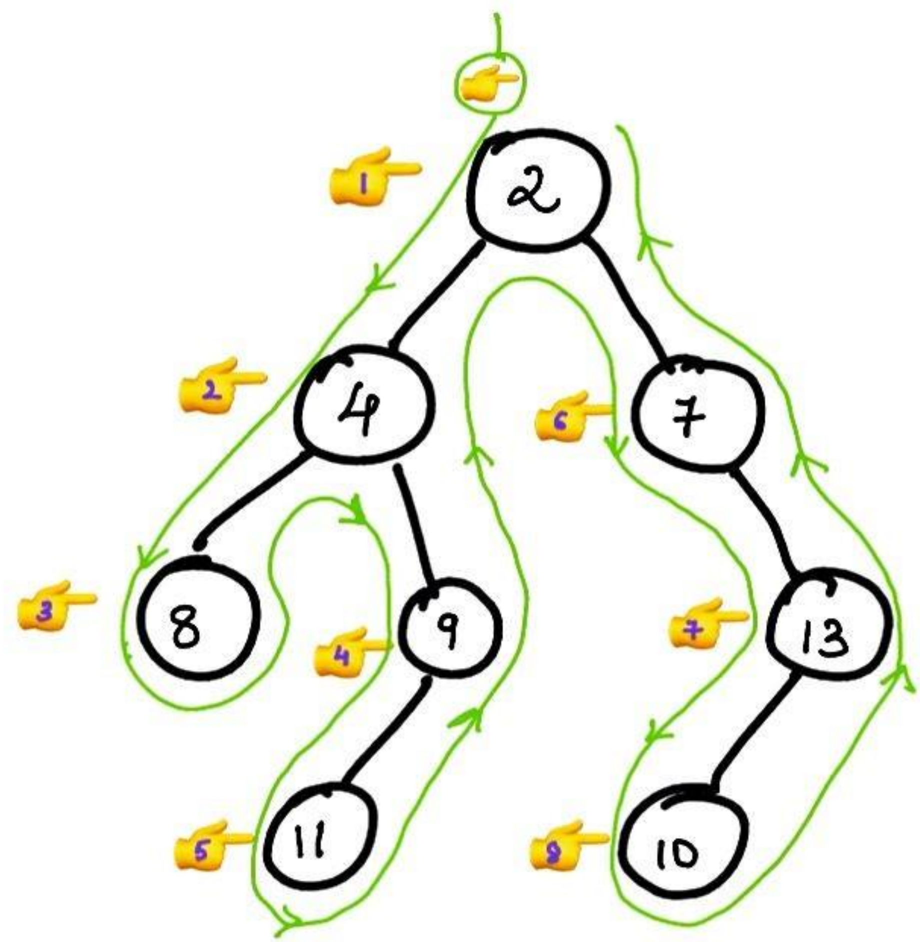
```cpp
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
public:
    int maxDepth(Node* root) {
        if(root==NULL) return 0;
        int ans=0;
        for(int i=0;i<root->children.size();i++)
        {
            int tempans = maxDepth(root->children[i]);
            ans = max(ans,tempans);
        }
        return ans+1;
    }
};
```

D2 Traversals → DFS → Preorder
→ Inorder
→ Postorder
→ BFS → Levelorder

(A) Preorder →

processing order → node
left child
right child

Eg



* Point finger as shown and traverse the tree starting from Root

* Order of visiting is the preorder traversal.

[2, 4, 8, 9, 11, 6, 13, 10]

TC → O(n)

SC → O(n) Recursive stack space → O(h) h → height.

(3) Pre-order traversal of Binary tree

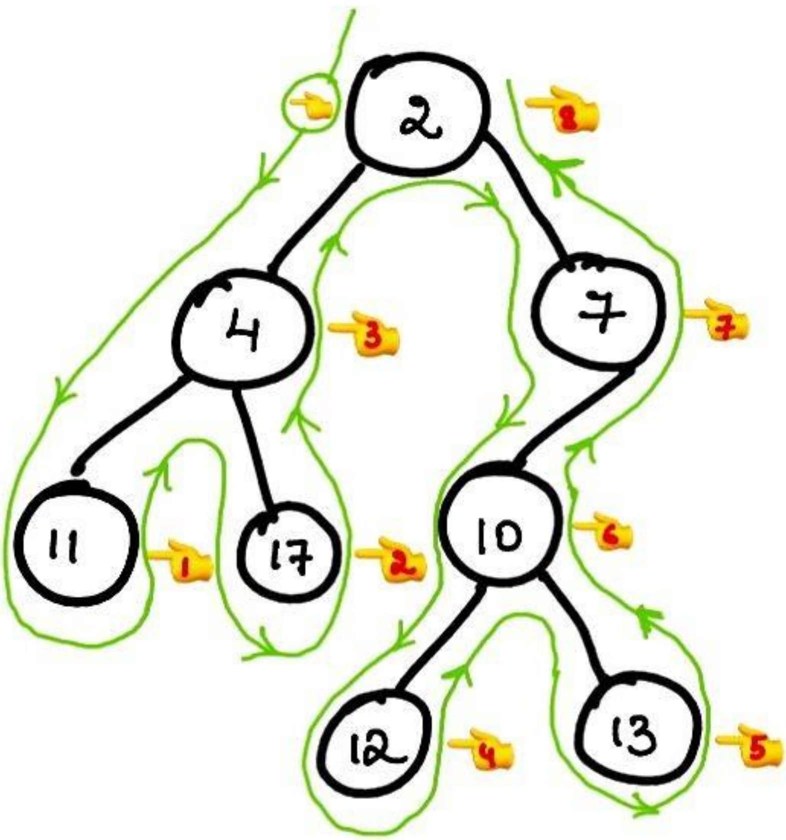
```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;
        ans.push_back(root->val);
        Preorder(root->left,ans);
        Preorder(root->right,ans);
        return;
    }
};
```

(4) Pre-order traversal of n-ary tree

```
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(Node* root, vector<int>&ans)
    {
        if(root==NULL) return;
        ans.push_back(root->val);
        for(int i=0;i<root->children.size();i++)
        {
           Preorder(root->children[i],ans);

        }
        return;
    }
};
```

(B) Postorder → processing order → left child
right child
node

Eg



* Point finger as shown and traverse the tree starting from Root

* Order of visiting is the postorder traversal.

[11, 17, 4, 12, 13, 10, 7, 2]

TC → O(n)

SC → O(n) Recursive stack space → O(h) h → height.

⑤ Postorder traversal of Binary tree

```cpp
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;

        Postorder(root->left,ans);
        Postorder(root->right,ans);
         ans.push_back(root->val);
        return;
    }
};
```

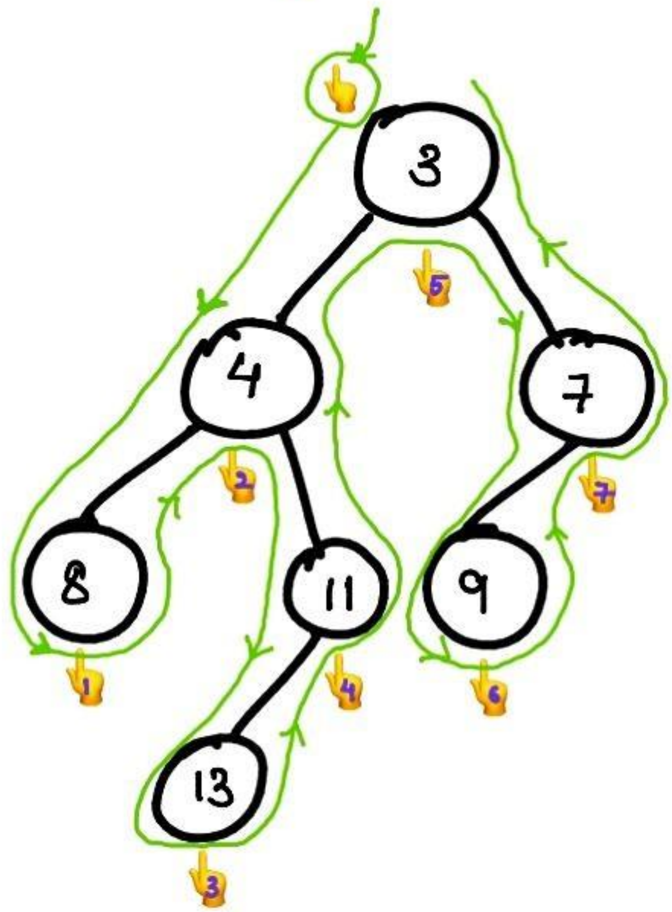
⑥ Postorder traversal of nary tree

```cpp
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(Node* root, vector<int>&ans)
    {
        if(root == NULL) return;
        for(int i=0;i<root->children.size();i++)
        {
            Postorder(root->children[i],ans);
        }
        ans.push_back(root->val);
        return;
    }
};
```

(c) Inorder →

processing order → left child
node
right child

Eg



* Point finger as shown and traverse the tree starting from Root

* Order of visiting is the Inorder traversal.

[8, 4, 13, 11, 3, 9, 7]

Tc → O(n)

Sc → O(n) Recursive Stack space → O(h) h → height.

7 In-order traversal of Binary tree

```cpp
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int>ans;
        Inorder(root,ans);
        return ans;
    }
    void Inorder(TreeNode* root, vector<int>&ans)
    {
        if(root==NULL) return;
        Inorder(root->left,ans);
        ans.push_back(root->val);
        Inorder(root->right,ans);
        return;
    }
};
```

Inorder traversal of n-ary tree

Approach:

The inorder traversal of an N-ary tree is defined as visiting all the children except the last then the root and finally the last child recursively.

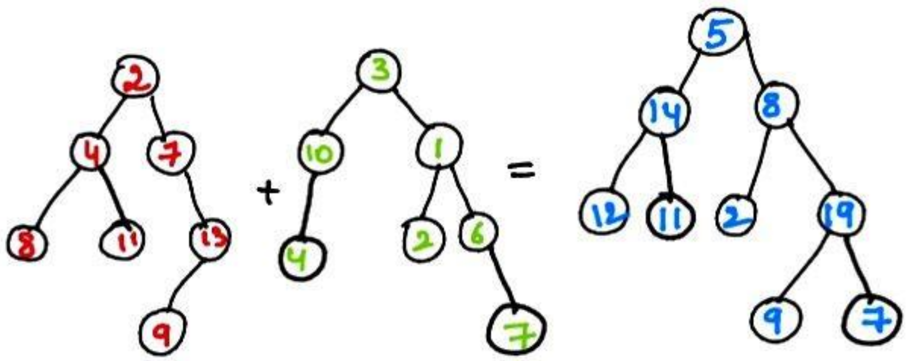
- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```cpp
void inorder(Node *node)
{
    if (node == NULL)
        return;

    // Total children count
    int total = node->length;

    // All the children except the last
    for (int i = 0; i < total - 1; i++)
        inorder(node->children[i]);

    // Print the current node's data
    cout<< node->data << " ";

    // Last child
    inorder(node->children[total - 1]);
}
```

D3 8 Merge two Binary trees →

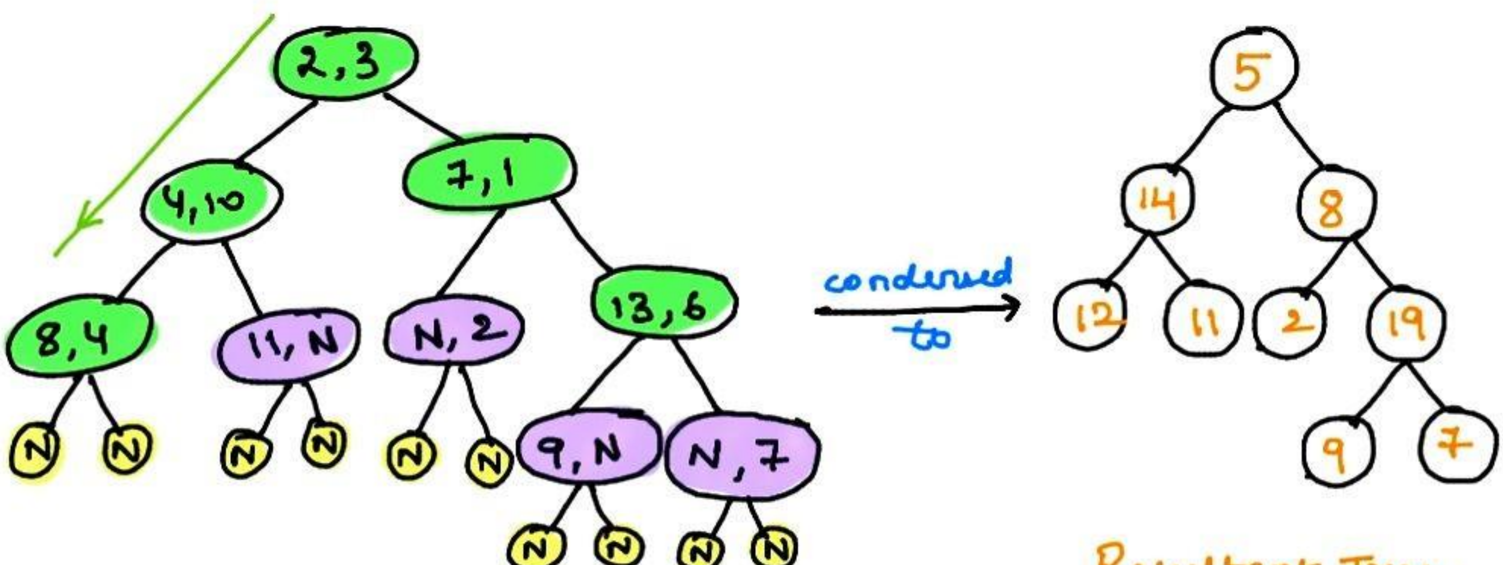
Given root nodes of 2 binary trees, return root of the sum trees.

Eg



we will perform preorder traversal on the binary tree because the node/root need to be processed first.

The recursive tree structure would be like :



Resultant Tree

- NULL & NULL
- Node & NULL
- Node & Node

TC → O(n+m)

SC → O(max(n,m))

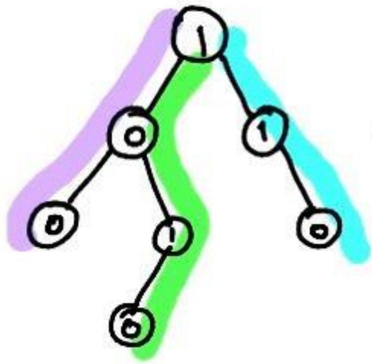
Recursive stack → O(max(h1,h2))

Code →

```
class Solution {
public:
    TreeNode* merge(TreeNode* root1, TreeNode* root2){

        if(root1==NULL && root2==NULL)  return NULL;
        if(root1==NULL) return root2;
        if(root2==NULL) return root1;

        // Create new node to store sum
        TreeNode *newNode = new TreeNode(root1->val+root2->val);

        // Recursively call the left sub-trees and right sub-trees
        newNode->left = merge(root1->left, root2->left);
        newNode->right = merge(root1->right, root2->right);

        // return the new node
        return newNode;
    }

    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        return merge(root1, root2);
    }
};
```

⑨ Sum of root to leaf paths →

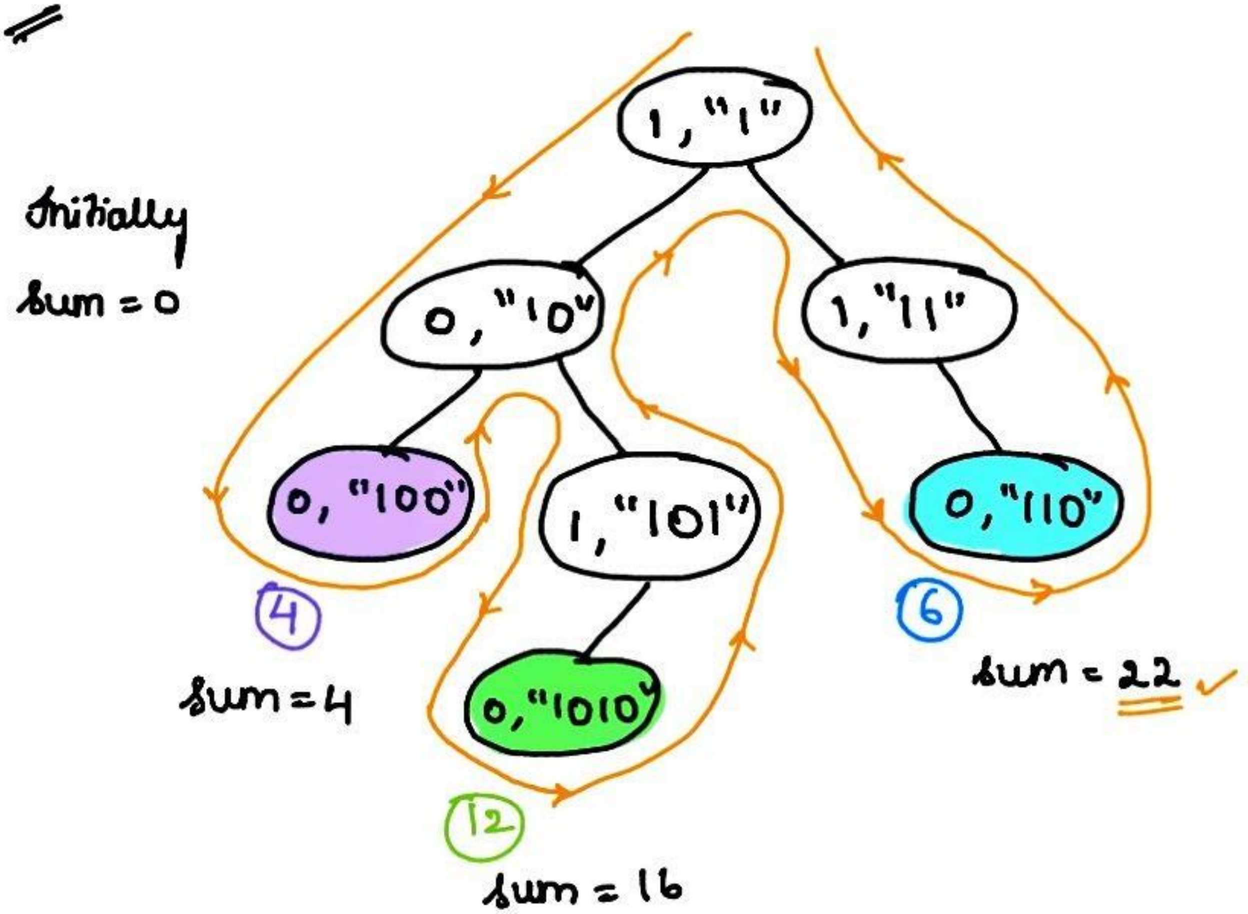
Eg



→ $(100)_2 + (1010)_2 + (110)_2$

4 + 12 + 6

= 22



* If root becomes null convert string to integer & add to sum.

Time → O(n)

Space → O(n)

Recursive stack → O(h)

Code

```
class Solution {
public:
    void rootToLeaf(TreeNode* root, string currentString,int* ans)
    {
        if(root->left== NULL && root->right==NULL)
        {
            currentString+=to_string(root->val);
            ans[0]+=stoi(currentString,0,2);
            return;
        }
        string curr=to_string(root->val);
        if(root->left!=NULL)
            rootToLeaf(root->left,currentString+curr,ans);
        if(root->right!=NULL)
            rootToLeaf(root->right,currentString+curr,ans);

    }
    int sumRootToLeaf(TreeNode* root) {
        int* ans=new int[1];
        ans[0]=0;
        rootToLeaf(root,"",ans);
        return ans[0];
    }
};
```

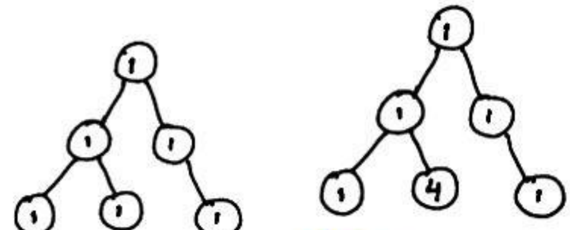
Note →

stoi() can take upto three parameters, the second parameter is for starting index and third parameter is for base of input number.

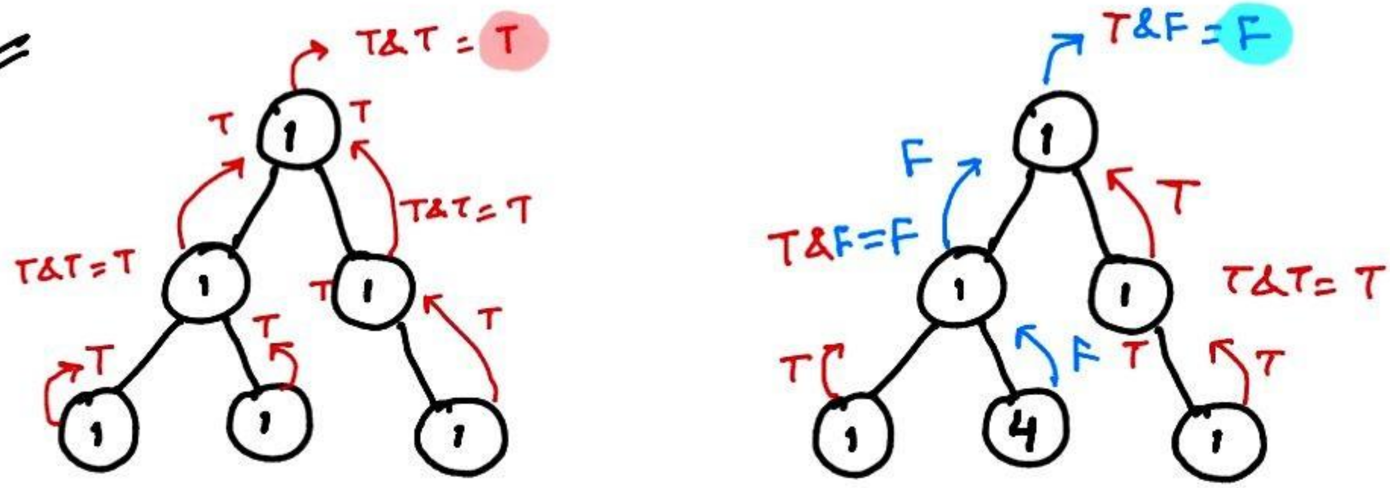
[to convert from binary to decimal we give it as 2]

(10) Univalued Binary Tree →

Eg



↳ returns true ↳ returns false



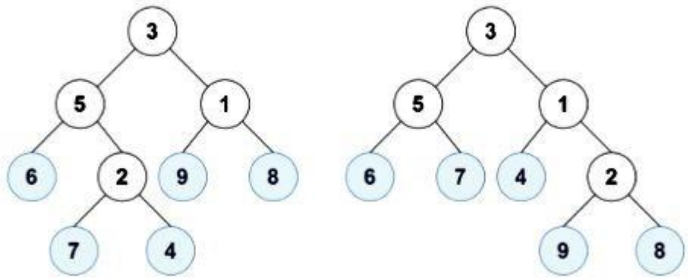
Code

```
class Solution {
public:
    bool isSame(TreeNode* root, int val){
        if(root==NULL)  return true;
        if(root->val!=val)  return false;

        bool left = isSame(root->left, val);
        bool right = isSame(root->right, val);

        return left && right;
    }

    bool isUnivalTree(TreeNode* root) {
        return isSame(root, root->val);
    }
};
```

(11) Leaf Similar trees

↳ return true if all leaves are in same order for both trees.

Eg



$v_1 = 6, 7, 4, 9, 8$ ⇒ $v_1 == v_2$

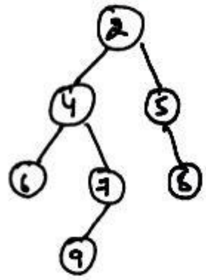
$v_2 = 6, 7, 4, 9, 8$ ↳ return true else false.

code →

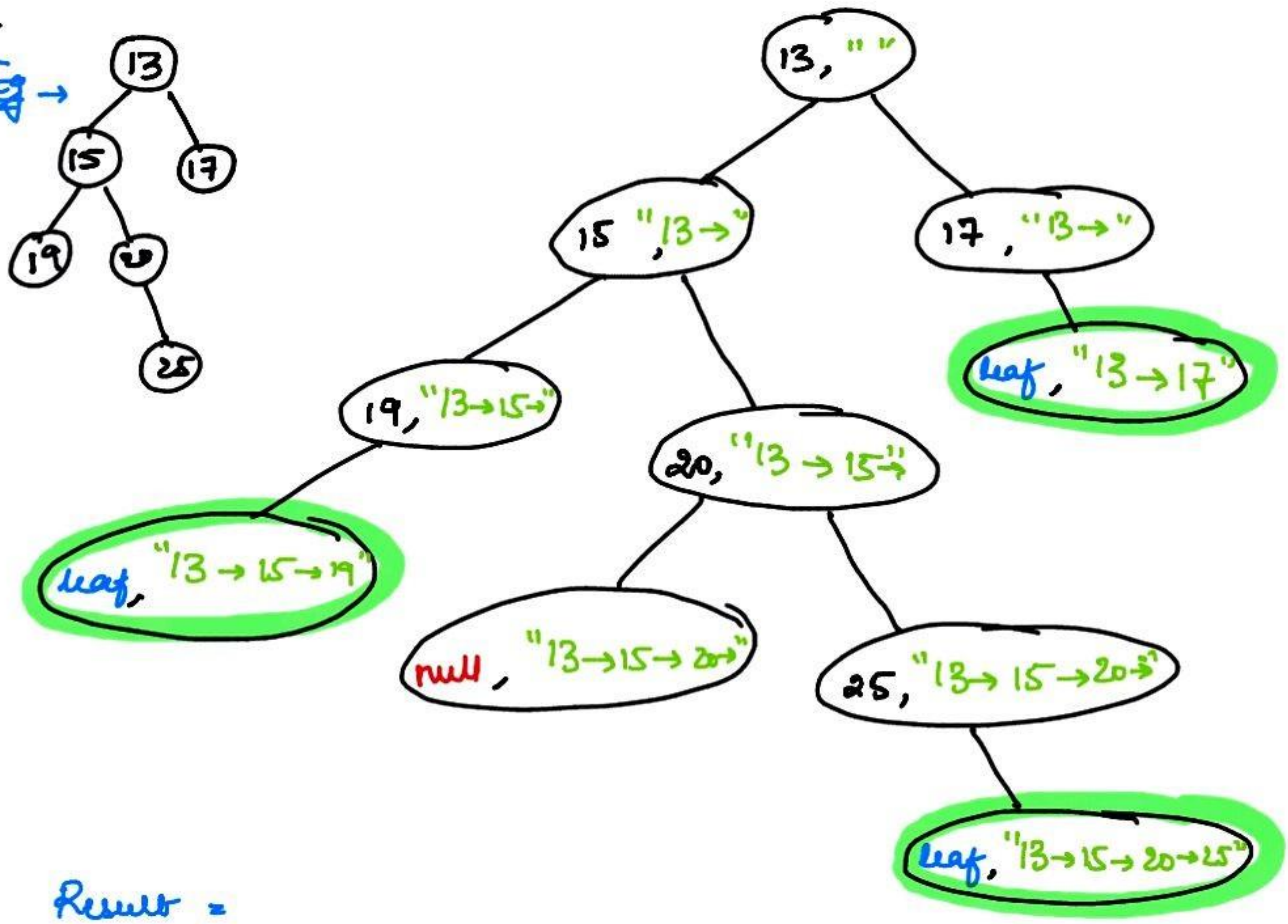
```
class Solution {
public:
    void traversal(TreeNode* root, vector<int>&v){
        if(root==NULL)
            return;

        if(root->left==NULL && root->right==NULL)
            v.push_back(root->val);

        if(root->left!=NULL)
            traversal(root->left, v);

        if(root->right!=NULL)
            traversal(root->right, v);
    }

    bool leafSimilar(TreeNode* root1, TreeNode* root2) {
        vector<int> a;
        vector<int> b;
        traversal(root1,a);
        traversal(root2,b);
        return a==b;
    }
};
```

D5 (12) Binary tree paths

↳ given root print all the paths from root to leaf.

Eg 

⇒ ["2→4→6", "2→4→7→9", "2→5→8"]

Eg → 

Result =

["13→15→19", "13→15→20→25", "13→17"]

Time complexity = $O(n)$

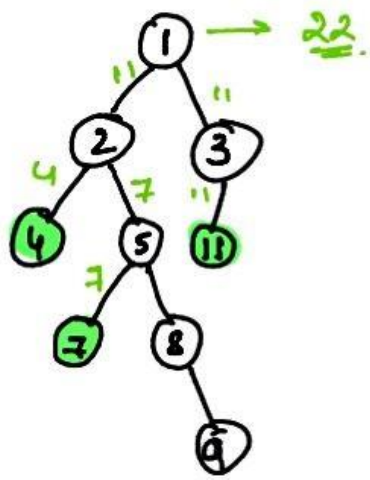
Space complexity = $\underbrace{O(n)}_{\text{Answer array}} + \underbrace{O(h)}$ → recursive stack.

Code →

```
class Solution {
public:
    void pathFinder(TreeNode *root, vector<string> &res, string currPath){

        if(root==NULL)  return;

        // if leaf then add it's value to currentPath
        if(root->left == NULL && root->right==NULL){
            currPath += to_string(root->val);
            res.push_back(currPath);
            return;
        }

        // else add the node's value to path
        currPath += to_string(root->val)+"->";

        if(root->left)  pathFinder(root->left, res, currPath);
        if(root->right) pathFinder(root->right, res, currPath);
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        pathFinder(root, res, "");
        return res;
    }
};
```

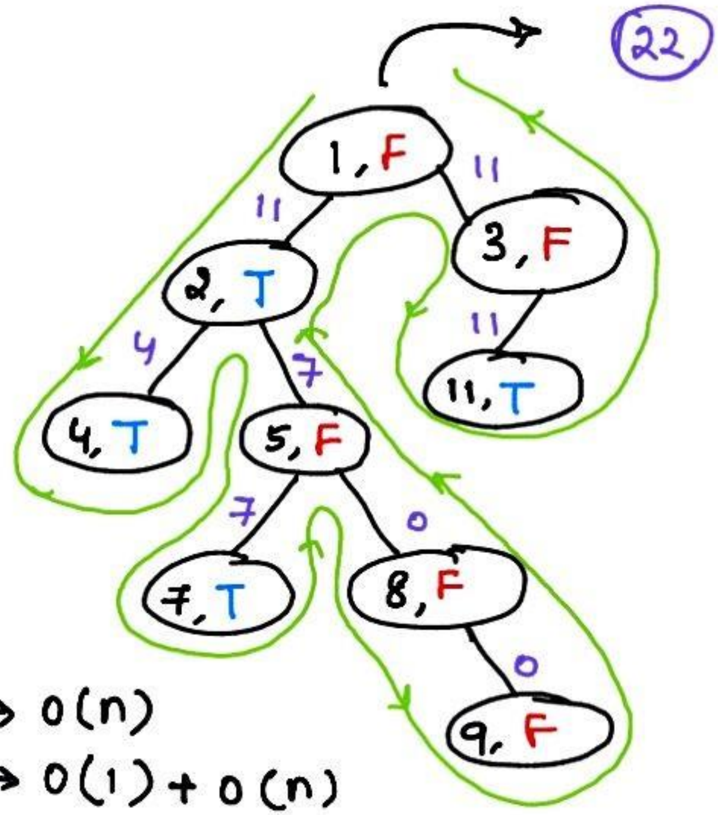
(13) Sum of left leaves →

Eg



Result = 4 + 7 + 11

= 22



Tc → O(n)

Sc → O(1) + O(n)

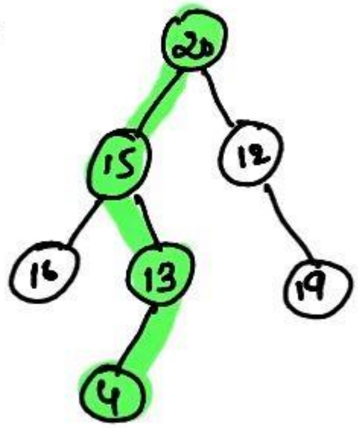
→ stack

Code →

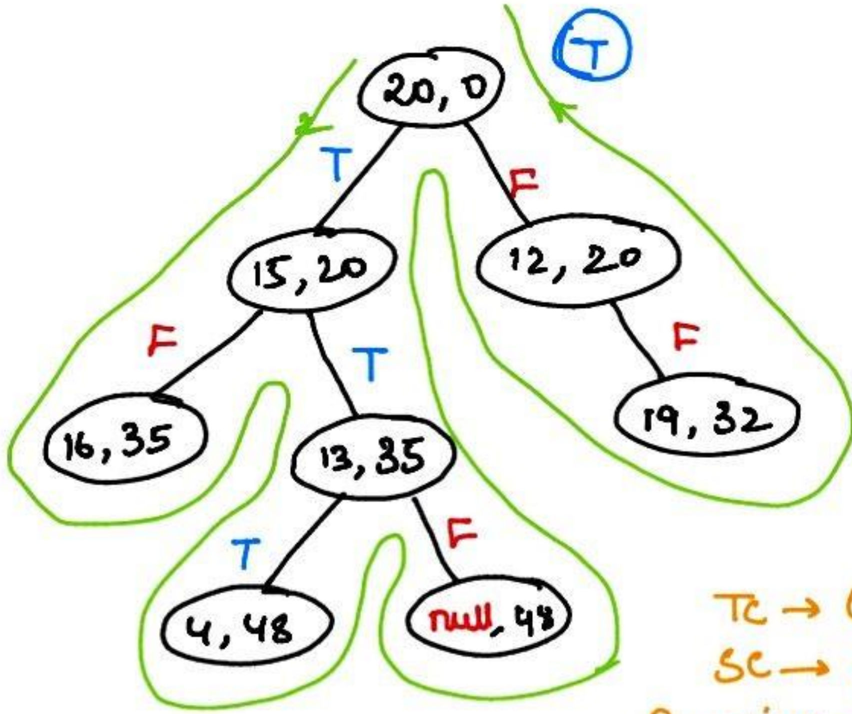
```cpp
class Solution {
public:
    int leftLeafSum(TreeNode *root, bool leaf){
        if(root==NULL){
            return 0;
        }
        if(root->left==NULL && root->right==NULL && leaf){
            return root->val;
        }
        int ls = leftLeafSum(root->left, true);
        int rs = leftLeafSum(root->right, false);
        return ls+rs;
    }

    int sumOfLeftLeaves(TreeNode* root) {
        return leftLeafSum(root, false);
    }
};
```

(14) Path sum → sum of all nodes from root to leaf is equal to target sum → then T else F.

Eg



target = 52



TC → O(n)

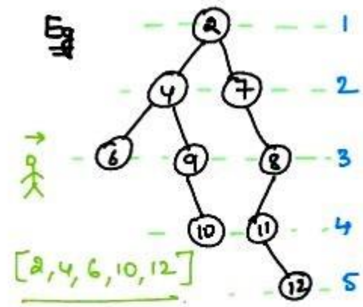
SC → O(1)

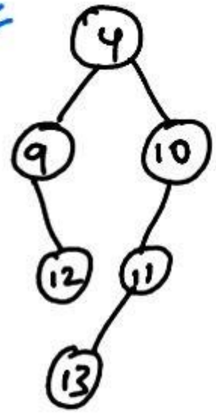
Recursive Stack → O(h)

Code

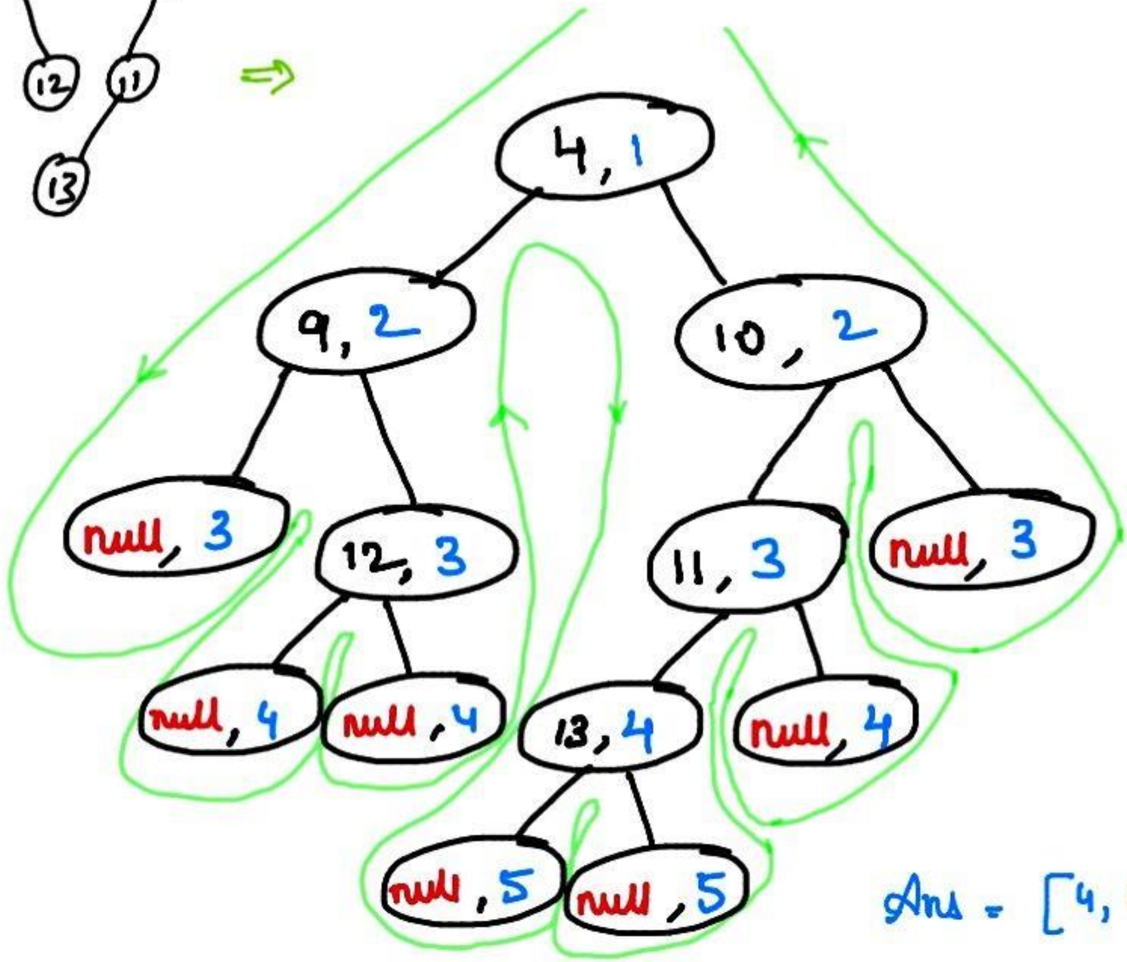
```cpp
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        if(root->left==NULL && root->right==NULL){
            return (currSum+root->val)==targetSum;
        }

        return pathSumUtil(root->left, currSum+root->val, targetSum)
            ||pathSumUtil(root->right, currSum+root->val, targetSum);
    }

    bool hasPathSum(TreeNode* root, int targetSum) {
        return pathSumUtil(root, 0, targetSum);
    }
};
```

D6 (15) Left view of a Binary Tree





⇒



Ans = [4, 9, 12, 13]

→ For every level traversed,
check if it already exist in the set,

if already exist then continue,
else add the root's value
to array & into the set

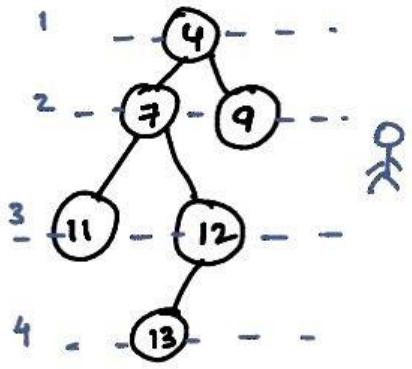
$Tc \rightarrow O(n)$

$Sc \rightarrow \underbrace{O(n)}_{result} + O(n) + O(h)$

Code →

```cpp
void viewGenerator(Node *root, vector<int> &res, set<int> &s, int currLevel){
    if(root==NULL) return;
    // if level is not reached, then add to result and the set
    if(s.find(currLevel)==s.end()){
        s.insert(currLevel);
        res.push_back(root->data);
    }
    // traverse the remaining branches
    viewGenerator(root->left, res, s, currLevel+1);
    viewGenerator(root->right, res, s, currLevel+1);
    return;
}

vector<int> leftView(Node *root)
{
    vector<int> res;
    set<int> s;
    viewGenerator(root, res, s, 0);
    return res;
}
```

(16) Right view of Binary Tree →



Result = [4, 9, 12, 13].

→ The entire approach to solve the problem is same as the left view of binary tree. Even the time complexities.

→ only order of calling the branches change.
① right
② left

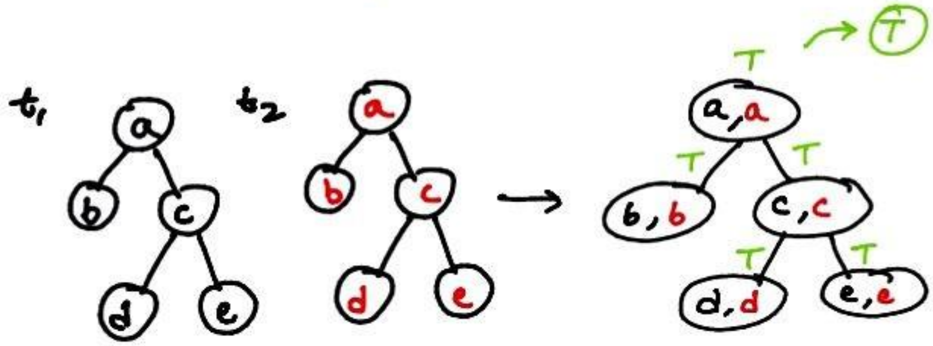
code

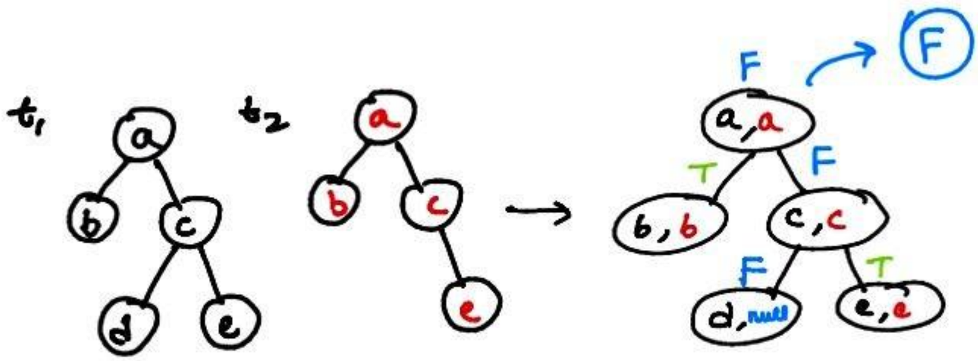
```
class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);
        }
        // traverse the remaining branch
        viewGenerator(root->right, res, s, currLevel+1);
        viewGenerator(root->left, res, s, currLevel+1);
        return;
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        set<int> s;
        viewGenerator(root, res, s, 0);
        return res;
    }
};
```

Tc → O(n)

Sc → $\underbrace{O(n)}_{\text{Result}}$ + O(n) + O(h)

(17) Same tree → return true if both tree are same else false





Tc → O(min(m, n))

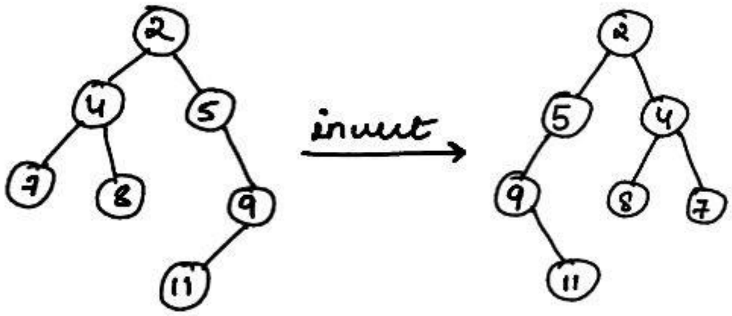
Sc → O(1) + O(min(h1, h2))

code →

```
class Solution {
public:

    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==NULL && q==NULL) return true;

        if(p==NULL || q==NULL || p->val != q->val) return false;

        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

    }
};
```

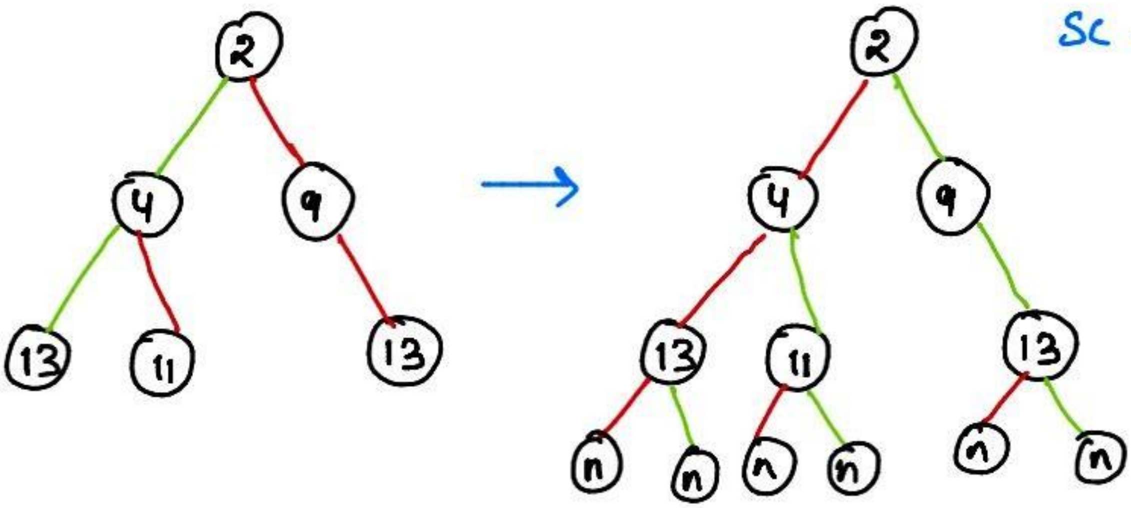
(18) Invert Binary Tree → given the root of BT, find its mirror img.

Eg



TC → O(n)

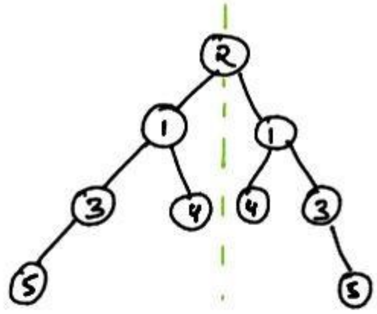
SC → O(n) + O(h)



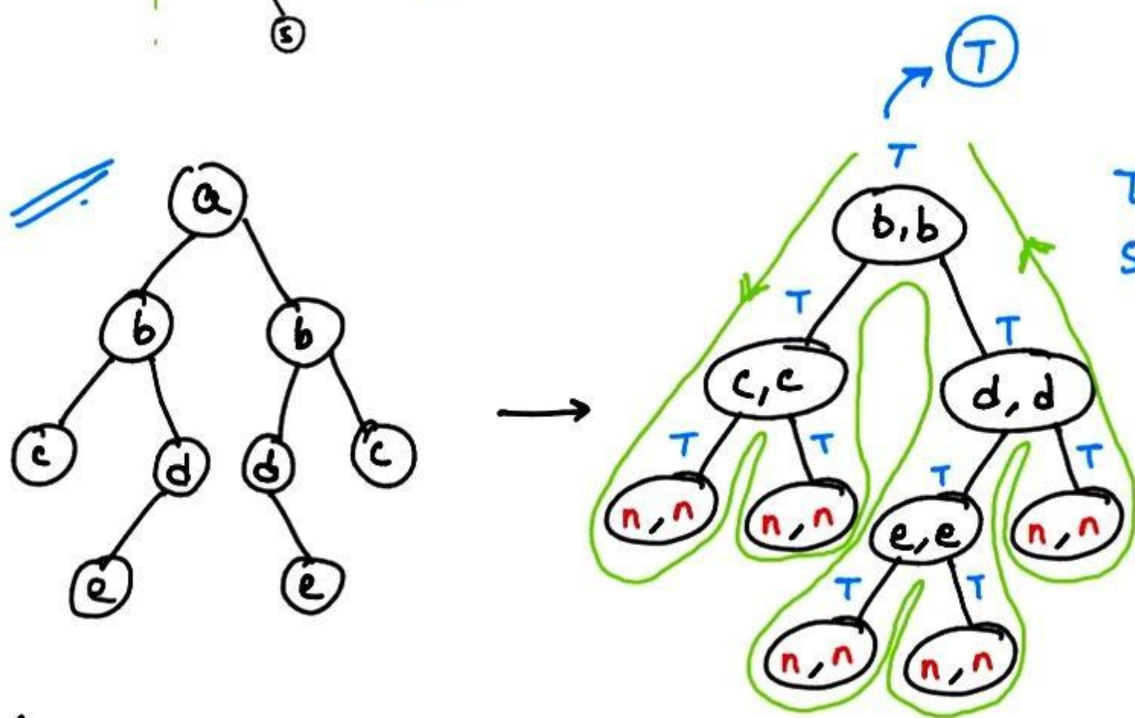
Code →

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root==NULL)  return root;

        /* invert the left and right sub-trees and store
            them separately */
        TreeNode *leftSub = invertTree(root->right);
        TreeNode *rightSub = invertTree(root->left);

        // attach the branches to root
        root->left = leftSub;
        root->right = rightSub;

        return root;
    }
};
```

D7 19 Symmetric Tree → return true if left subtree is equal to right subtree, else return false



→ returns true



Tc → O(n)
Sc → O(1) + O(h)

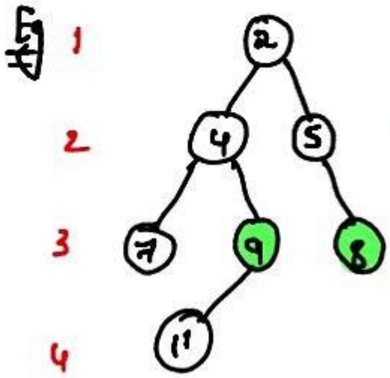
Code →

```cpp
class Solution {
public:
    bool isMirror(TreeNode* l, TreeNode* r){

        if(l== NULL && r==NULL)
            return true;
        else if(l==NULL || r==NULL)
            return false;
        else if(l->val != r->val)
            return false;

        return isMirror(l->left,r->right) && isMirror(l->right, r->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(root==NULL) return true;
        return isMirror(root->left, root->right);
    }
};
```

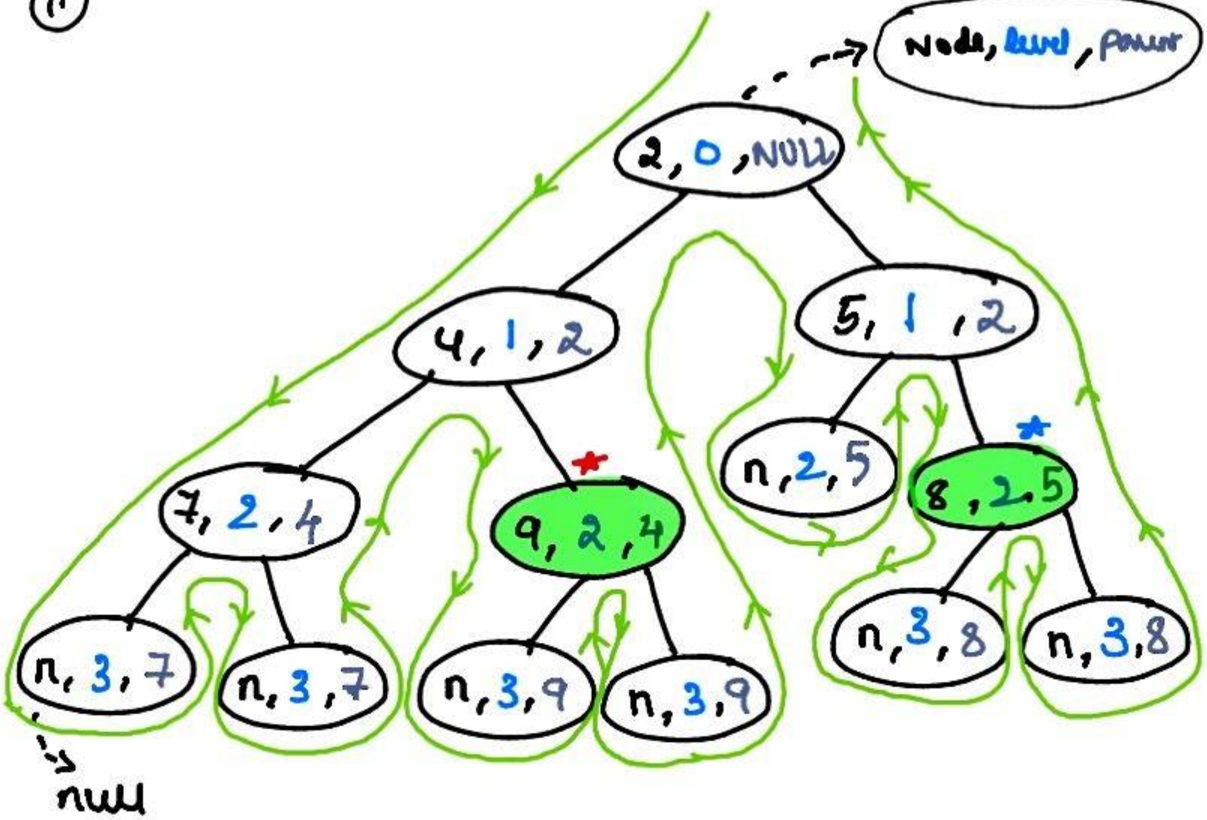
(20) Cousins of a Binary Tree → given two nodes, find if they are cousins of each other.

↳ same level but diff parents.

Eg



$X = 9$, $Y = 8$ } cousins



* at this step as value = 9 i.e. X is found store it's parent & level in separate variables
* later compare its value with other occurrence i.e. Y such that

1) X.parent != Y.parent
2) X.level == Y.level.

TC → O(n)

SC → O(1)

Recursive Stack → O(n)

Code

```cpp
class Solution {
public:
    void findNodes(TreeNode* root, int x, int y,int level[2],int parents[2],int currlevel,TreeNode* currparent)
    {
        if(root==NULL)return;
        if(root->val == x)
        {
            level[0]=currlevel;
            parents[0]=currparent->val;
        }
        if(root->val == y)
        {
            level[1]=currlevel;
            parents[1]=currparent->val;
        }
        findNodes(root->left, x, y, level, parents, currlevel+1, root);
        findNodes(root->right, x, y, level, parents, currlevel+1, root);
    }
    bool isCousins(TreeNode* root, int x, int y) {
        int level [2] = {-1,-1};
        int parents[2] = {-1,-1};
        findNodes(root, x, y, level, parents, 0, new TreeNode(-1));
        if(level[0]==level[1] && parents[0]!=parents[1])
             return true;
        return false;
    }
};
```

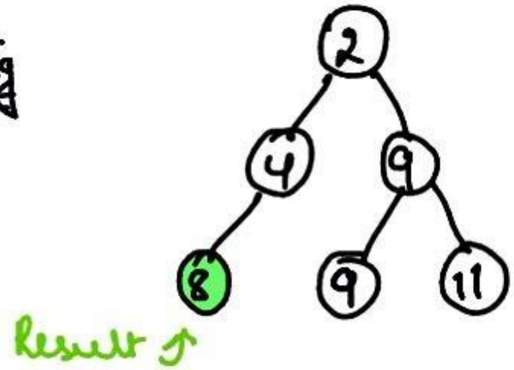
Trees - Part 2

- Karun Karthik

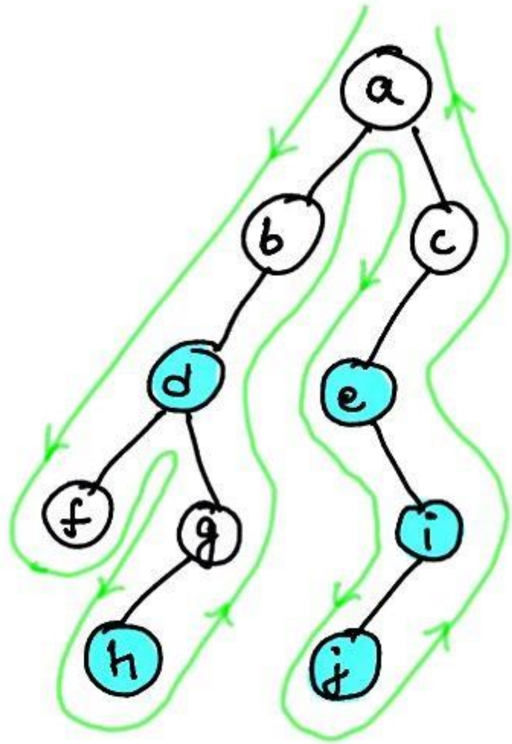
Contents

D7 (21) Print all nodes that donot have any siblings

Eg



Sibling → same level, same parent



TC → O(n)

SC → O(n)

At every node, check if

both branches exist → then call both of them recursively

only left branch exist → then call left branch recursively

only right branch exist → then call right branch recursively

Code →

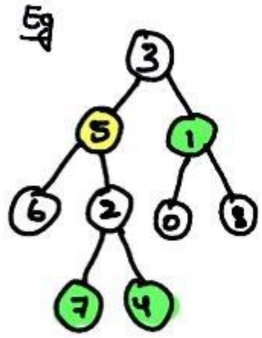
```
void findNode(Node* root, vector<int>&res){

    if(root==NULL) return;
    if(root->left == NULL && root->right==NULL) return;

    // both branches present then call recursively
    if(root->left!=NULL && root->right!=NULL)
    {
        findNode(root->left, res);
        findNode(root->right, res);
    }
    else if(root->left!=NULL)   // right branch absent
    {
        res.push_back(root->left->data);
        findNode(root->left, res);

    } else if(root->right!=NULL)    // left branch absent
    {
        res.push_back(root->right->data);
        findNode(root->right, res);
    }
    return;
}

vector<int> noSibling(Node* node)
{
    vector<int> res;
    findNode(node, res);
    if(res.size()==0) res.push_back(-1);
    sort(res.begin(), res.end());
    return res;
}
```

D8 (22) All nodes distance K in Binary Tree

Given a source node, find all the nodes that are at a distance of K units.

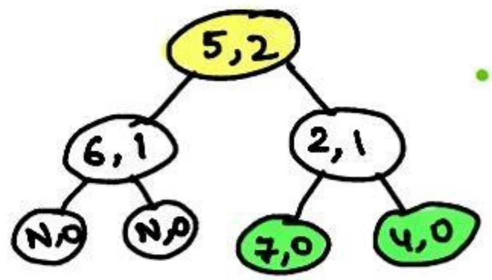
① Consider nodes in downward direction of source.

Eg



Source = 5
k = 2

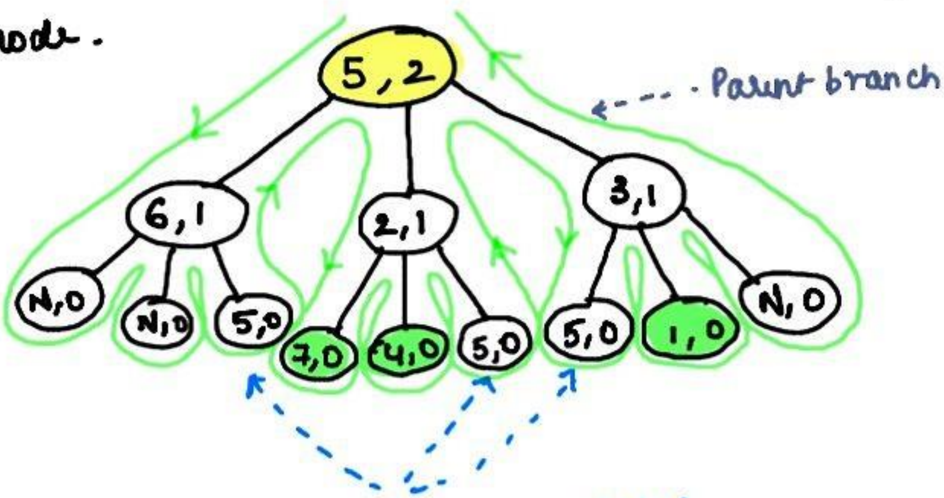
src, k



• Store in result if k = 0
Return if k < 0

② To solve for the upward direction we can use hashing to store the parent node.

Node	Parent
3	NULL
5	3
1	3
6	5
2	5
0	1
8	1
7	2
4	2



↳ to handle such cases we use a set.

* If k = 0 & the node value is valid (not null) then add to result array

$$TC \rightarrow \underbrace{O(n)}_{\text{populating}} + \underbrace{O(n)}_{\text{traversal}}$$

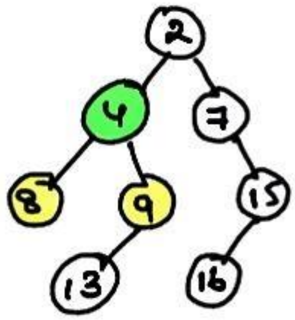
$$SC \rightarrow O(n) + O(n) + \underbrace{O(x)}_{\text{result}}$$

Code.

```
class Solution {
public:
    // to create hashtable
    void populatemap(TreeNode* currnode, TreeNode* currparent,
                     unordered_map<TreeNode*,TreeNode*>&parentmap){
        if(currnode == NULL) return;
        parentmap[currnode] = currparent;
        populatemap(currnode->left,currnode,parentmap);
         populatemap(currnode->right,currnode,parentmap);
        return;
    }

    // finding all the nodes at distance K
    void printkdistance(TreeNode* currnode, int k,set<TreeNode*>&s,
        unordered_map<TreeNode*,TreeNode*>&parentmap,vector<int>&ans)
    {
        if(currnode == NULL || s.find(currnode)!=s.end()|| k<0)
            return;

        s.insert(currnode);

        if(k==0)
        {
            ans.push_back(currnode->val);
            return;
        }

        printkdistance(currnode->left,k-1,s,parentmap,ans);   // call left child
        printkdistance(currnode->right,k-1,s,parentmap,ans);   // call right child
        printkdistance(parentmap[currnode],k-1,s,parentmap,ans);  // call the parent
        return;
    }

    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        vector<int>ans;
        set<TreeNode*>s;
        unordered_map<TreeNode*,TreeNode*>parentmap;
        populatemap(root,NULL,parentmap);
        printkdistance(target,k,s,parentmap,ans);
        return ans;
    }
};
```

(23) Lowest Common Ancestor

Eg



node to root paths

$n_1 = 8$, $n_2 = 9$ then $\left.\begin{array}{l}\bar{n}_1 = [8, 4, 2] \\ \bar{n}_2 = [9, 4, 2]\end{array}\right\} \rightarrow 4$.

$n_1 = 9$ $n_2 = 13$ then $\left.\begin{array}{l}\bar{n}_1 = [9, 4, 2] \\ \bar{n}_2 = [13, 9, 4, 2]\end{array}\right\} \rightarrow 9$

→ for every node, check if it matches n1 or n2.
if found return node
else call recursively in both branches.
if both return non-null value ⇒ root is LCA
else return the branch value that is non-null.

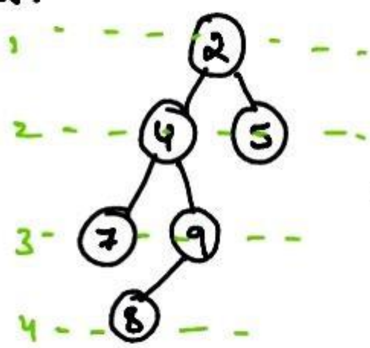
<u>code</u>

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;
        if(root->val == p->val || root->val ==q->val) return root;
        TreeNode* leftSubTree = lowestCommonAncestor(root->left, p, q);
        TreeNode* rightSubTree = lowestCommonAncestor(root->right, p, q);
        if(leftSubTree!=NULL && rightSubTree!=NULL) return root;
        if(leftSubTree!=NULL) return leftSubTree;
        if(rightSubTree!=NULL) return rightSubTree;
        return NULL;
    }
};
```

D9 (24) **Level order traversal Binary Tree**

Given root node, find level order traversal.

Eg.



⇒ [[2], [4, 5], [7, 9], [8]]

TC → O(n)

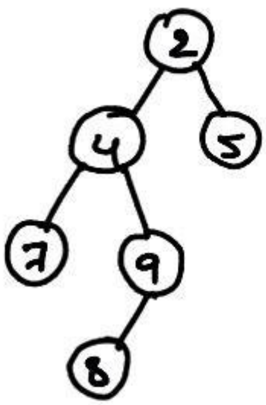
SC → O(n)

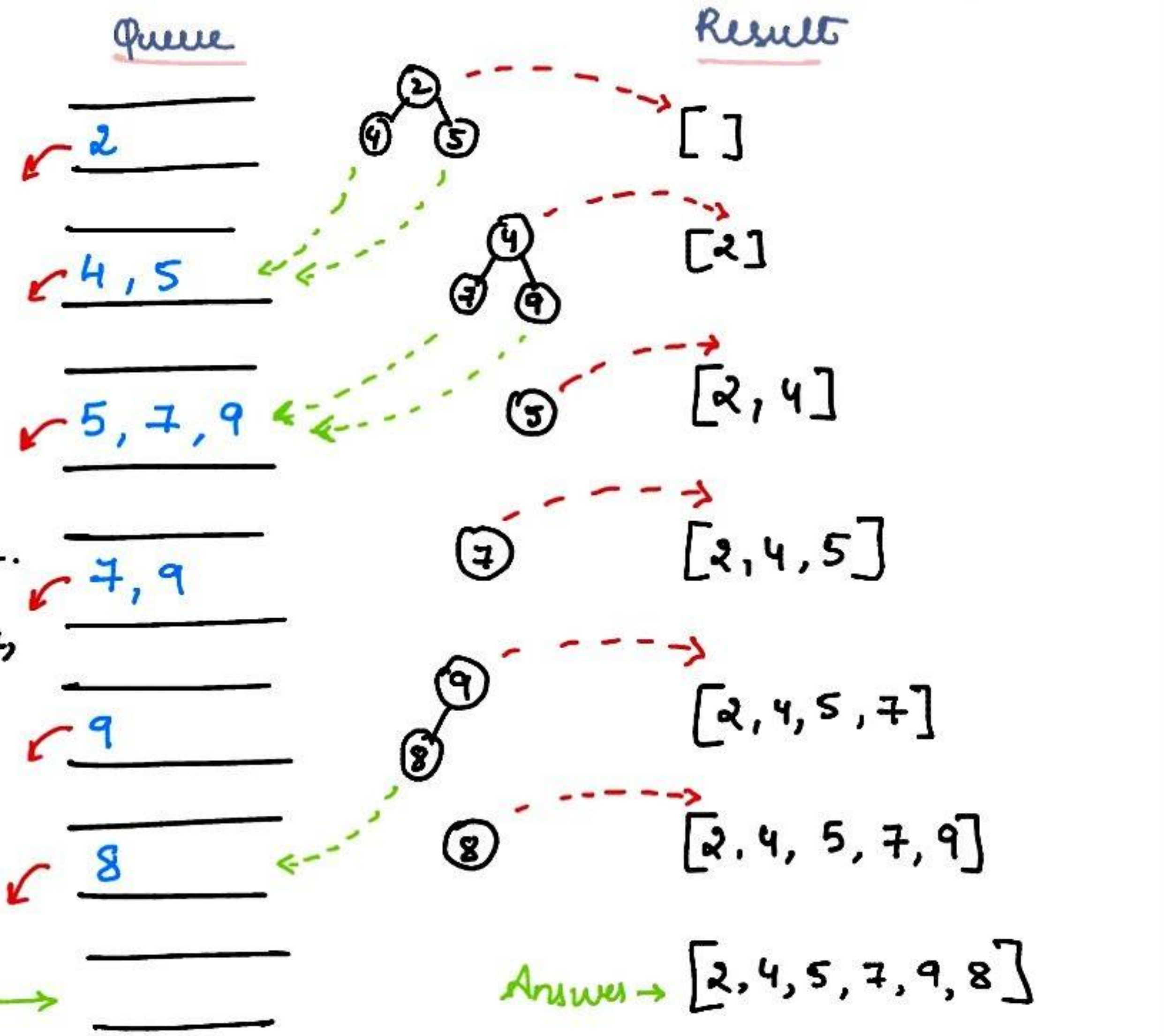
→ To find level order traversal use queue. → FIFO datastructure

Inserting → Rear

Removing → Front

→ Before removing from queue, add the children to the queue (BFS)





① For every node, enqueue.

② while dequeue, enqueue its branches.

Code →

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        queue<TreeNode*> q;

        if(root==NULL)  return res;
        q.push(root);

        while(!q.empty()){

            int currsize = q.size();
            vector<int>currLevel;

            while(currsize>0)
            {
                TreeNode* currnode = q.front();
                q.pop();
                currLevel.push_back(currnode->val);
                currsize--;

                if(currnode->left!=NULL)
                q.push(currnode->left);

                if(currnode->right!=NULL)
                q.push(currnode->right);
            }
            res.push_back(currLevel);
        }
        return res;
    }
};
```

(25) Level Order traversal N-ary Tree

→ Everything is same as previous problem, intuition & complexity

TC → O(n)

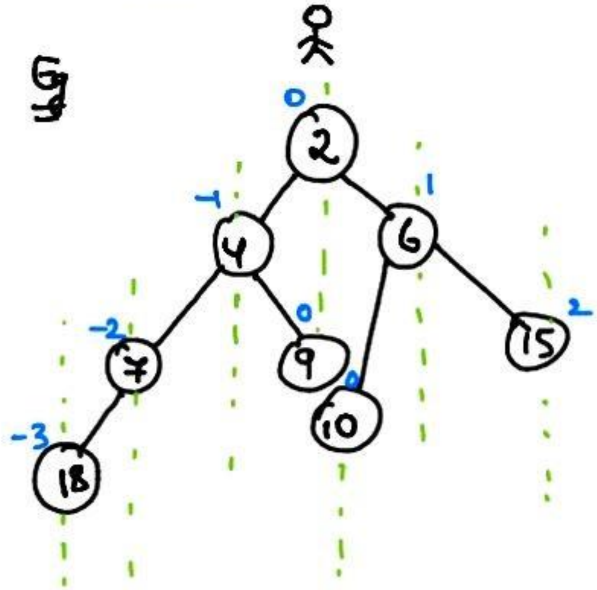
SC → O(n)

code →

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> res;
        queue<Node*>q;

        if(root == NULL) return res;
        q.push(root);

        while(!q.empty())
        {
            int currsize = q.size();
            vector<int>currLevel;
            while(currsize>0)
            {
                Node* currnode = q.front();
                q.pop();
                currLevel.push_back(currnode->val);
                currsize--;

                // enqueue all the children
                for(auto child:currnode->children)
                    q.push(child);
            }
            res.push_back(currLevel);
        }
         return res;
    }
};
```

(26) Top View of Binary Tree

Eg

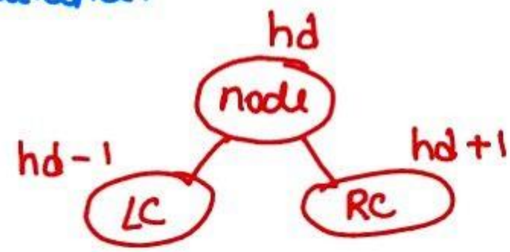


Left to Right

↳ [18, 7, 4, 2, 6, 15]

* For top view or bottom view we use concept of horizontal distance.

*



* hd of root = 0

* make a pair with node & it's hd. & perform bfs.

<node, hd>

↳

①	② ③	④ ⑤	⑥ ⑦	⑧
(2,0)	(4,-1) (6,1)	(7,-2) (9,0)	(10,0) (15,2)	(18,-3)

use a hashmap to store result.

HD	NODE
0	2
-1	4
1	6
-2	7
2	15
-3	18

① As hd = 0 is not present in map add 2 to map.

② As hd = -1 is not present in map add 4 to map.

③ As hd = 1 is not present in map add 6 to map.

④ As hd = -2 is not present in map add 7 to map.

⑤ hd = 0 is already present.

⑥ hd = 0 is already present.

⑦ As hd = 2 is not present in map add 15 to map.

⑧ As hd = -3 is not present in map add 18 to map.

⟶ convert into array & return as result.

code

```cpp
class Solution
{
    public:
    vector<int> topView(Node *root)
    {
        vector<int> res;
        if(root==NULL) return res;

        map<int,int> mp;
        queue<pair<Node*,int>> q;

        q.push({root,0});

        while(!q.empty()){

            auto it = q.front();
            q.pop();

            Node* node = it.first;
            int hd = it.second;

            if(mp.find(hd) == mp.end())
                mp[hd] = node->data;

            if(node->left!=NULL)
                q.push({node->left,hd-1});


            if(node->right!=NULL)
                q.push({node->right,hd+1});
        }

        // store in vector or array
        for(auto it:mp)
            res.push_back(it.second);

        return res;
    }
};
```

logn → map.

Tc → O(nlogn)

Sc → O(n)

(27) Bottom View of Binary Tree

→ Similar to top view, but replace entries in hashmap so you'll get last possible element with particular hd.

Code →

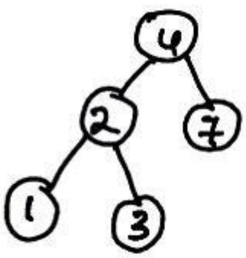
```
class Solution {
  public:
    vector <int> bottomView(Node *root) {
        vector<int> res;
        if(root==NULL) return res;

        map<int, int> mp;
        queue<pair<Node*, int>>q;

        q.push({root, 0});
        while(!q.empty()){
            auto it = q.front();
            q.pop();

            Node* node = it.first;
            int hd = it.second;

            mp[hd] = node->data;

            if(node->left!=NULL)
                q.push({node->left, hd-1});

            if(node->right!=NULL)
                q.push({node->right, hd+1});
        }

        for(auto it:mp)
            res.push_back(it.second);

        return res;
    }
};
```

D10 Binary Search Trees

- every node is > than previous node & < than next node.
- If duplicates, then it'll be mentioned that it'll be included in LC or RC

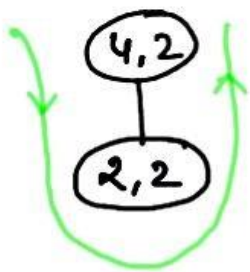
  ① LC < node < RC
  ② LC ≤ node < RC
  ③ LC < node ≤ RC

(28) Search in a BST



Val = 2.

⇒ return the subtree with given value.



- as 2 < 4, search in LST.
- as 2 == 2 return node.

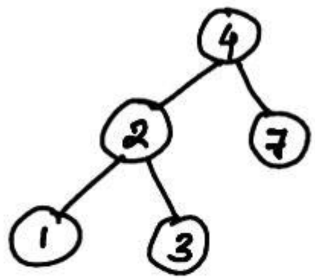
TC → $O(\log_2 n)$ avg, $O(n)$ worst

SC → $O(n)$

code

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root==NULL)  return NULL;
        if(root->val == val) return root;
        if(root->val < val) return searchBST(root->right, val);
        return searchBST(root->left, val);
    }
};
```

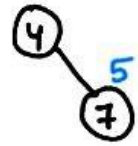
29 Insert into BST



val = 5.

1) 

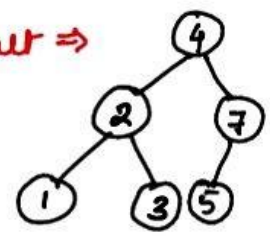
As $5 > 4$, go to RST

2) 

As $5 < 7$, go to LST

3) • As LST of 7 is null, create node with value = 5. 5

• Link 5 as LST of 7.

4) Result ⇒ 

TC → $O(\log_2 n)$ (avg), $O(n)$ (worst)

SC → $O(1)$

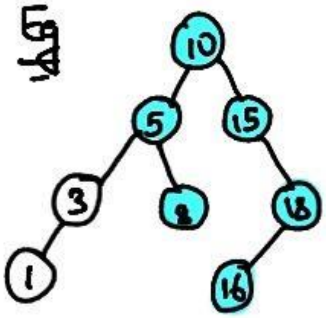
Code →

```cpp
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* node, int val) {
        if(node==NULL){
            return new TreeNode(val);
        }
        if (val < node->val) {
            node->left = insertIntoBST(node->left, val);
        }
        else {
            node->right = insertIntoBST(node->right, val);
        }
        return node;
    }
};
```

(30) Range sum of BST

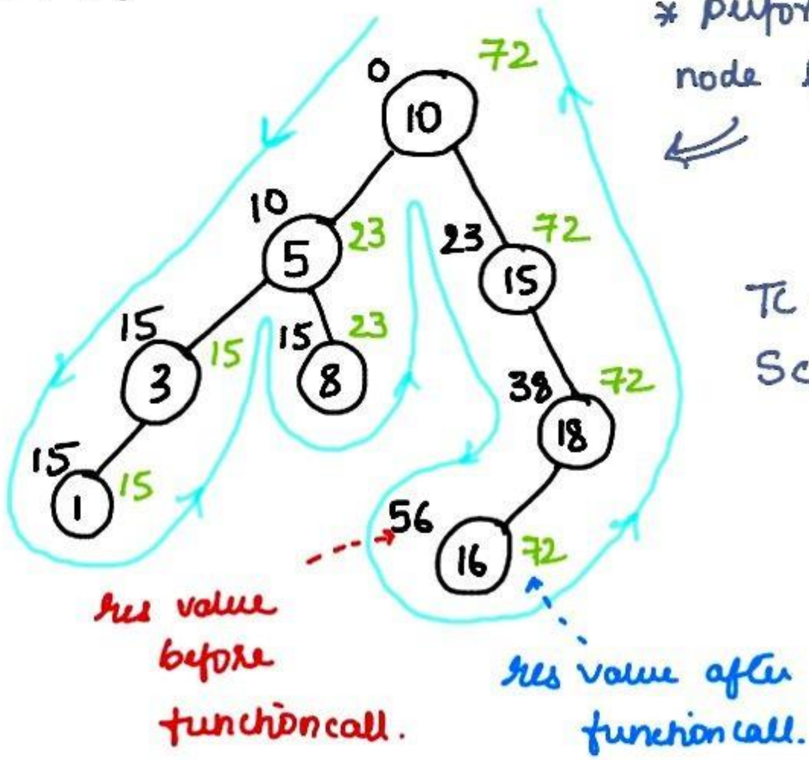
given a root node & interval [X,Y], find sum of all nodes that lies in [X,Y].

Ex



range → [5, 18]

sum = 72



* perform addition if node lies b/w [X,Y]

TC → O(n)

SC → O(n)

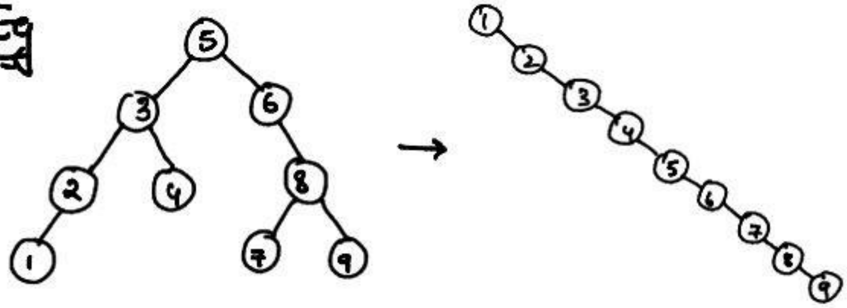
Code →

```
class Solution {
public:
    void sumUtil(TreeNode* root, int low, int high, int &res){
        if(root==NULL)  return;
        if(root->val <= high && root->val >= low){
            res += root->val;
        }
        sumUtil(root->left, low, high, res);
        sumUtil(root->right, low, high, res);
    }

    int rangeSumBST(TreeNode* root, int low, int high) {
        int res = 0;
        sumUtil(root, low, high, res);
        return res;
    }
};
```

(31) Increasing order search tree

Given a BST, create an increasing order search tree.

Eg



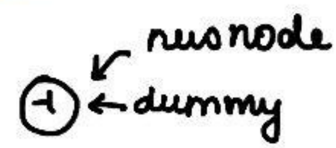
1. Perform inorder traversal.
2. create a skewed tree using elements in inorder traversal.

Code

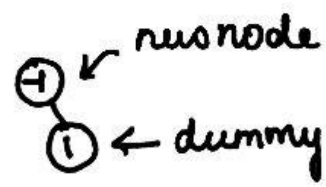
```cpp
1  class Solution {
2  public:
3      void inorder(TreeNode* root, vector<int> &res){
4          if(root==NULL)  return;
5          inorder(root->left, res);
6          res.push_back(root->val);
7          inorder(root->right, res);
8      }
9      TreeNode* increasingBST(TreeNode* root) {
10         vector<int> res;
11         inorder(root, res);
12
13         // create right skewed tree
14         TreeNode* dummy = new TreeNode(-1);
15         TreeNode* newNode = dummy;
16         for(auto it: res){
17             dummy->right = new TreeNode(it);
18             dummy = dummy->right;
19         }
20         return newNode->right;
21     }
22 };
```

Lines 16-20

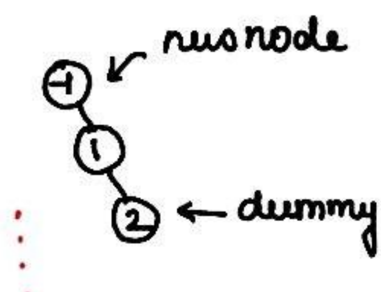
1)



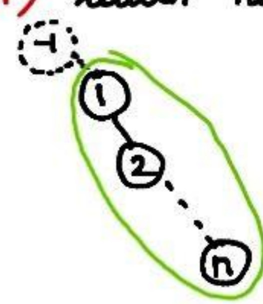
2)



3)



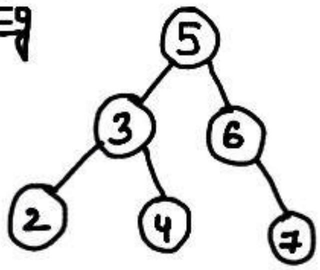
last) return newNode->right



(32) Two sum IV - Input is a BST

↳ return true if sum of any 2values == k

Eg



① Perform Inorder & Store in array

② use 2-pointer approach

k = 9 ⇒

	0	1	2	3	4	5
v =	2	3	4	5	6	7
	f					r

as v[f] + v[r] == k , return true, else f++ or r--
as per sum & k.

code →

```
class Solution {
public:
    void inorder(TreeNode* root, vector<int> &res){
        if(root==NULL)  return;
        inorder(root->left, res);
        res.push_back(root->val);
        inorder(root->right, res);
    }
    bool findTarget(TreeNode* root, int k) {
        vector<int> res;
        inorder(root, res);
        int front = 0;
        int rear = res.size()-1;
        while(front<rear){
            if(res[front]+res[rear]==k) return true;
            if(res[front]+res[rear]>k)  rear--;
            else front++;
        }
        return false;
    }
};
```

TC → O(n)+O(n)

SC → O(n)

D11 33 Delete Node in BST

given root of BST & a target node, delete the target node & return the tree.

Cases →

Tc →

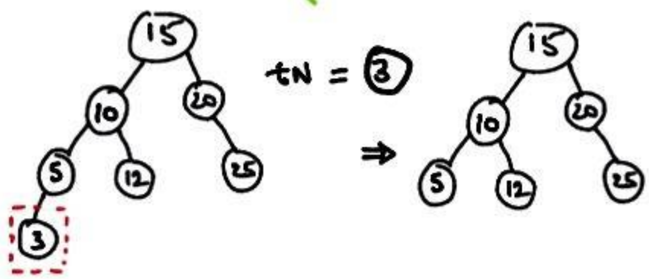
Avg ⇒ $O(\log n)$

worst ⇒ $O(n)$

SC → $O(h)$

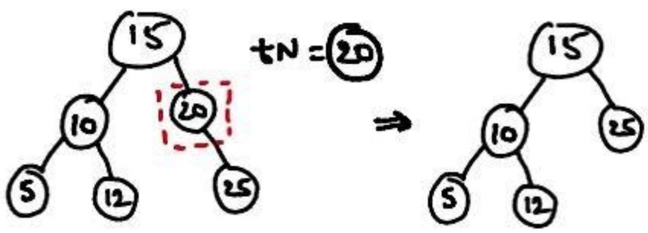
① If target node is leaf →

then simply delete it



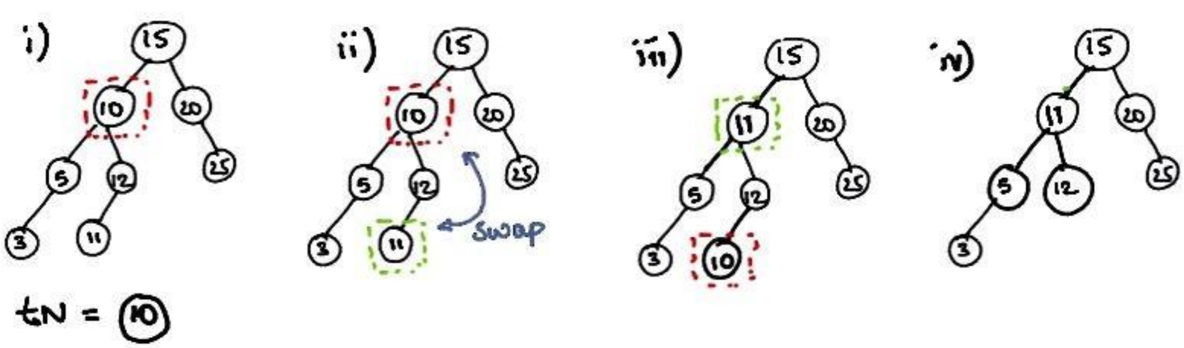
② If target node has 1 child →

then remove node & return the subtree



③ If target node has 2 children →

then go to right child's left subtree & swap its value with target node & then delete it.



Code

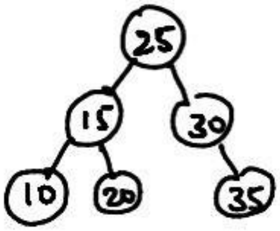
```cpp
class Solution {
public:
    TreeNode* findleftmostNode(TreeNode* root){
        while(root->left!=NULL)
            root = root->left;
        return root;
    }

    TreeNode* deleteNode(TreeNode* root, int key) {

        if(root==NULL)  return NULL;

        if(root->val > key)
            root->left = deleteNode(root->left, key);

        else if(root->val < key)
            root->right = deleteNode(root->right, key);

        else {  // root->val == key
            if(root->left == NULL && root->right == NULL){
                root = NULL;
                return root;
            }
            if(root->left != NULL && root->right == NULL){
                root = root->left;
                return root;
            }
            if(root->right != NULL && root->left == NULL){
                root = root->right;
                return root;
            }

            // finding left most node in right subtree
            TreeNode* temp = findleftmostNode(root->right);

            //swapping root's value with left most node's val
            int tempVal = root->val;
            root->val = temp->val;
            temp->val = tempVal;

            // performing delete in right subtree
            root->right = deleteNode(root->right, key);
            return root;
        }
        return root;
    }
};
```

(34) Inorder Successor of BST

Given root, find inorder successor of given node

↳ the element just after the node in inorder traversal.

Eg



n = 15 o/p → 20

n = 35 o/p → null.

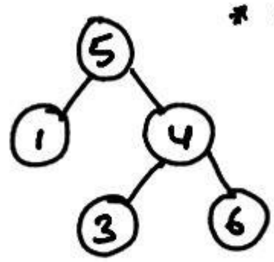
Code →

```
class Solution{
  public:

    void inorder(Node *root, vector<Node*> &res){
        if(root==NULL)  return;
        inorder(root->left, res);
        res.push_back(root);
        inorder(root->right, res);
    }

    Node * inOrderSuccessor(Node *root, Node *x)
    {
        vector<Node*> res;
        inorder(root, res);
        for(int i=0; i<res.size(); i++){
            if(res[i]==x && i<res.size()-1){
                return res[i+1];
            }
        }
        return NULL;
    }
};
```

D12 35 Validate BST — given a root node, return true if it is valid BST

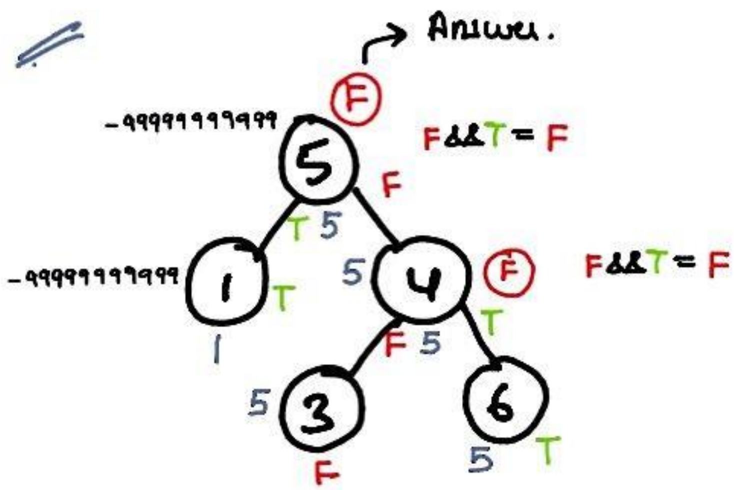
* Every value should be less than previous one in Inorder traversal

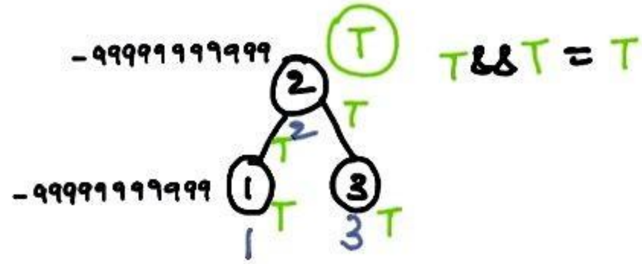
Eg.



* prev = -99999999999

false





→ Return true on NULL nodes

→ Check for left subtree

→ previous value gets updated before checking Right subtree & after checking left subtree

→ if currVal <= previous then return false

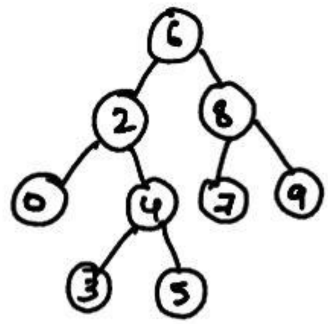
→ return true if both LST & RST are BST

code

```
class Solution {
public:
    bool isBST(TreeNode* root, long int &prev){
        if(root==NULL)  return true;
        bool isLeftBalanced = isBST(root->left, prev);
        if(root->val <= prev)   return false;
        prev = root->val;
        bool isRightBalanced = isBST(root->right, prev);
        return isLeftBalanced && isRightBalanced;
    }

    bool isValidBST(TreeNode* root) {
        long int prev = -99999999999;
        return isBST(root, prev);
    }
};
```

(36) LCA of BST →

Eg.



$P=2,\ q=8$

if currNode > both p & q
then LCA lies in LST

if currNode < both p & q
then LCA lies in RST

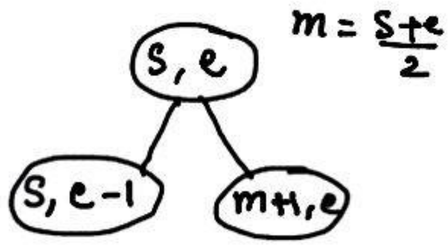
in every other case the currNode is LCA as p & q will be on

	worst	Avg
TC →	O(n)	O(logn)
SC →	O(n)	

code

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL)  return NULL;

        if(root->val < p->val && root->val < q->val){
            return lowestCommonAncestor(root->right, p, q);
        }
        else if(root->val > p->val && root->val > q->val){
            return lowestCommonAncestor(root->left, p, q);
        }
        else {
            return root;
        }
    }
};
```

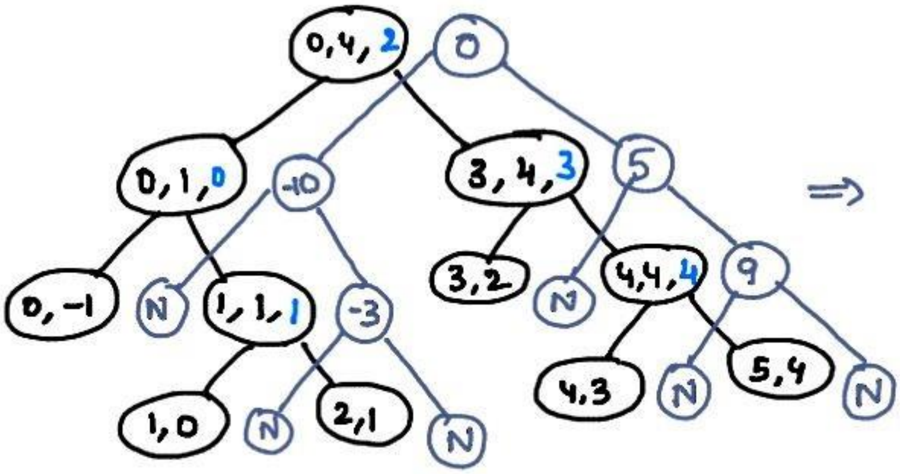
(37) Convert Sorted Array to BST



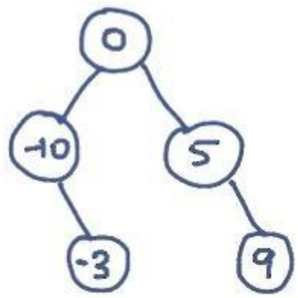
Given sorted array, create a BST

Eg [-10, -3, 0, 5, 9]
0 1 2 3 4

start, end, mid



⟹



Code →

```
class Solution {
public:
    TreeNode* createBST(vector<int>&nums, int start, int end){
        if(start>end)   return NULL;

        int mid = (start + end)/2;
        TreeNode* root = new TreeNode(nums[mid]);

        root->left = createBST(nums, start, mid-1);
        root->right = createBST(nums, mid+1, end);
        return root;
    }

    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return createBST(nums, 0, nums.size()-1);
    }
};
```

D13 (38) Construct Binary Tree from Pre & Inorder traversal

$TC \rightarrow O(n^2)$

$SC \rightarrow O(n)$

Eg

0 1 2 3 4

pre = [3, 9, 20, 15, 7]

in = [9, 3, 15, 20, 7]

* for every node in Pre, the corresponding LST & RST are in In

is 3 → [9, (3), 15, 20, 7]

LST CI RST

0 1 2 3 4

LST = (instart, CI − 1)

RST = (CI+1, inend)

CI = index of pre[0] in In

0 1 2 3 4

pre = [3, 9, 20, 15, 7]

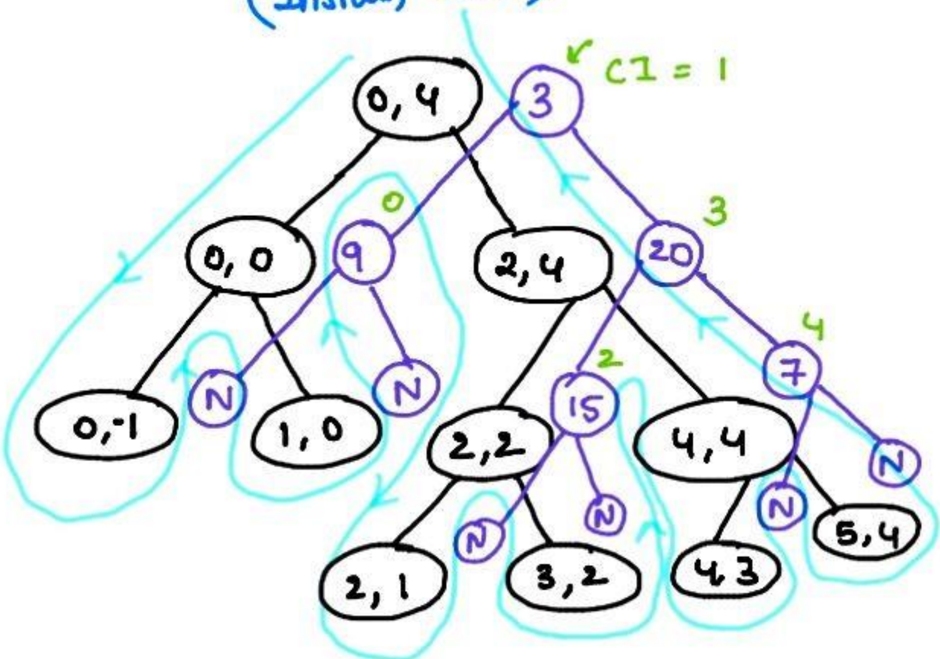
in = [9 3, 15, 20, 7]

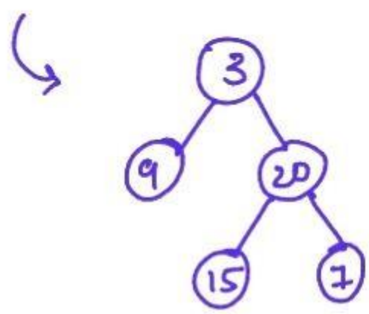
(1) for preorder index = 0, inorder boundary = [0, 4]

(2) find root value in Inorder array & it's index is currIndex

(3) if instart > CI−1 or CI+1 < inend return NULL

(Instart, Inend)





To reduce TC

we can use hashtable to find indexing

$TC \rightarrow O(n)$

$SC \rightarrow O(n) + O(n)$

code →

```cpp
class Solution {
public:
    TreeNode* constructTree(vector<int>& preorder, unordered_map<int, int> &mp,
    int start, int end, int &preIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(preorder[preIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[preorder[preIdx]];
        // increment preIdx to find next root
        preIdx++;

        // recursively call LST & RST
        root->left = constructTree(preorder, mp, start, currIdx-1, preIdx);
        root->right = constructTree(preorder, mp, currIdx+1, end, preIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int,int> mp = populate(inorder);
        int preIdx = 0;
        return constructTree(preorder, mp, 0, inorder.size()-1, preIdx);
    }
};
```

(39) Construct Binary Tree from In & Postorder traversal

Intuition is same as previous program, only changes are

→ traverse from last element in post order array

→ process RST & then go for LST

code →

```
class Solution {
public:

    TreeNode* constructTree(vector<int>& postorder, unordered_map<int, int> &mp,
    int start, int end, int &postIdx ){

        if(start>end)   return NULL;
        TreeNode* root = new TreeNode(postorder[postIdx]);

        // find currIndex as per inorder array
        int currIdx = mp[postorder[postIdx]];
        postIdx--;

        // recursively call RST & LST
        root->right = constructTree(postorder, mp, currIdx+1, end, postIdx);
        root->left = constructTree(postorder, mp, start, currIdx-1, postIdx);
        return root;
    }

    unordered_map<int,int> populate(vector<int>&inorder){
        unordered_map<int,int> mp;
        for(int i=0; i<inorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return mp;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int,int> mp = populate(inorder);
        int postIdx = postorder.size()-1;
        return constructTree(postorder, mp, 0, inorder.size()-1, postIdx);
    }
};
```

(40) Construct BST from Preorder traversal

[8, 5, 1, 7, 10, 12]

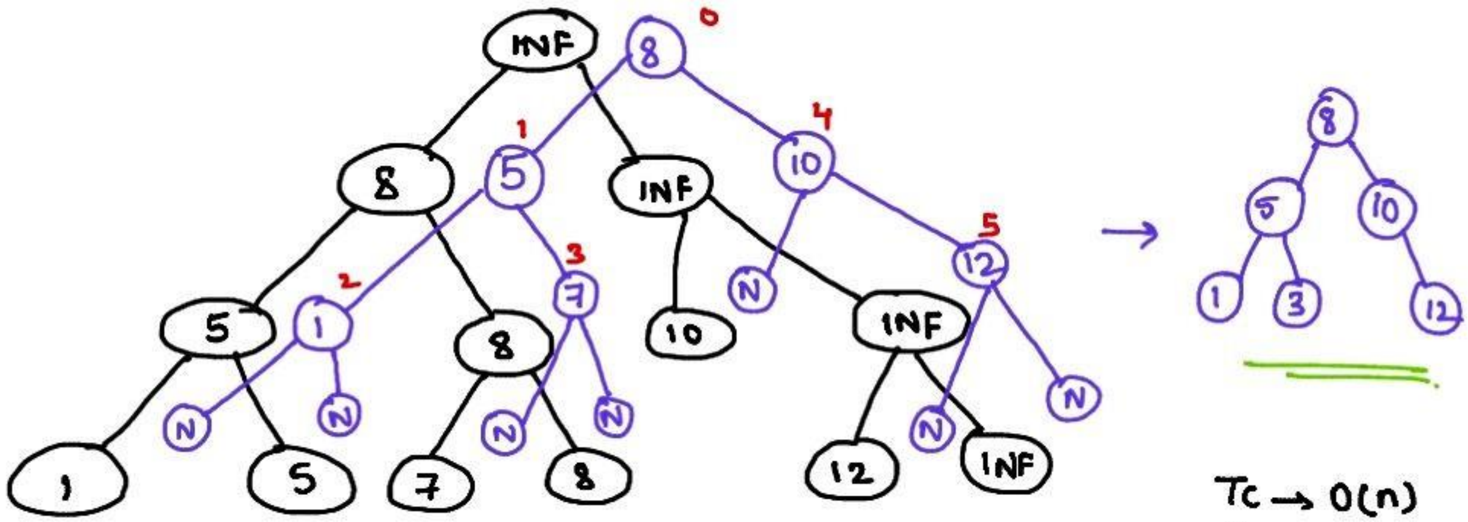
Tc → O(n log n) (due to sorting)

Approach 1 → Sort given Preorder to get Inorder, now similar to problem (38).

Approach 2 →

[8, 5, 1, 7, 10, 12]
 0 1 2 3 4 5

Boundary of LST → val
RST → boundVal → initially (INF)



Tc → O(n)
Sc → O(n)

code →

```cpp
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, int &preIdx, int boundary){
        if(preIdx >= preorder.size() || preorder[preIdx] >= boundary)
            return NULL;

        // create root using preIdx
        TreeNode* root = new TreeNode(preorder[preIdx]);
        preIdx++;

        // recursively call LST & RST
        root->left = buildTree(preorder, preIdx, root->val);
        root->right = buildTree(preorder, preIdx, boundary);
        return root;
    }

    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int preIdx = 0;
        return buildTree(preorder, preIdx, 1001);
    }
};
```

Graph

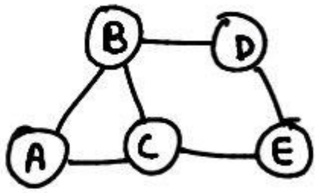
– Karun Karthik

Contents

Graphs

Graph G is a pair (V, E) where V is set of vertices & E is set of edges. $n = |V|$ & $e = |E|$

Eg



$V = \{A, B, C, D, E\}$ $n = 5$

$E = \{AB, AC, BC, BD, CE, DE\}$ $e = 6$

Applications →

Google maps → To find shortest routes

Social network → user, connection
↑ vertex ↑ edge

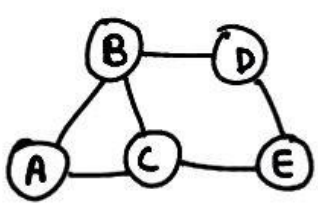
Representation →

Adj. matrix
SC → $O(V^2)$

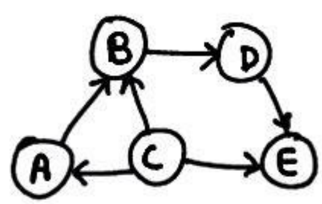
Adj. list
SC → $O(V+E)$

Types →

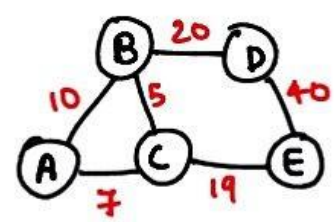
1) Undirected



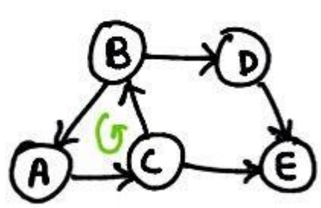
2) Directed



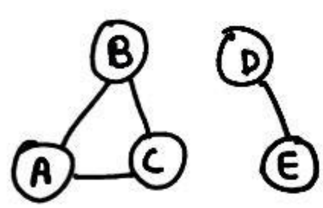
3) Weighted



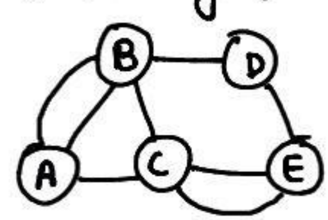
4) Cyclic



5) Disconnected



6) Multigraph

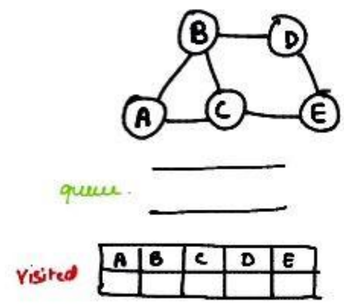


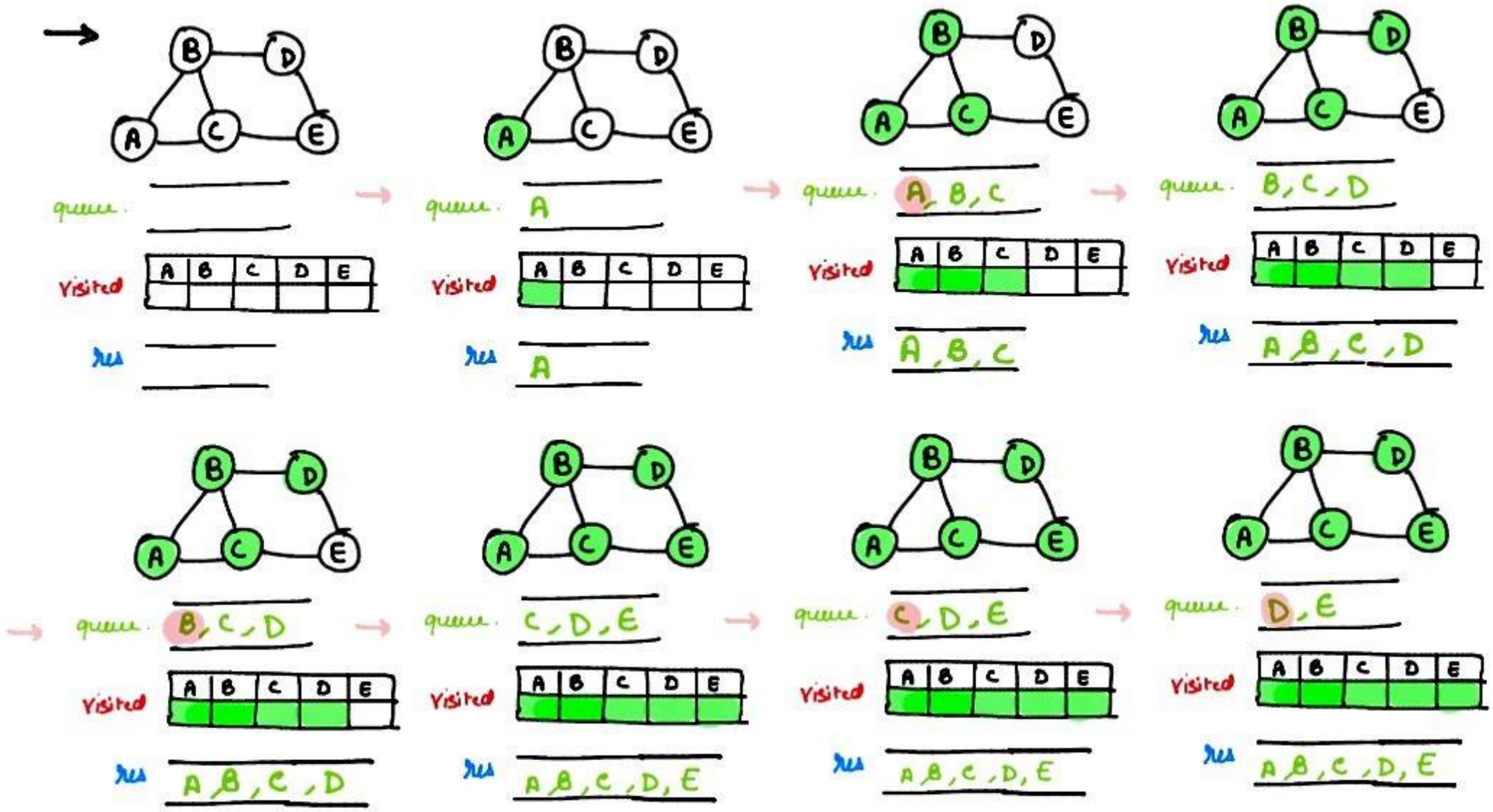
[no self loops]

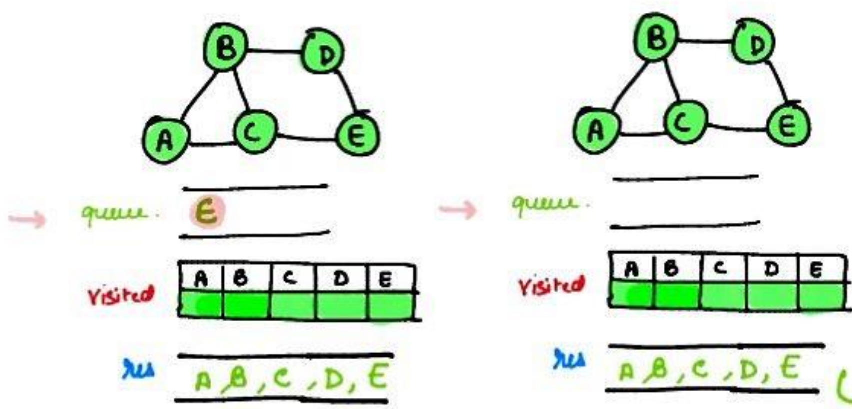
Graph Traversal

(a) BFS → visit each and every vertex in a defined order.

- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into queue
- if no neighbours then pop.
- repeat till queue is empty







↳ Return res.

TC → $O(V+E)$

SC → $O(V)$

Code

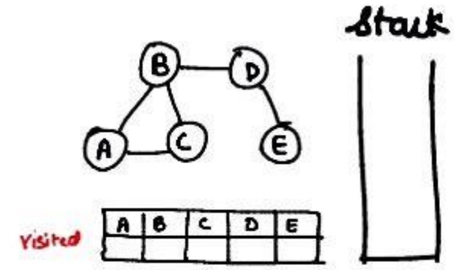
```cpp
class Solution {
  public:

    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        vector<int>ans;
        vector<int>vis(V,0);
        queue<int>q;
        q.push(0);

        while(!q.empty()){
            int curr = q.front();
            q.pop();
            vis[curr]=1;
            ans.push_back(curr);
            for(auto it:adj[curr]){
                if(vis[it]==0){
                    vis[it]=1;
                    q.push(it);
                }
            }
        }
        return ans;
    }
};
```

Applications → [BFS]

1. Shortest path
2. Min. spanning tree for unweighted graph
3. Cycle detection
4. GPS
5. Social network.

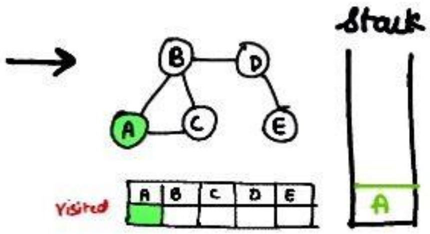
⑥ DFS →

- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into stack
- if no neighbours then pop.
- repeat till stack is empty

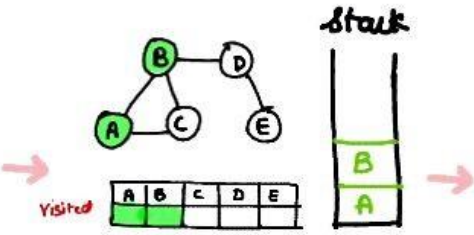


res

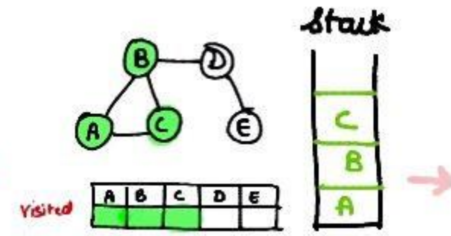
→



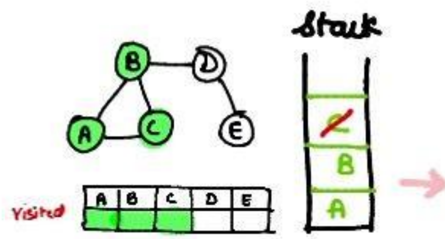
res A,



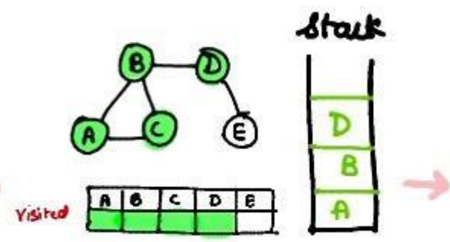
res A, B



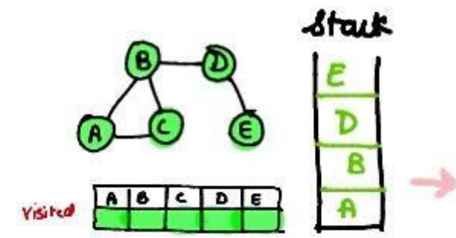
res A, B, C



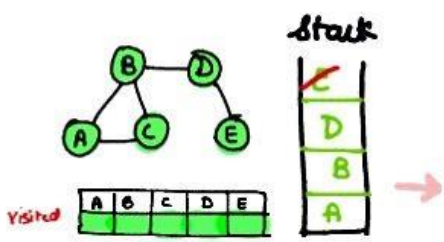
res A, B, C



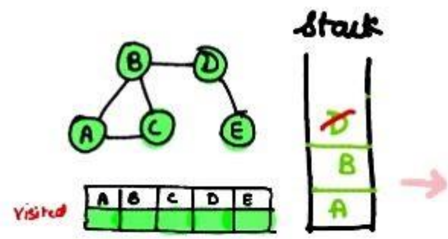
res A, B, C, D



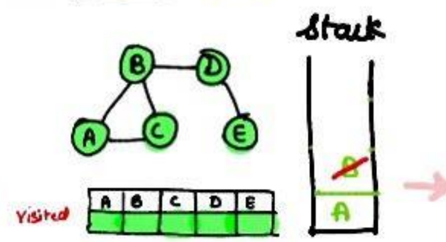
res A, B, C, D, E



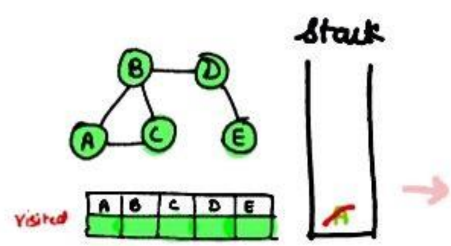
res A, B, C, D, E



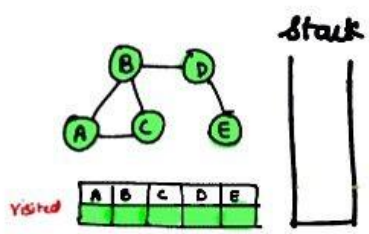
res A, B, C, D, E



res A, B, C, D, E



res A, B, C, D, E



res A, B, C, D, E → Return res.

TC → O(V+E)

SC → O(V)

code

```cpp
class Solution {
  public:

    void dfs(vector<int>&ans, vector<int>&vis, int node, vector<int>adj[]){
        vis[node] = 1;
        ans.push_back(node);
        for(auto it:adj[node]){
            if(!vis[it]){
                vis[it] = 1;
                dfs(ans, vis, it, adj);
            }
        }
    }
    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<int>vis(V,0);
        for(int i=0; i<V; i++){
            if(vis[i]==0)
            dfs(ans, vis, i, adj);
        }
        return ans;
    }
};
```

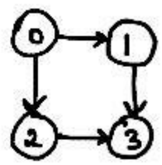
Applications → [DFS]

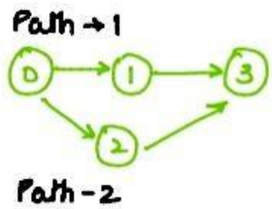
1. Path finding
2. Cycle detection
3. Topological sort
4. Finding strongly connected components.

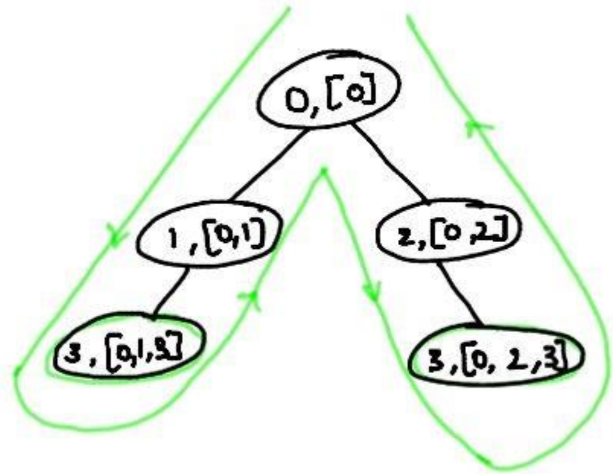
① All paths from src to target

Given a directed acyclic graph, return all paths from node 0 to node n-1.

Eg







Code →

```cpp
class Solution {
public:
    void findAllPaths(vector<vector<int>>&graph, int currNode, vector<bool>&visited,
                      int n, vector<int> &currPath, vector<vector<int>>&res){

        if(currNode==n-1){
            res.push_back(currPath);
            return;
        }

        if(visited[currNode]==true) return;

        // backtrack for every node
        visited[currNode] = true;

        for(auto neighbour: graph[currNode]){
            currPath.push_back(neighbour);
            findAllPaths(graph, neighbour, visited, n, currPath, res);
            currPath.pop_back();
        }

        visited[currNode] = false;
    }

    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        vector<vector<int>> res;
        vector<int> currPath;
        int n = graph.size();
        vector<bool> visited(n);

        // traversing from 0 node
        currPath.push_back(0);

        findAllPaths(graph, 0, visited, n, currPath, res);
        return res;
    }
};
```

TC → O(V+E)

V → vertices

E → edges

SC →

Recursive stack + Result

② Flood Fill → If 0 then blocked, else fill it with given color.

	0	1	2
0	1	1	1
1	1	1	0
2	1	0	1

⇨

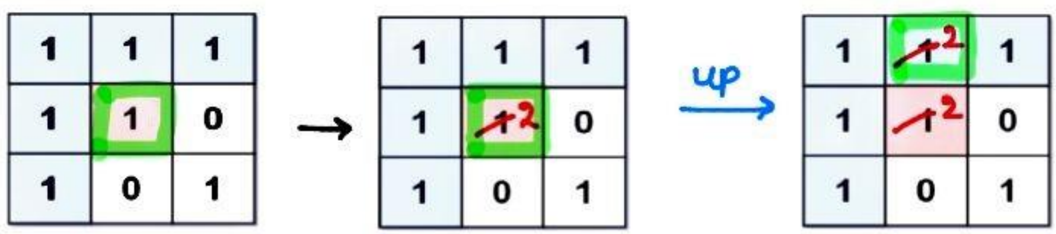
2	2	2
2	2	0
2	0	1

* perform flood fill from given index in all 4-directions & the cell should have same color as src

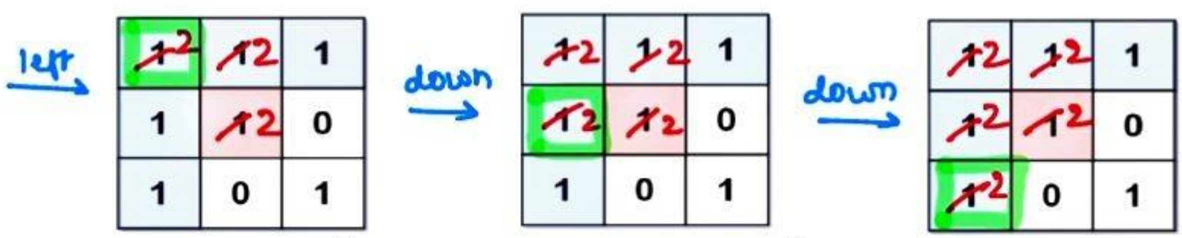
✓ lets follow the order to fill → UP, DOWN, LEFT, RIGHT

Eg In above case starting point is (1,1) & value = 1 so

(*)



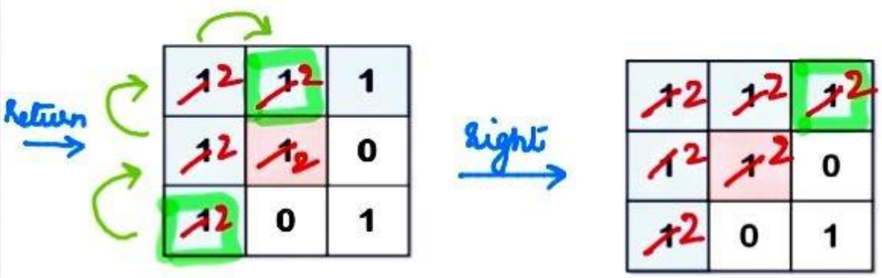
← from here up is not possible & down is filled so left



from here up not possible so down

from here up not possible so down

→ no direction is possible so return



no other way possible

→

Result

2	2	2
2	2	0
2	0	1

Code

```cpp
class Solution {
public:
    void floodFiller(vector<vector<int>>& image, int i, int j,
    int m, int n, int currColor, int newColor)
    {
        if(i<0 || i>=m || j<0 || j>= n || image[i][j] == newColor
            || image[i][j] != currColor)
            return;

        image[i][j] = newColor;
        floodFiller( image, i-1, j, m, n, currColor, newColor);
        floodFiller( image, i+1, j, m, n, currColor, newColor);
        floodFiller( image, i, j-1, m, n, currColor, newColor);
        floodFiller( image, i, j+1, m, n, currColor, newColor);
    }

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
    int sc, int newColor)
    {
        int m = image.size();
        int n = image[0].size();
        int currColor = image[sr][sc];
        floodFiller(image, sr, sc, m, n, currColor, newColor);
        return image;
    }
};
```

TC → $O(mn)$

SC → $O(h)$

↳ Recursive stack

③ Number of Islands → Given grid of 1 (land) & 0 (water), return no. of islands.

Eg

```
     0  1  2  3  4
0  [[1, 1, 0, 0, 0],
1   [1, 1, 0, 0, 0],
2   [0, 0, 1, 0, 0],
3   [0, 0, 0, 1, 1]]
```

- Always start dfs only if value = 1 & change its value to 0, so it cannot be visited again
- if initial value = 0 then skip.
- initially ans = 0

- let's start from (0,0) & try moving U, D, L, R

→ the traversal goes in this order

(0,0) → (1,0) → (1,1) → (0,1) i.e

& update ans.

```
[[1, 1, 0, 0, 0],
 [1, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 1]]
```

ans = $\cancel{0}$ 1.

→ now grid becomes

```
     0  1  2  3  4
0  [[0, 0, 0, 0, 0],
1   [0, 0, 0, 0, 0],
2   [0, 0, 1, 0, 0],
3   [0, 0, 0, 1, 1]]
```

- now, we can skip every entry from (1,0) to (2,1) as they are 0s.
- now start from (2,2), as U, D, L, R is not possible, set its value = 0 & update ans.

ans = $\cancel{1}$ 2.

→ now grid becomes

```
     0  1  2  3  4
0  [[0, 0, 0, 0, 0],
1   [0, 0, 0, 0, 0],
2   [0, 0, 0, 0, 0],
3   [0, 0, 0, 1, 1]]
```

- now, we can skip every entry from (2,3) to (3,2) as they are 0s.
- now start from (3,3), it goes as follows (3,3) → (3,4)
- further traversal from (3,4) is not possible

ans = $\cancel{2}$ 3. $\boxed{\text{Ans} = 3}$

Code

```cpp
class Solution {
public:
    void countIsland(vector<vector<char>>& grid, int currRow, int currCol, int row, int col){
        if(currRow<0 || currRow>=row || currCol<0 || currCol>=col || grid[currRow][currCol]=='0')
            return;

        grid[currRow][currCol] = '0';
        countIsland(grid, currRow-1, currCol, row, col);
        countIsland(grid, currRow+1, currCol, row, col);
        countIsland(grid, currRow, currCol-1, row, col);
        countIsland(grid, currRow, currCol+1, row, col);
    }

    int numIslands(vector<vector<char>>& grid) {
        int ans = 0;
        int row = grid.size();
        int col = grid[0].size();

        for(int currRow = 0; currRow < row; currRow++)
            for(int currCol = 0; currCol < col; currCol++)
                if(grid[currRow][currCol]=='1'){
                    ans++;
                    countIsland(grid, currRow, currCol, row, col);
                }

        return ans;
    }
};
```

TC $\rightarrow$ $O(mn)$ Avg case

$O(m^2n^2)$ Worst case

(4) Max Area of the Island

* Intuition is same as previous problem.
* Minor Tweak to count number of 1s in island.
* Once entire island Traversal is done,
  compare for max Area of island.

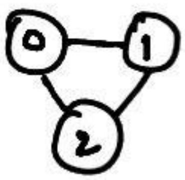
TC → O(mn) Avg case.

code →

```
class Solution {
public:
    int findArea(vector<vector<int>>& grid, int currRow, int currCol, int m, int n){
        if(currRow<0 || currCol<0 || currRow>=m || currCol>=n || grid[currRow][currCol]==0)
            return 0;

        grid[currRow][currCol]=0;

        // this is for single cell where we started traversing
        int count = 1;
        count += findArea(grid, currRow-1, currCol, m, n);
        count += findArea(grid, currRow+1, currCol, m, n);
        count += findArea(grid, currRow, currCol-1, m, n);
        count += findArea(grid, currRow, currCol+1, m, n);
        return count;
    }
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int ans = 0;
        for(int currRow = 0; currRow<m; currRow++)
            for(int currCol = 0; currCol<n; currCol++){
                if(grid[currRow][currCol]==1){
                    ans = max(ans, findArea(grid, currRow, currCol, m, n));
                }
            }
        return ans;
    }
};
```

(5) Find if path exist in graph.

Given src, dest, no. of nodes & set of edges, find if path exist b/w src & dest.

Eg  → [[0,1], [1,2], [2,0]] edges.

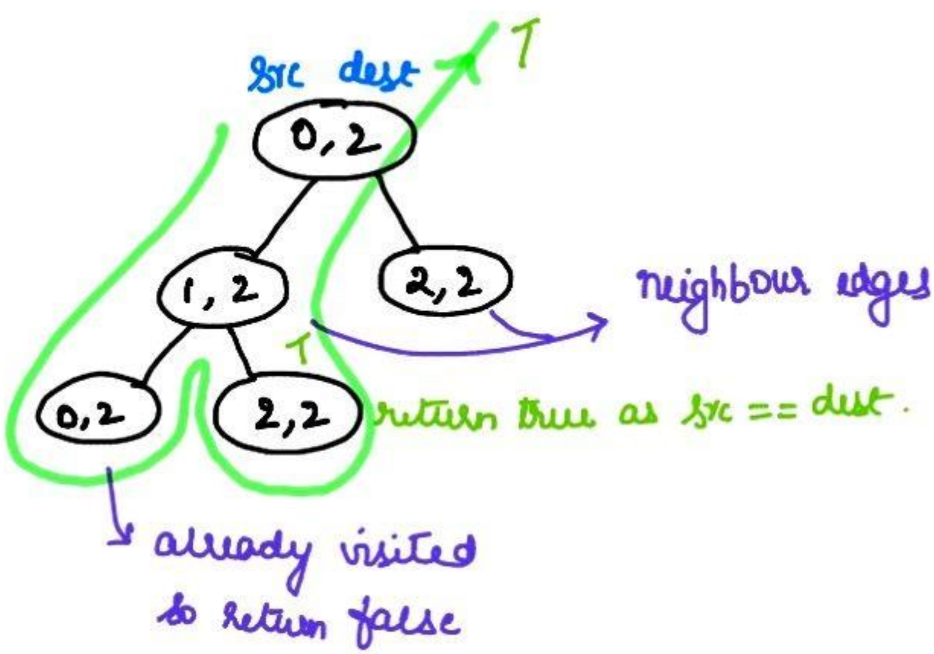
n = 3 edges = [[0,1], [1,2], [2,0]] src = 0, dest = 2.

1) Create a graph using adj list rep. [[1,2], [0,2], [1,0]]
   0 1 2

2) Perform dfs

[[1,2], [0,2], [1,0]]
0 1 2

T ~~F~~	T ~~F~~	F
0	1	2

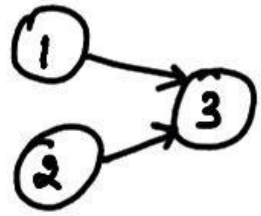


Code →

```cpp
class Solution {
public:
    bool validPath(int n, vector<vector<int>>& edges, int src, int dest) {

       vector<vector<int>>graph(n);
        for(int i=0;i<edges.size();i++)
        {
            int v1 = edges[i][0];
            int v2 = edges[i][1];
            graph[v1].push_back(v2);
            graph[v2].push_back(v1);

        }
        vector<bool>vis(n,false);
        return pathExist(src,  dest, graph, vis);
    }

    bool pathExist(int src , int dest,vector<vector<int>>&graph,vector<bool>&vis){

        if(src==dest)return true;

        vis[src]=true;

        for(int i=0;i<graph[src].size();i++)
            if(vis[graph[src][i]]==false)
                if(pathExist(graph[src][i],dest,graph,vis)==true)
                    return true;

        return false;
    }
};
```

⑥ Find the town judge

n = 3 , trust = [[1,3], [2,3]] →



* Indegree of town judge = n-1
  & outdegree = 0.

✓ Create 2 arrays

	0	1	2	3
outdegree	~~0~~ 1	0	~~0~~ 1	0
indegree	0	0	0	~~0~~ ~~1~~ 2

for [1, 3]
indegree of 1 ↑
outdegree of 3 ↑

for [2,3]
indegree of 2 ↑
outdegree of 3 ↑

→ traverse both indegree & outdegree
if indegree == 0 &&
outdegree == n-1
then return that vertex

code

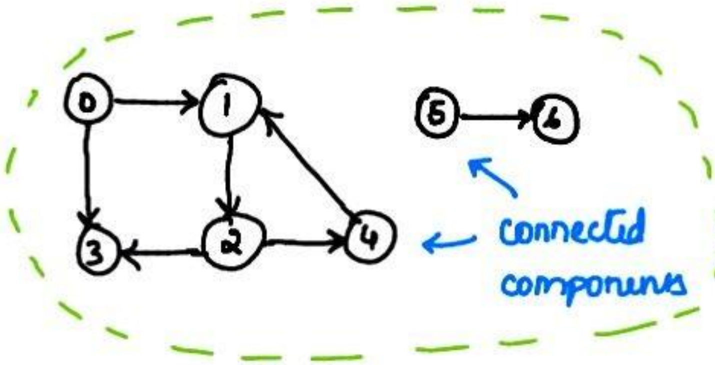
```
class Solution {
public:
    int findJudge(int n, vector<vector<int>>& trust) {
        vector<int>indegree(n+1,0);
        vector<int>outdegree(n+1,0);
        for(int i=0;i<trust.size();i++)
        {
            int v1 = trust[i][0];
            int v2 = trust[i][1];
            outdegree[v1]+=1;
            indegree[v2]+=1;
        }
        for(int i=1;i<=n;i++)
        {
            if(outdegree[i]==0 && indegree[i]==n-1)
                return i;
        }
        return -1;
    }
};
```

7 Detect cycle in a directed graph

Consider a graph with 'n' vertices labelled as [0..n-1]

Eg $n = 7$ [0, 1, 2, 3, 4, 5, 6]

Graph →



* To detect cycle, check for backedge.

Let's start dfs from 0 vertex.

* At every vertex, check if its already visited, if already visited then check if it is present in recursive stack.

If present, then it indicates back edge → Returns True

* If vertex is not visited then mark it in visited array & recursive stack

Visited → { 0, 1, 2, 3, 4

Recursive stack → {0, 1, 2, ~~3~~, 4

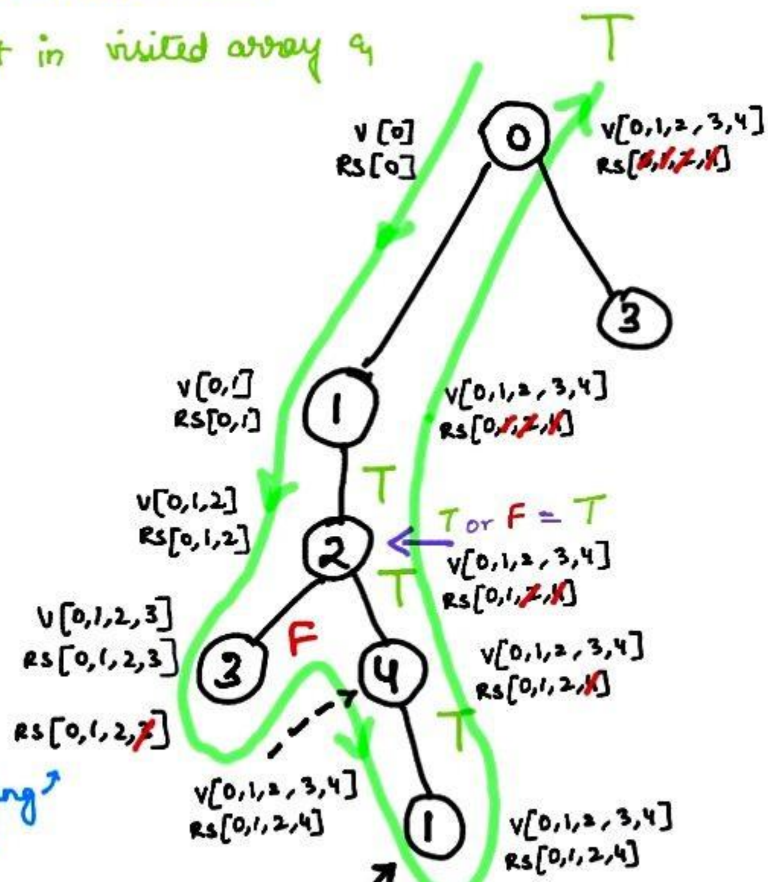
* At 3 vertex, there's no neighbour & no cycle is detected so return F.

Before returning, undo change made in Recursive stack by popping it.

Visited → { 0, 1, 2, 3, 4, 1 } &

Recursive stack → {0, 1, 2, ~~3~~, 4 }

1 is already present in recursive stack so return true.



already visited & present in RS so returns True

Code

$Tc \to O(V+E)$

$Sc \to O(V)$

```cpp
class Solution {
  public:
    bool dfs(int node, vector<int>&vis, vector<int>&rs, vector<int> adj[])
    {
        vis[node]=1;
        rs[node]=1;
        for(auto it:adj[node])
        {
            if(vis[it]==0){
                if(dfs(it,vis,rs,adj))
                    return true;
            }
            else if(rs[it]==1)
                return true;
        }
        rs[node]=0;
        return false;
    }
    bool isCyclic(int V, vector<int> adj[]) {

        vector<int>vis(V,0);
        vector<int>rs(V,0);

        for(int i=0;i<V;i++)
        {
            if(vis[i]==0)
            if(dfs(i,vis,rs,adj))
                return true;
        }
        return false;
    }
};
```

Kahn's Algorithm → To find topological Ordering

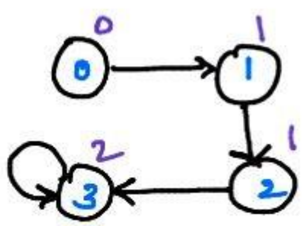
↓ can be used to find cycle using BFS.

① Find indegree of every vertex in graph & answer = 0

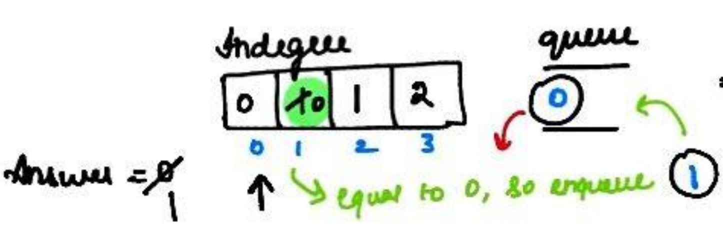
② If indegree of vertex is 0, then push into queue & do bfs till queue is not empty & while doing bfs decrease the indegree of neighbour by 1. if indegree of neighbour = 0, then enqueue & increment answer by 1

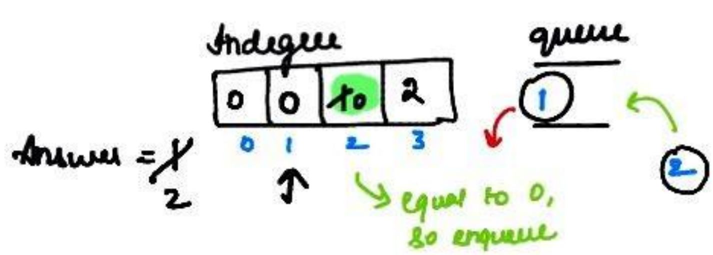
③ If answer != no. of vertices then cycle is present.

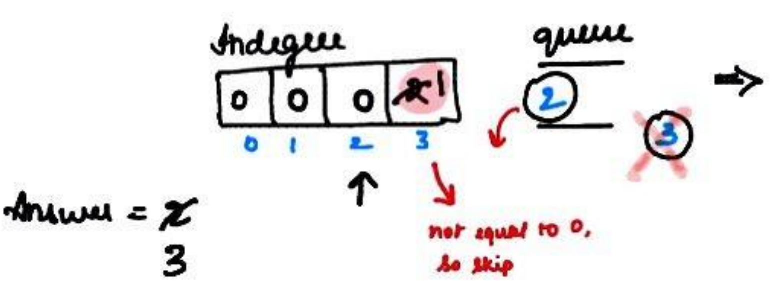
Eg



→ As indegree of ⓪ is 0, we push into queue & do bfs till queue is not empty.







Answer = 3

No of vertices = 4

∴ cycle present.

code

```cpp
class Solution{
  public:
    bool isCyclic(int V, vector<int> adj[]) {

      vector<int>indegree(V,0);
      for (int i = 0; i <V; i++)
        for(int it : adj[i])
          indegree[it]++;

      queue<int>q;
      int ans = 0;
      unordered_set<int>vis;

      for (int i=0;i<V;i++)
      {
          if(indegree[i]==0){
              q.push(i);
              ans+=1;
          }
      }

      while(!q.empty())
      {
          int currvertex = q.front();
          q.pop();
          if(vis.find(currvertex)!=vis.end())
              continue;
          vis.insert(currvertex);
          for(int neighbour:adj[currvertex])
          {
              indegree[neighbour]-=1;
              if(indegree[neighbour]==0)
               {
                   q.push(neighbour);
                   ans+=1;
               }
          }
      }
      if(ans==V)  return false;
      return true;
    }
};
```

8 Topological sort

→ use Kahn's algorithm. & add node to result while performing dfs.

Code →

TC → $O(V + E)$

SC → $O(V)$

```cpp
class Solution
{
    public:
    vector<int> topoSort(int V, vector<int> adj[])
    {
        vector<int> indegree(V,0) ,res;

        for(int i=0; i<V; i++)
            for(auto it:adj[i])
                indegree[it]++;

       queue<int> q;
       int ans = 0;
       unordered_set<int>vis;

       for(int i=0; i<V; i++)
       {
           if(indegree[i]==0){
               q.push(i);
               ans+=1;
           }
       }

       while(!q.empty())
       {
           int curr = q.front();
           q.pop();

           // add to res
           res.push_back(curr);

           if(vis.find(curr)!=vis.end())
               continue;

           vis.insert(curr);

           for(int neighbour: adj[curr])
           {
               indegree[neighbour]-=1;
               if(indegree[neighbour]==0)
               {
                   q.push(neighbour);
                   ans+=1;
               }
           }
       }

       return res;
    }
};
```

⑨ Course Schedule → can be solved using Kahn's algo.

Tc → O(V + E)

Sc → O(V + E)

Code →

```
class Solution {
public:
    vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
        vector<vector<int>> graph(n);
         for(auto it:pre){
             int v = it[1];
             int u = it[0];
             graph[v].push_back(u);
         }
         return graph;
    }

    bool canFinish(int n, vector<vector<int>>& pre) {
        vector<vector<int>> graph = createGraph(n, pre);
        vector<int>indegree(n,0);
        for(int i=0; i<n; i++)
            for(int it: graph[i])
                indegree[it]++;

        queue<int> q;
        int ans = 0;
        unordered_set<int> vis;

        for(int i=0; i<n; i++)
            if(indegree[i]==0){
                q.push(i);
                ans++;
            }

        while(!q.empty()){
            int currvertex = q.front();
            q.pop();
            if(vis.find(currvertex)!=vis.end())
                continue;
            vis.insert(currvertex);
            for(int neighbour: graph[currvertex]){
                indegree[neighbour]--;
                if(indegree[neighbour]==0){
                    q.push(neighbour);
                    ans++;
                }
            }
        }
        if(ans==n) return true;
        return false;
    }
};
```

(10) Course Schedule – II

pre → edges [v, u]

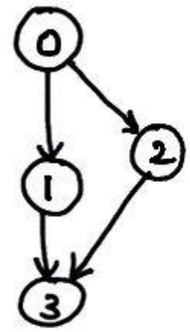
"u should be completed before v"

n → no. of courses [vertices]

Topological sort only for DAG.

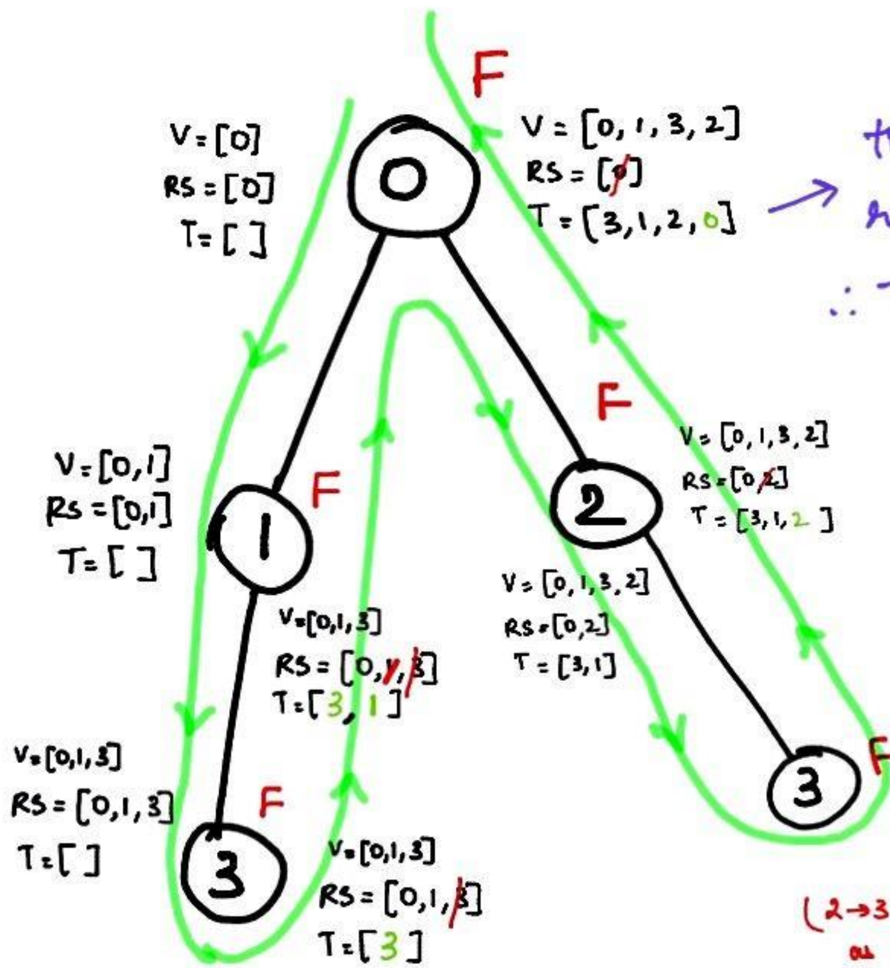
Eg n → 4 (0, 1, 2, 3)

pre → [[1,0], [2,0], [3,1], [3,2]]



Initially

V = [ ], RS = [ ], traversal = [ ]



→ topological order in reverse!

∴ T = [0,1,2,3]

(2→3 is not backedge as it is not in RS)

while returning from 3 ↑

pop 3 & push into traversal array.

return F, as no cycle is found

Code →

Tc → O(V + E)

Sc → O(V + E)

```cpp
class Solution {
public:
    bool dfs(vector<vector<int>>&graph, int i, vector<int> &vis,
    vector<int> &rs, vector<int> &traversal){

        vis[i] = 1;
        rs[i] = 1;
        for(int neighbour: graph[i]){
            if(vis[neighbour]==0){
                if(dfs(graph, neighbour, vis, rs, traversal))
                    return true;
            }
            else if(rs[neighbour]==1)   return true;
        }
        traversal.push_back(i);
        rs[i]=0;
        return false;
    }

    vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
        vector<vector<int>> graph(n);
         for(auto it:pre){
             int v = it[1];
             int u = it[0];
             graph[v].push_back(u);
         }
         return graph;
    }

    vector<int> findOrder(int n, vector<vector<int>>& pre) {
        vector<vector<int>> graph = createGraph(n, pre);
        vector<int> vis(n,0), rs(n,0), traversal;
        for(int i=0; i<n; i++){
            if(vis[i]==0)
            if(dfs(graph, i, vis, rs, traversal)) return {};
        }
        reverse(traversal.begin(), traversal.end());
        return traversal;
    }
};
```

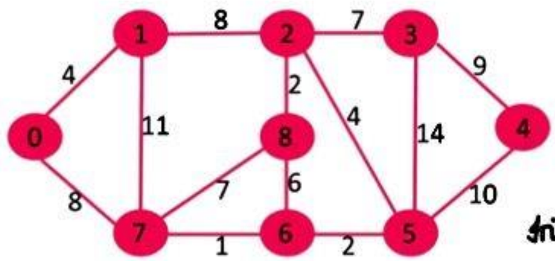
Graph - 2

- Karun Karthik

Content

(11) Dijkstra Algorithm → single source shortest path (only +ve weights)

→ Helps in finding the shortest path to every node from src node.



n = 9 (nodes from 0 to 8)
src = 1
cost away = min cost from src to every other vertex

Initially cost =

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

vis = { }

→ As it is weighted graph, we'll use priority queue (PQ) instead of normal queue. & element pushed into it will be of form (curr node, curr cost)

→ pq always pops element with least curr cost. → always calculated from src to curr node.

✓ (1,0)

vis = {1}
cost[1] = 0

⇒ now neighbours of 1 = (0,4) (7,11) (2,8) ∴ push

~~(1,0)~~ (0,4) (7,11) (2,8)

→ lowest cost among 4, 11, 8 is 4 ∴ pop it & push its neighbours.

⇒ ~~(1,0)~~ (0,4) (7,11) (2,8)

vis = {1, 0}
cost[0] = 4

⇒ now neighbours of 0 = 1 (visited), (7,12) ∴ push

~~(1,0)~~ ~~(0,4)~~ (7,11) (2,8) (7,12)

→ lowest cost is 8 ∴ pop & push its neighbours

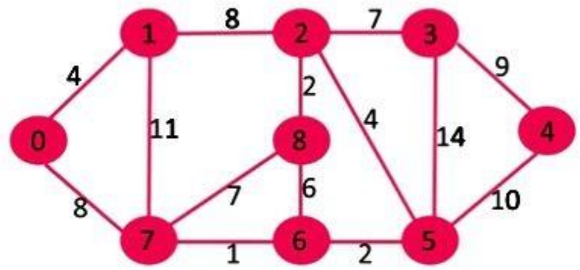
⇒ ~~(1,0)~~ ~~(0,4)~~ (7,11) (2,8) (7,12)

⇒ neighbours of 2 = 1 (visited), (8,10), (3,15), (5,12) ∴ push

vis = {1, 0, 2}
cost[2] = 8

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) (8,10), (3,15), (5,12)

→ lowest cost is 10 ∴ pop & push its neighbours



⇒

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) (8,10), (3,15), (5,12) ⇒ neighbours of 8 = 2(visited), (7,17), (6,16)
∴ push

vis = {1,0,2,8}
cost[8] = 10

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16)

↳ lowest cost = 11 ∴ pop & push its neighbours.

⇒

~~(1,0)~~ ~~(0,4)~~ (7,11) ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16) ⇒ neighbours of 7 = 0,1,8 are visited.
& (6,12) ∴ push

vis = {1,0,2,8,7}
Cost[7] = 11

~~(1,0)~~ ~~(0,4)~~ ~~(7,11)~~ ~~(2,8)~~ (7,12) ~~(8,10)~~, (3,15), (5,12), (7,17), (6,16), (6,12) ≃ (7,12), (3,15), (5,12), (7,17), (6,16), (6,12)

↳ lowest cost = 12

∴ Anything among 5,6 can be selected & pop & push its neighbours
Not 7, because its already visited & cost is < 12.

⇒

(3,15), (5,12), (7,17), (6,16), (6,12) ⇒ neighbours of 5 = (4,22), (3,26), (6,14) ∴ push

vis = {1,0,2,8,7,5}
cost[5] = 12

(3,15), ~~(5,12)~~, (7,17), (6,16), (6,12) (4,22), (3,26), (6,14) → lowest cost = 12
∴ pop & push its neighbours

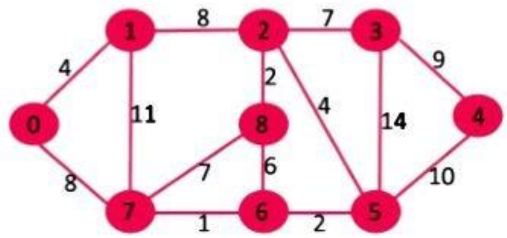
⇒ (3,15), (7,17), (6,16), (6,12) (4,22), (3,26), (6,14) ⇒ neighbours of 6 = 5, 7, 8 are visited.
∴ no push

vis = {1,0,2,8,7,5,6}
cost[6] = 12

(3,15), (7,17), (6,16), ~~(6,12)~~ (4,22), (3,26), (6,14) → next lowest is 14, but 6 is already visited.

(3,15), (7,17), (6,16), (4,22), (3,26), ~~(6,14)~~

∴ Next lowest is 15, ∴ pop & push it's neighbours



⇒ (3,15), (7,17), (6,16), (4,22), (3,26) ⇒ neighbours of 3 = 2,5 (visited) (4,24) ∴ push

vis = {1, 0, 2, 8, 7, 5, 6, 3}
cost[3] = 15

~~(3,15)~~, (7,17), (6,16), (4,22), (3,26), (4,24) → next lowest cost = 16 but 6 is already visited ∴ pop

(7,17), ~~(6,16)~~, (4,22), (3,26), (4,24) ⇒ (4,22), (3,26), (4,24) → next lowest cost = 22 ∴ pop & push its neighbours

→ next lowest cost = 17 but 7 is already visited ∴ pop

⇒ (4,22), (3,26), (4,24) ⇒ neighbours of 4 = 3, 5 (visited) ∴ no push

vis = {1, 0, 2, 8, 7, 5, 6, 3, 4}
cost[4] = 22

~~(4,22)~~, (3,26), (4,24) → next lowest cost = 24 but 4 is already visited ∴ pop

(3,26), ~~(4,24)~~ → next lowest cost = 26 but 3 is already visited ∴ pop ⇒ ~~(3,26)~~ ⇒ ∴ empty PQ.

Answer ⇒ cost =

4	0	8	15	22	12	12	11	10
0	1	2	3	4	5	6	7	8

$\boxed{\text{Dijkshas} = \text{BFS} + \text{PQ}}$

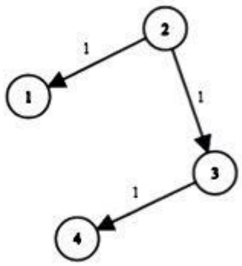
Tc → $O(V + E \log V)$

Sc → $O(V)$

Code →

```cpp
class Solution
{
    public:
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int src)
    {
        vector<int>cost(V,0);
        cost[src]=0;

        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;

        pq.push({0,src}); // {cost, node}

        while(!pq.empty())
        {
            pair<int,int>p = pq.top();
            int currCost = p.first;
            int currNode = p.second;
            pq.pop();

            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int weight = adj[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push
                pq.push({currCost + weight, neighbourNode});
            }
        }
        return cost;
    }
};
```

12 Network Delay Time



src = 2.

You are given a network of `n` nodes, labeled from `1` to `n`. You are also given `times`, a list of travel times as directed edges `times[i] = (ui, vi, wi)`, where `ui` is the source node, `vi` is the target node, and `wi` is the time it takes for a signal to travel from source to target.

We will send a signal from a given node `k`. Return the time it takes for all the `n` nodes to receive the signal. If it is impossible for all the `n` nodes to receive the signal, return `-1`.

✓. Similar to dijkstra's algo. cost =

0	0	0	0	0
0	1	2	3	4

vis = {} pq =

⇒ push (2,0) to pq. ⇒ (2,0)

⇒ (2,0) neighbours = (1,1), (3,1) ∴ push

vis = {2}
cost[2] = 0

~~(2,0)~~ (1,1) (3,1) → next lowest cost = 1 ∴ choose 1 or 3
pop & push their neighbours.

⇒ (1,1) (3,1) no new neighbours ∴ pop

vis = {2, 1}
cost[1] = 1

~~(1,1)~~ (3,1) → next lowest cost = 1
∴ pop & push their neighbours.

⇒ (3,1) neighbour = (4,2) ∴ push

vis = {2, 1, 3}
cost[3] = 1

~~(3,1)~~ (4,2) → next lowest cost = 2
∴ pop & push neighbours.

⇒ (4,2) no new neighbours ∴ pop

TC → O(V + E log V)
SC → O(V)

vis = {2, 1, 3, 4}
cost[4] = 2

~~(4,2)~~ → pq is empty.

→ check if all nodes are in visited, else return -1.

∴ cost =

0	1	0	1	2
0	1	2	3	4

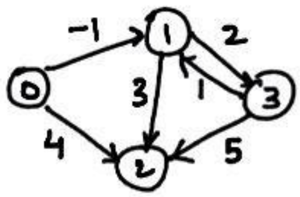
→ Return max value in cost as

Code →

```
class Solution {
public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<vector<int>>> graph = createGraph(times,n);
        return minTime(graph,n,k);
    }

    vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {

        vector<vector<vector<int>>> graph(n+1);

        for(int i=0;i<=n;i++) {
            graph.push_back({{}});
        }
        // add every edge to the graph
        for(vector<int> edge:edges) {
            int source = edge[0];
            int dest = edge[1];
            int cost = edge[2];
            graph[source].push_back({dest,cost});
        }
        return graph;
    }

    int minTime(vector<vector<vector<int>>> &graph,int n,int src) {

        vector<int> cost(n+1,0);
        cost[src] = 0;
        vector<bool>vis(n+1, false);

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,src}); // {cost, node}

        while(!pq.empty()) {
            pair<int,int>p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();
            // if already visited then skip
            if(vis[currNode])   continue;

            vis[currNode] = true;
            cost[currNode] = currCost;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i][0];
                int weight = graph[currNode][i][1];
                // if already visited then skip
                if(vis[neighbourNode])  continue;
                // else push into pq
                pq.push({currCost + weight, neighbourNode});
            }
        }

        for(int i=1; i<=n; i++)
            if(vis[i]==0)   return -1;

        int ans = 0;
        for(int x:cost)    ans = max(ans,x);
        return ans;
    }
};
```

13) Bellman Ford Algorithm → useful when weights < 0 (Dijkstra fails)
↳ dp algo
→ useful when finding negative weight cycle.

Eg $n = 4$ edges = [[0,1,-1], [0,2,4], [1,2,3], [1,3,2], [3,1,1], [3,2,5]]
[src, dest, wt]



initially dist

inf	inf	inf	inf
0	1	2	3

⇒ dist [src] = 0 &

⇒ relax every edge n-1 time ie run for loop & perform the following operation

$$dist[dest] = \min(dist[src] + weight,\ dist[dest])$$

⇒ finally relax one more time &
if dist [dest] > dist [src] + wt ⇒ -ve weight cycle present

⇒ we should relax 3 times & src = 0 ⇒ dist [src] = 0 dist

0	inf	inf	inf
0	1	2	3

→ for edge [0,1,-1], dist[1] = min(0+(-1), inf) = -1
[0,2,4], dist[2] = min(0+4, inf) = 4
[1,2,3], dist[2] = min(-1+3, 4) = 2
[1,3,2], dist[3] = min(-1+2, inf) = 1
[3,1,1], dist[1] = min(1+1, -1) = -1
[3,2,5], dist[2] = min(1+5, 2) = 2.

∴ dist =

0	-1	2	1
0	1	2	3

→ now use the above dist & perform same operation twice, in this case dist remains same.

→ during final relaxation, -ve weight cycle condition is not met.

Answer ⇒ dist =

0	-1	2	1
0	1	2	3

TC → O(V*E)
SC → O(V)

(14) Negative weight cycle → Bellman Ford Algorithm.

→ To check the presence of negative weight cycle using Bellman Ford Algorithm.

TC → $O(V*E)$
SC → $O(V)$

code →

```cpp
class Solution {
public:
    int isNegativeWeightCycle(int n, vector<vector<int>>edges){
        vector<int>dis(n,INT_MAX);
        // initially, dist to src is 0
        dis[0] = 0;
        // relax n-1 times
        for(int i=0;i<n-1;i++)
        {
            for(auto edge:edges)
            {
                int src = edge[0];
                int dest = edge[1];
                int wt = edge[2];
                if(dis[src]!=INT_MAX) // to avoid integer overflow
                    dis[dest] = min(dis[dest],dis[src]+wt);
            }
        }
        // final relaxation
        for(auto edge:edges)
        {
            int src = edge[0];
            int dest = edge[1];
            int wt = edge [2];
            if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)
              return 1;
        }
        return 0;
    }
};
```

(15) Floyd Warshall Algorithm

→ All source shortest path & -ve edges allowed.

→ Since its all source shortest path we need to run the loop for all nodes, considering it as intermediatery vertex.

→ $cost[i][j] = \min(cost[i][j], cost[i][k] + cost[k][j])$

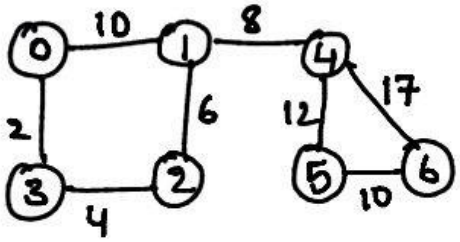
$TC \rightarrow O(N^3)$ $SC \rightarrow O(N^2)$

Code →

```
class Solution {
  public:
    void shortest_distance(vector<vector<int>>&matrix){
        int V = matrix.size();
        vector<vector<int>>costs(matrix.size(),vector<int>(matrix.size()));

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                costs[i][j] = matrix[i][j];

        for(int k=0;k<V;k++)
            for(int i=0;i<V;i++)
                for(int j=0;j<V;j++){
                    // if intermediate is not -1 then
                    if(costs[i][k]!=-1 && costs[k][j]!=-1){
                        if(costs[i][j]==-1)
                            costs[i][j] = costs[i][k]+costs[k][j];
                         else
                            costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);
                    }
                }

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                matrix[i][j] = costs[i][j];

    }
};
```

(16) Prim's Algorithm → Minimum spanning Tree (MST)

Eg



vis = { }

pq (node, weight)

↑ returns node with lowest cost/weight.

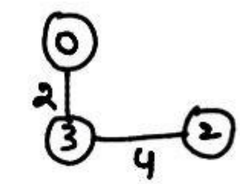
* To find MST, just push node along with its weight.

→ pq: (0,0) — vis = { }

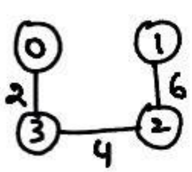
→ pq: (1,10) (3,2) — vis = {0}

→ pq: (1,10) (2,4) — vis = {0,3}

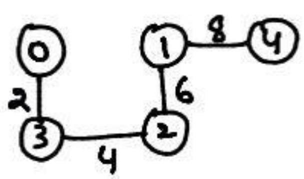
→ pq: (1,10) (1,6) — vis = {0, 3, 2}



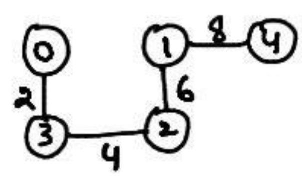
pq: (1,10) (4,8) — vis = {0, 3, 2, 1}



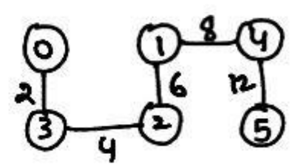
→ pq: (1,10) (5,12) (6,17) — vis = {0,3,2,1,4}



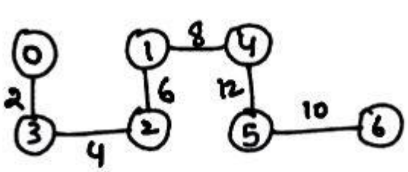
→ pq: (5,12) (6,17) — vis = {0,3,2,1,4}



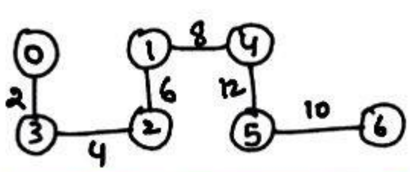
→ pq: (6,17) (6,10) — vis = {0,3,2,1,4,5}



pq: (6,17) — vis = {0,3,2,1,4,5,6}



→ pq: (empty) — vis = {0,3,2,1,4,5,6}



TC → $O(V + E\log V)$

SC → $O(V)$

Code →

```cpp
class Solution
{
    public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        int minCost = 0;
        vector<int>costs(V,INT_MAX);
        costs[0] = 0;
        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,0}); // {cost, Node}

        while(!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if(vis[currNode])    continue;

            minCost += currCost;

            vis[currNode] = true;
            costs[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int neighbourNodeCost = adj[currNode][i][1];
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }
        }
        return minCost;
    }
};
```

(17) Min cost to connect all points

→ Create graph with each node containing Wt & Node value

Wt = abs($X_i - X$) + abs ($Y_i - Y$)

→ Perform Prims algo.

Tc → O(V+ ElogV)

SC → O(V)

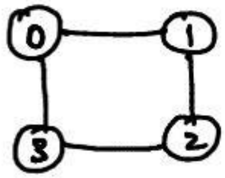
code →

```
class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {

        int n = points.size();
        vector<vector<pair<int, int>>> graph(n);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) continue;
                graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});
            }
        }

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        vector<bool> vis(n, false);
        pq.push({0, 0}); // {cost, Node}

        int ans = 0;
        while (!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if (vis[currNode]) continue;
            ans += currCost;
            vis[currNode] = true;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i].second;
                int neighbourNodeCost = graph[currNode][i].first;
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }

        }
        return ans;
    }
};
```

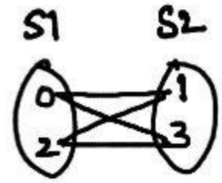
(18) Is graph Bipartite

Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets, S1 & S2 and no two vertices present in same set share an edge.

Eg n = 4



then



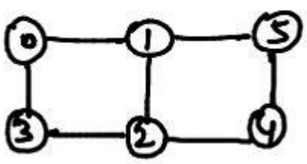
∴ the graph is bipartite.

⇒ for graph to be bipartite,

- it needs to be undirected acyclic graph (or)
- it needs to be even length cyclic graph

→ we generally denote sets by coloring it, color = 0, 1.
(0 ↓ S1, 1 ↓ S2)

Eg n = 6



vis = { } S1 = { } S2 = { }

initially color

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

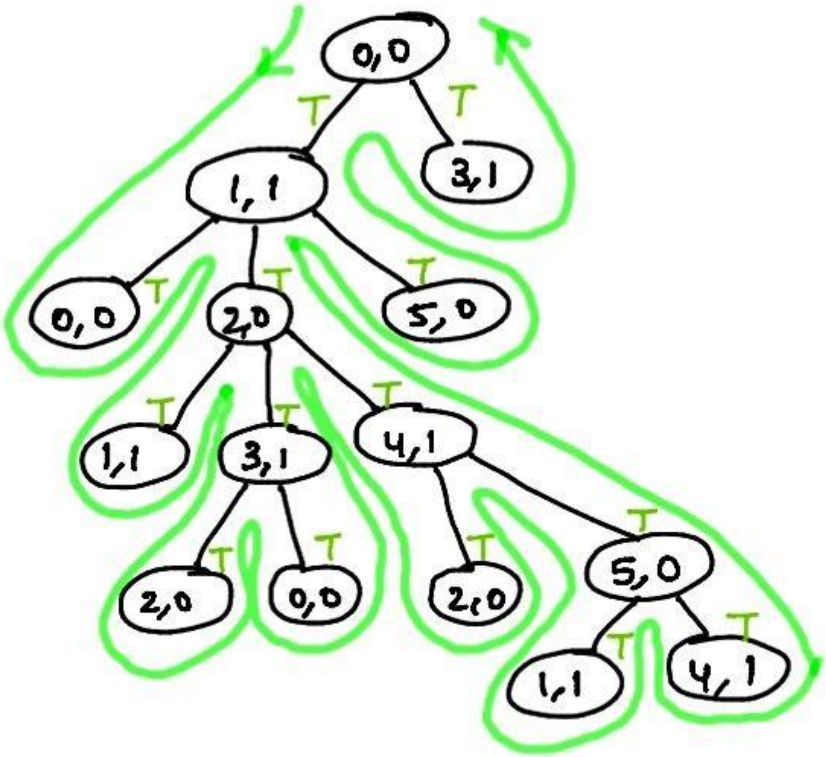
→ at each vertex, check if its visited or not.

→ if visited then check if its present in the intended set or not.

→ if yes then return true, else false

→ return AND of all the boolean values.

⇒



Code →

```cpp
class Solution {
public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n= graph.size();
        vector<int>colors(n,-1);

        for(int curr=0; curr<n ; curr++){
            // if already colored then skip
            if(colors[curr]!=-1)    continue;
            // check for even length cycle
            if(hasEvenLengthCycle(graph, curr, 0, colors)==false)  return false;
        }
        return true;
    }

    bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)
    {
        if(colors[curr]!=-1)
            return colors[curr]==color;

        // if not colored then color it
        colors[curr] = color;

        // check for neighbours
        for(int neigh: graph[curr])
        {
            if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)
                // 1- color will handle both changing colors 0 to 1 and 1 to 0
                return false;
        }
        return true;
    }

};
```

(19) Possible Bipartition →

→ create a graph using dislikes array.

→ use previous problem's approach to solve it.

code → $TC \rightarrow O(V+E)$ $SC \rightarrow O(V+E)$

```cpp
class Solution {
public:

    bool dfs(vector<int> graph[], int curr, vector<int>& color){

        // if not colored then color
        if(color[curr] == -1)
            color[curr] = 1;

        // process the neighbours and check their colors
        for(auto neigh : graph[curr])
        {
            if(color[neigh] == -1)
            {
                color[neigh] = 1 - color[curr];
                if(dfs(graph, neigh, color)==false) return false;
            }
            else if(color[neigh] == color[curr]) return false;
        }
        return true;
    }

    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {
        vector<int> color(n+1, -1);
        vector<int> graph[n+1];

        // populating the graph
        for(auto edge : dislikes){
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
        }

        for(int i=1; i<=n; i++){
            if(color[i] == -1)
                if(!dfs(graph, i, color)) return false;
        }

        return true;
    }
};
```

(20) **Disjoint set** → **UNION** & **FIND./getParent**

↳ helps in finding parent of component

↳ helps in UNION of components/vertices.

Eg (0) (1) ⟹ UNION (0,1) → (0)—(1)

Eg $n = 7$ initially every component is parent of itself

parent =

0	1	2	3	4	5	6
0	1	2	3	4	5	6

now getParent(2) = 2, getParent(3) = 3.

& if UNION(0,1) ⟹ (0)—(1) & parent[1] = 0

now getParent(1) = 0

& UNION(1, 2) ⟹ (0)—(1)—(2)

UNION(2,3) ⟹ (0)—(1)—(2)—(3)

& getParent(3) = 0

This increases the recursive calls and the tree is unbalanced so we'll use rank array to store min. height tree for node.

n = 7 initially every component is parent of itself

parent =

0	1	2	3	4	5	6
0	1	2	3	4	5	6

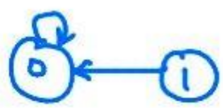
rank =

0	1	2	3	4	5	6
0	0	0	0	0	0	0

⇒ UNION (0, 1) ⇒ then find(0) & find(1) & 0 != 1 ∴ diff components.

as they are diff components find rank & rank[0] = rank[1] = 0

∴ select either 0 or 1 & make it as root & inc the rank by 1

⇒ 

parent =

0	1	2	3	4	5	6
0	0	2	3	4	5	6

rank =

0	1	2	3	4	5	6
1	0	0	0	0	0	0

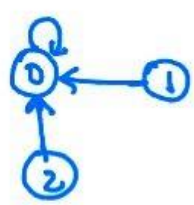
⇒ UNION (1, 2) ⇒ parent(1) = 0 & parent(2) = 2

now rank[0] = 1 & rank[2] = 0

as rank[0] > rank[2] ,

vertex 0 should be the parent

& donot update rank if they are unequal.

⇒ 

parent =

0	1	2	3	4	5	6
0	0	0	3	4	5	6

rank =

0	1	2	3	4	5	6
1	0	0	0	0	0	0

Code →

```cpp
class DisjSet {
    int *rank, *parent, n;

    public:
    DisjSet(int n)
    {
        rank = new int[n];
        parent = new int[n];
        this->n = n;
        makeSet();
    }

    void makeSet()
    {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x)
    {
        // if x is not parent of itself then
        // find parent recursively
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void Union(int x, int y)
    {
        int xset = find(x);
        int yset = find(y);

        // if set of x and y are same then return
        if (xset == yset)   return;

        // place the elements in small rank
        if (rank[xset] < rank[yset]) {
            parent[xset] = yset;
        }
        else if (rank[xset] > rank[yset]) {
            parent[yset] = xset;
        }
        // if same rank then increment it
        else {
            parent[yset] = xset;
            rank[xset] = rank[xset] + 1;
        }
    }
};
```

(21) Kruskal's Algorithm →

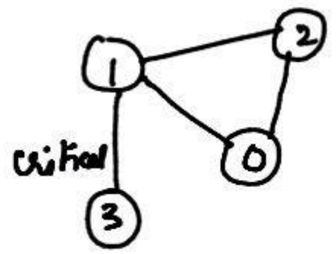
→ This is used to find minimum spanning tree.
→ can be implemented using Disjoint set.
→ sort all the edges in ↑ order of weight.
→ pick smallest edge & check if it contributes to cycle in graph
→ if yes then discard else include.

code →

```cpp
class Graph {
    vector<vector<int> > edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU - DisjointSet
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y)) {
                s.union(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};
```

(22) Critical Connection in a Network →

Eg n=4 edges = [[0,1],[1,2],[2,0],[1,3]]



→ Critical connection is a connection, when removed from graph, would result in breaking graph into different components.

Here if [1,3] is removed then graph becomes disconnected.

Approach 1

→ Remove one edge each time
→ Perform dfs
→ If all vertices are not visited then
→ Removed edge is a critical connection.

Approach 2

→ initialise distime array & mintime array with -1.

distime → discovery time for vertex

mintime → min time for vertex to be discovered.

→ perform dfs from one node
→ if neighbour == parent then continue
→ else if neighbour is already visited then

mintime [curr] = min (mintime [curr], distime [neigh])

→ while returning mintime [curr] = min (mintime [curr], mintime [neigh])
& at any point if distime [curr] < mintime [neigh]

This indicates critical connection

code →

```cpp
class Solution {
public:

    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        vector<int> graph[n];
        for(vector<int> edge: connections){
            int u = edge[0];
            int v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        return findCriticalConnections(n, graph);
    }

    vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
        vector<int> disTime(n,-1);
        vector<int> lowTime(n,-1);
        int time = 0;
        vector<vector<int>> answer;
        tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
        return answer;
    }

    void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
    vector<int> &lowTime, int &time, vector<vector<int>> &answer){

        disTime[curr] = time;
        lowTime[curr] = time;
        time += 1;

        for(int neigh: graph[curr]){
            if(neigh == parent) continue;

            if(disTime[neigh]!=-1){
                lowTime[curr] = min(lowTime[curr], disTime[neigh]);
                continue;
            }

            tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
            lowTime[curr] = min(lowTime[curr], lowTime[neigh]);

            if(disTime[curr] < lowTime[neigh]){
                vector<int> temp;
                temp.push_back(curr);
                temp.push_back(neigh);
                answer.push_back(temp);
            }
        }
        return;
    }

};
```

Dynamic Programming - 1

- Karun Karthik

Contents

Introduction

Dynamic programming is a technique to solve problems by breaking it down into a collection of sub-problems, solving each of those sub-problems just once and storing these solutions inside the cache memory in case the same problem occurs the next time.

Dynamic Programming is mainly an optimization over plain recursion .
Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
This simple optimization reduces the time complexities from exponential to polynomial.

There are two different ways to store our values so that they can be reused at a later instance. They are as follows:

1. Memoization or the Top Down Approach.
2. Tabulation or the Bottom Up approach.

In Memoization we start from the extreme state and compute result by using values that can reach the destination state i.e the base state.

In Tabulation we start from the base state and then compute results all the way till the extreme state.

Note: To store the intermediate results we can use Array, Matrix, Hashmap etc., all we need is data storage and retrieval with a specific key.

How to find the use case of Dynamic Programming?

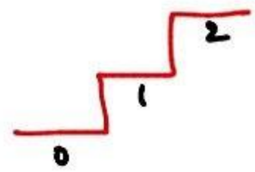
You can use DP if the problem can be,

1. Divided into sub-problems
2. Solved using a recursive solution
3. Containing repetitive sub-problems

① **Climbing Stairs** → given a value 'N', find the number of ways to reach N & jumps possible are ONE or TWO.

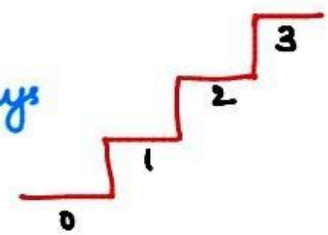
Eg $n = 2 \Rightarrow$ $0 \xrightarrow{1} 1 \xrightarrow{1} 2$
$0 \xrightarrow{2} 2$

} for N = 2 we have 2 ways

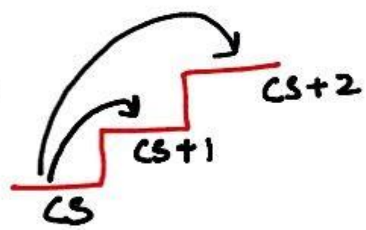


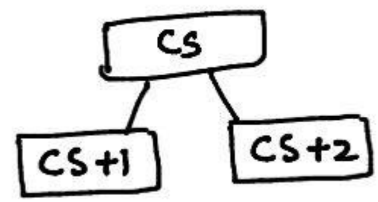
$n = 3 \Rightarrow$ $0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3$
$0 \xrightarrow{1} 1 \xrightarrow{2} 3$
$0 \xrightarrow{2} 2 \xrightarrow{1} 3$

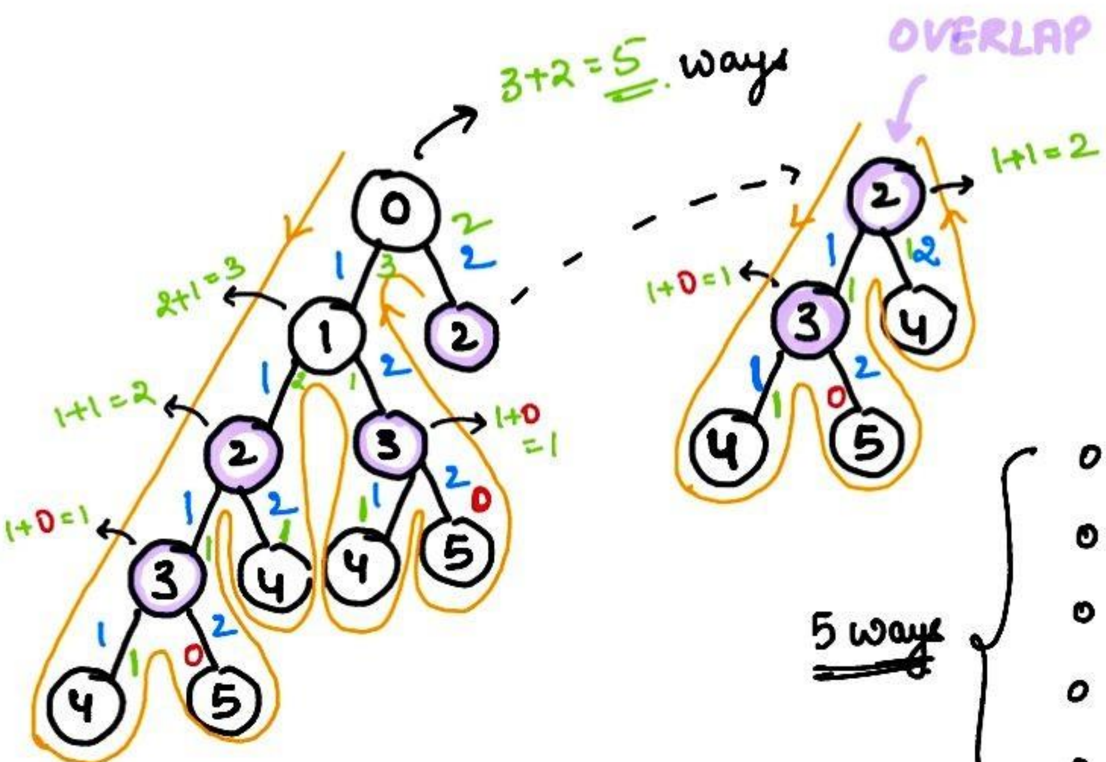
} for N = 3 we have 3 ways



n = 4
↳ for every stair we have 2 cases ie ⇒







if cs == n
return 1

if cs > n
return 0

5 ways {
$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$
$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{2} 4$
$0 \xrightarrow{1} 1 \xrightarrow{2} 3 \xrightarrow{1} 4$
$0 \xrightarrow{2} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$
$0 \xrightarrow{2} 2 \xrightarrow{2} 4$

★ Here we can see for ②, ③ the subproblem is being done multiple times, we can solve using dp.

code →

```
class Solution {
public:
    int totalWays(int currentStair, int targetStair, unordered_map<int,int> &memo){

        if(currentStair==targetStair){
            return 1;
        }

        if(currentStair > targetStair){
            return 0;
        }

        int currentKey = currentStair;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        int oneStep = totalWays(currentStair+1, targetStair, memo);
        int twoStep = totalWays(currentStair+2, targetStair, memo);

        memo[currentKey] = oneStep+twoStep;

        return oneStep+twoStep;

    }

    int climbStairs(int n) {
        unordered_map<int,int> memo;
        return totalWays(0,n,memo);
    }
};
```

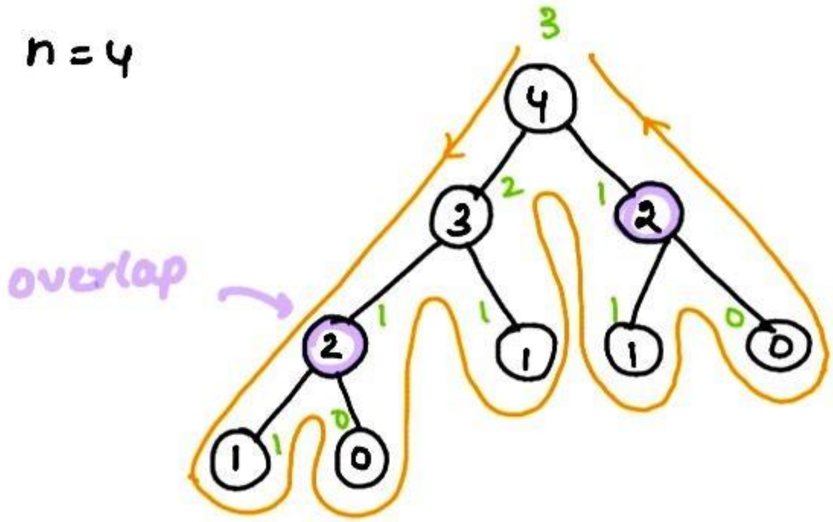
② Fibonacci Number →

$f(n) = f(n-1) + f(n-2)$ & $f(0) = 0$

$f(1) = 1$

$n \geq 0.$

Eg → $n = 4$

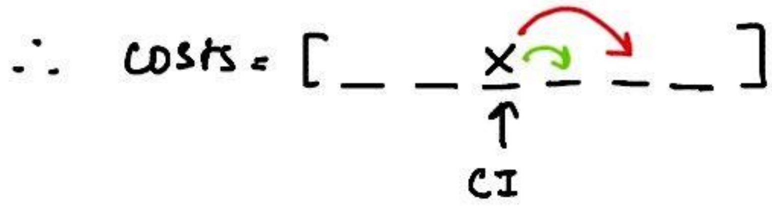


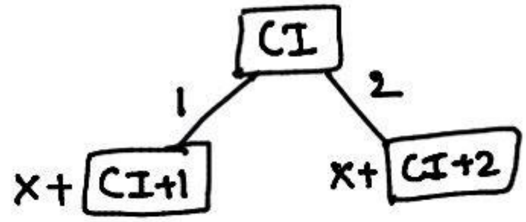
code →

```cpp
class Solution {
public:
    int helper(int n, unordered_map<int,int>&memo){

        if(n<=1){
            return n;
        }

        int currentKey = n;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }


        int a = helper(n-1,memo);
        int b = helper(n-2,memo);

        memo[currentKey] = a+b;
        return memo[currentKey];
    }


    int fib(int n) {
        unordered_map<int,int>memo;
        return helper(n,memo);
    }
};
```

3 Min Cost Climbing Stairs →

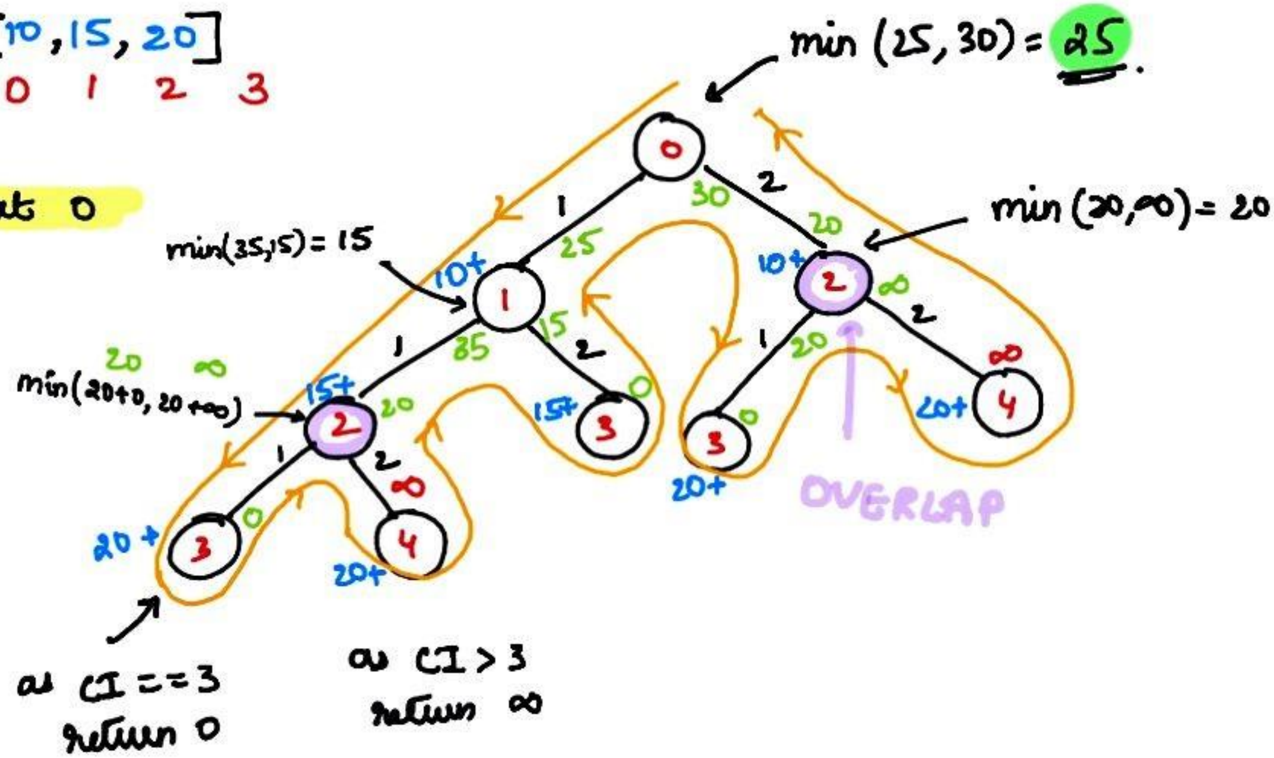
Given costs array, find min cost to reach the end, starting from 0 or 1 & making 1 or 2 jumps.



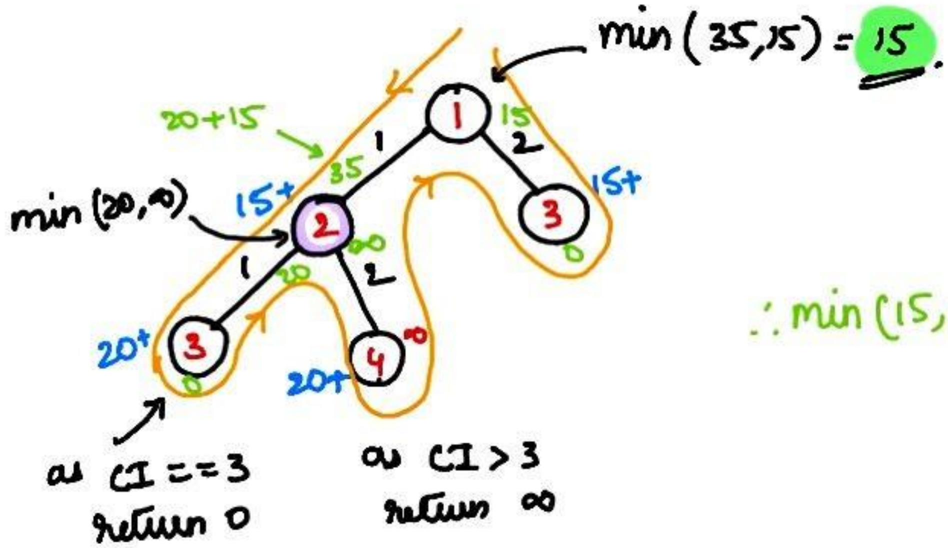


Eg cost = [10, 15, 20]
0 1 2 3

starting at 0



starting at 1



∴ min (15, 25) = 15

Code →

```cpp
class Solution {
public:

    int minCost(vector<int>&cost, int currentIndex, unordered_map<int,int> &m){

        if(currentIndex == cost.size()){
            return 0;
        }

        if(currentIndex > cost.size()){
            return 1000;    // large values, serves as INFINITY
        }

        if(m.find(currentIndex)!=m.end()){
            return m[currentIndex];
        }

        int oneJump = cost[currentIndex] + minCost(cost,currentIndex+1, m);
        int twoJump = cost[currentIndex] + minCost(cost,currentIndex+2, m);

        m[currentIndex] = min(oneJump, twoJump);
        return m[currentIndex];
    }

    int minCostClimbingStairs(vector<int>& cost) {
        unordered_map<int,int> m;
        return min( minCost(cost,0,m), minCost(cost,1,m));
    }
};
```

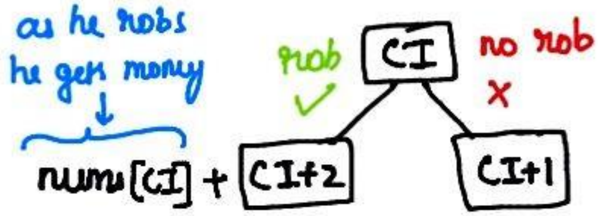
④ House Robber → given an array of no. representing money, find max amount, that can be robbed without choosing the adjacent houses.

Eg nums = [2, 7, 9, 3, 1]
0 1 2 3 4

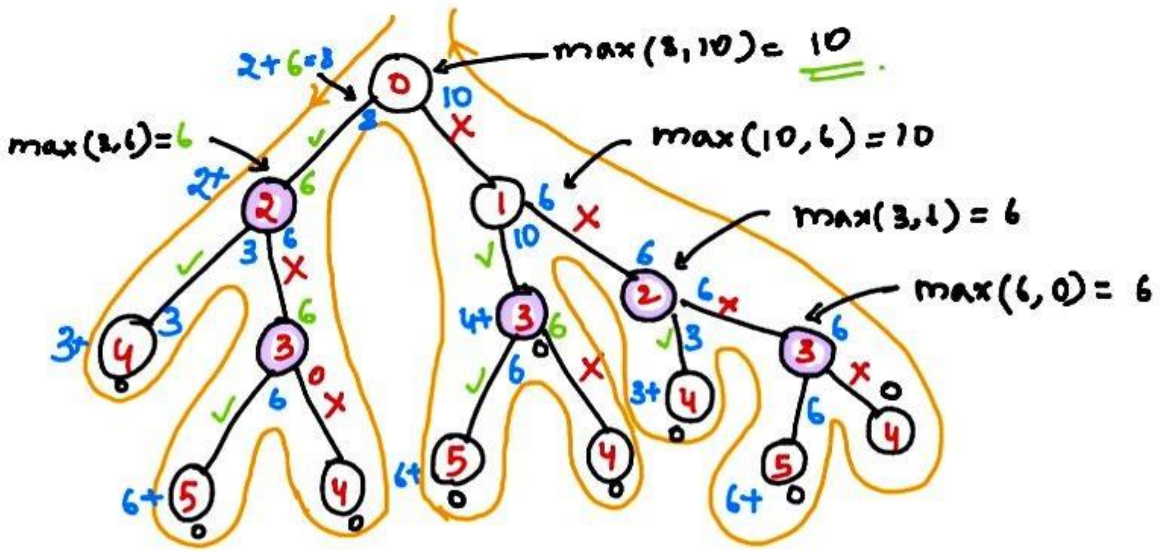
⇒ max amount = 2 + 9 + 1 = 12
0 → 2 → 4

→ if robber robs house, then amount = nums[CI] & CI = CI + 2 (to avoid adjacent)
else CI = CI + 1

ie nums = [_ _ _ CI _ _ _ _]



Eg [2, 4, 3, 6]
0 1 2 3



as CI > 3
return 0

∴ at every node find max(left, right) & add it's value to the nums[CI] if selected, else continue

Code →

```cpp
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, unordered_map<int,int>&m){

        if(currentIndex >= nums.size()){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, m);
        int noRob = helper(nums, currentIndex+1, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];
    }

    int rob(vector<int>& nums) {
        unordered_map<int,int> m;
        return helper(nums,0,m);
    }
};
```

⑤ House Robber - II →

In this problem, the approach will be similar to previous one, but the houses are in circle, which means that

* if we start from 1st house, then we can't rob the last house.
* if we start from 2nd house, then we can rob the last house.
* and return max value between 1st house & 2nd house
* if only 1 house is present, then rob it directly.

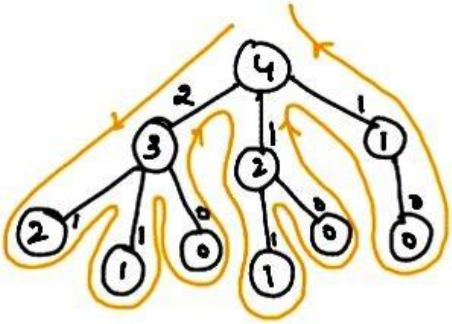
code →

```
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, int lastIndex, unordered_map<int,int>&m){

        if(currentIndex > lastIndex){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, lastIndex, m);
        int noRob = helper(nums, currentIndex+1, lastIndex, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];

    }

    int rob(vector<int>& nums) {

        int n = nums.size();
        if(n==1)     return nums[0];

        unordered_map<int,int> memo1,memo2;
        // we can start robbing from 2 houses
        int firstHouse = helper(nums, 0, n-2, memo1);
        int secondHouse = helper(nums, 1, n-1, memo2);
        return max(firstHouse, secondHouse);
    }
};
```

⑥ N-th Tribonacci → given n, find T_n

$$T_{n+3} = T_n + T_{n+1} + T_{n+2} \text{ \& } n \geq 0 \quad T_0 = 0,\ T_1 = 1,\ T_2 = 1.$$

Eg $n = 4$



4 (2+1+1)

code →

```cpp
class Solution {
public:

    int helper(int n, unordered_map<int,int> &m){
        if(n<=1){
            return n;
        }

        if(n==2){
            return 1;
        }

        int currentNum = n;

        if(m.find(currentNum)!=m.end()){
            return m[currentNum];
        }

        int a = helper(n-1,m);
        int b = helper(n-2,m);
        int c = helper(n-3,m);

        m[currentNum] = a+b+c;

        return m[currentNum];
    }

    int tribonacci(int n) {
        unordered_map<int,int>m;
        return helper(n,m);
    }
};
```

7. 0 - 1 Knapsack Problem

→ find max profit such that the weight of all items <= capacity.

$$\begin{array}{l} \quad\;\; 0\;\;1\;\;2\;\;3 \\ wt = [3,4,6,5] \end{array}$$

profits = [2,3,1,4] → if we select 0 & 3,

then total weight = wt[0] + wt[3]

= 3+5 = 8.

capacity = 8

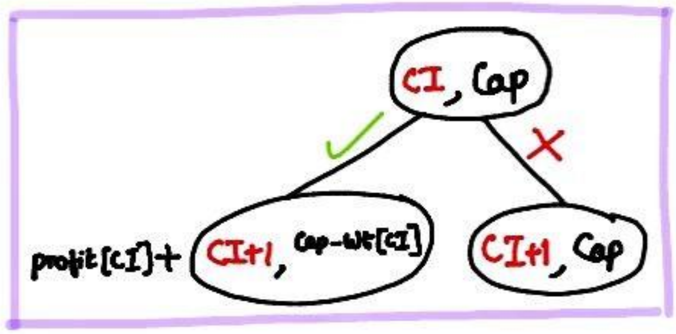
& profits are 2 + 4 = 6 That's max profit possible

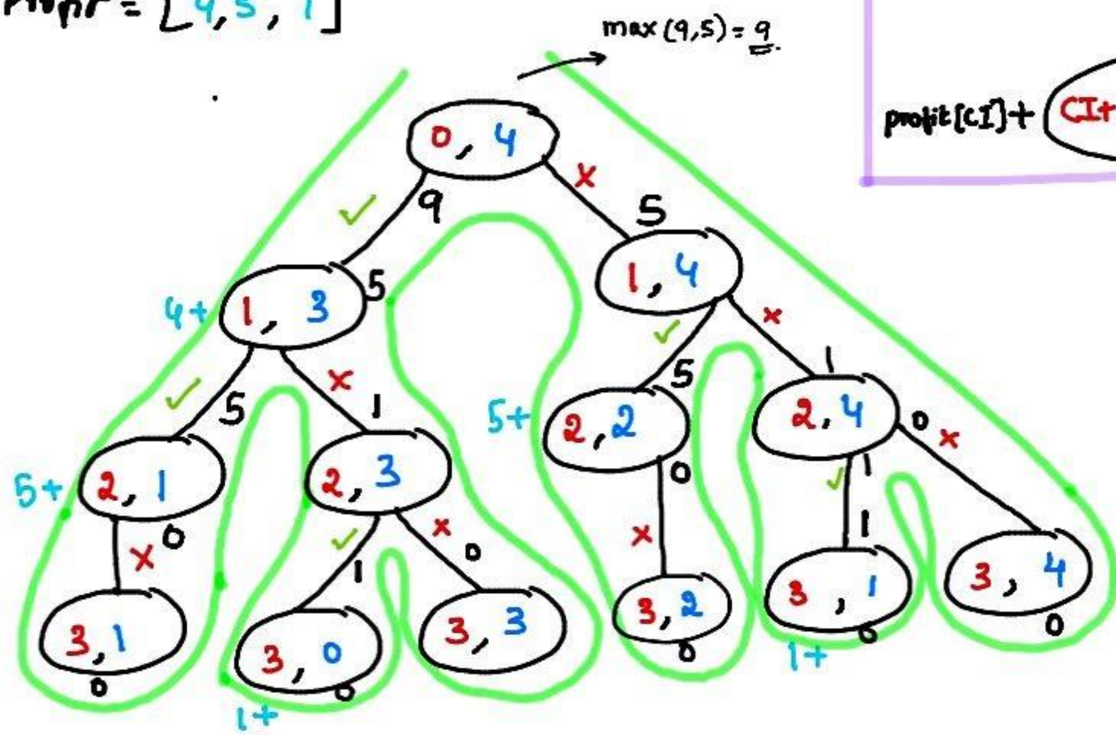
Eg

$$\begin{array}{l} \quad\;\; 0\;\;1\;\;2 \\ wt = [1,2,3] \end{array}$$

capacity = 4

Profit = [4,5,1]





∴ at every step,

→ if selecting an index then reduce capacity by wt[CI] & add profit[CI] to result

→ if not selecting, increment CI by 1

→ find max(left, right)

code →

```cpp
class Solution
{
    public:

    int helper(int W, int wt[], int val[], int n, int curr,
                                unordered_map<string,int> &memo){
        if(curr==n) return 0;

        // Instead of Matrix we can use strings as unique keys
        string currKey = to_string(curr)+"_"+to_string(W);

        if(memo.find(currKey)!=memo.end()) return memo[currKey];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, n, curr+1, memo);
        }

        int notSelected = helper(W, wt, val, n, curr+1, memo);

        memo[currKey] = max(selected, notSelected);
        return memo[currKey];
    }


    int knapSack(int W, int wt[], int val[], int n)
    {
       unordered_map<string,int> memo;
       return helper(W, wt, val, n, 0, memo);
    }
};
```

(8) Partition Equal Subset Sum →

Given an array, find if it can be divided into two subsets whose sum is equal.

Eg nums = [1, 5, 11, 5] can be divided into S1 = {1, 5, 5} & S2 = {11}
& sum of S1 == sum of S2 ∴ returns True.

∴ initially find sum of elements in array.

1) if sum is odd then return False

2) if sum is even, then proceed.

→ find a subset whose value == sum/2
which means that the other subset will have value == sum/2.

→ lets say ts = sum/2 (ts is target sum)

→ At every index, we have 2 choices

current index

1) if we select then $ts = ts - nums[CI]$
&
$CI = CI + 1$

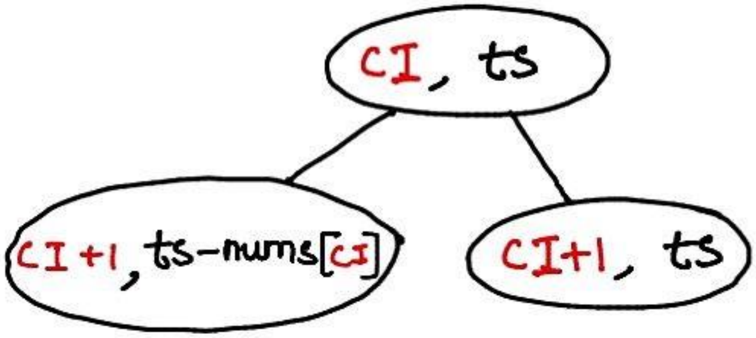
2) if we donot select then $ts = ts$ (i.e remains same)
$CI = CI + 1$

3) return OR of left & right branch.

⇒ $nums = [\overset{0}{1}, \overset{1}{5}, \overset{2}{11}, \overset{3}{5}]$

sum = 22

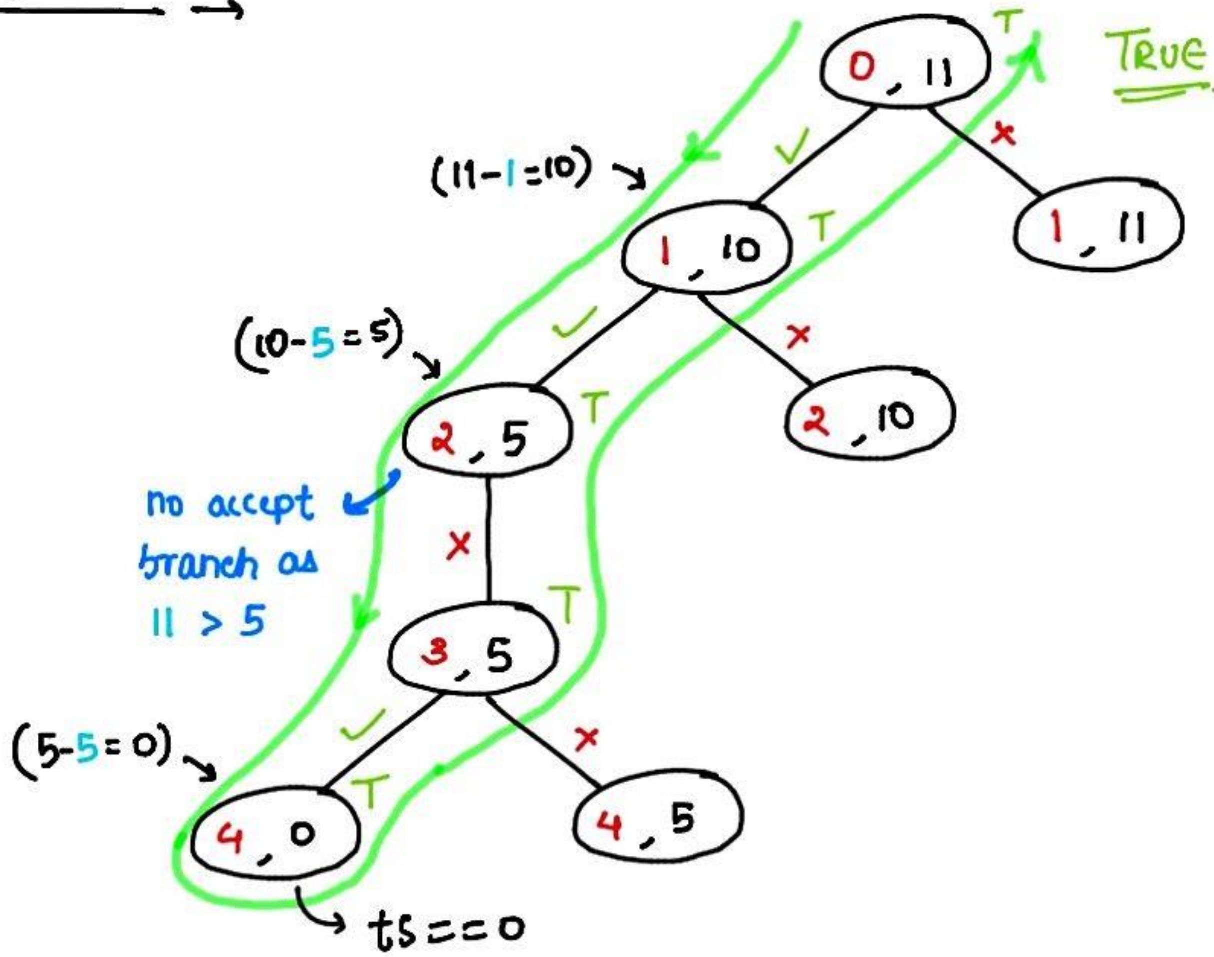
ts = 11



Here sum % 2 == 0

∴ dividing into 2 subsets is possible.

Explanation →



⇒ that subset is found & return True

as we are using OR, one True branch is sufficient

Code →

```cpp
class Solution {
public:

    bool isPossible(int targetSum,int currentIndex, vector<int>&nums,
                                    unordered_map<string, bool> &memo){

        if(targetSum == 0)
            return true;

        if(currentIndex >= nums.size())
            return false;

        string currentKey = to_string(currentIndex)+"_"+to_string(targetSum);

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        bool possible = false;

        if(nums[currentIndex]<=targetSum)
            possible = isPossible(targetSum-nums[currentIndex], currentIndex+1, nums, memo);

        // if already Possible then return True directly
        if(possible){
            memo[currentKey] = possible;
            return true;
        }

        bool notPossible = isPossible(targetSum, currentIndex+1, nums, memo);

        memo[currentKey] = possible||notPossible;
        return memo[currentKey];
    }

    bool canPartition(vector<int>& nums) {

        int total = 0;
        for(auto it:nums) total+= it;

        if(total%2!=0)  return false;

        unordered_map<string, bool> memo;
        return isPossible(total/2,0, nums,memo);
    }
};
```

(9) Target Sum →

Given an array & target, find the number of ways to reach target by using + or - before each element in array.

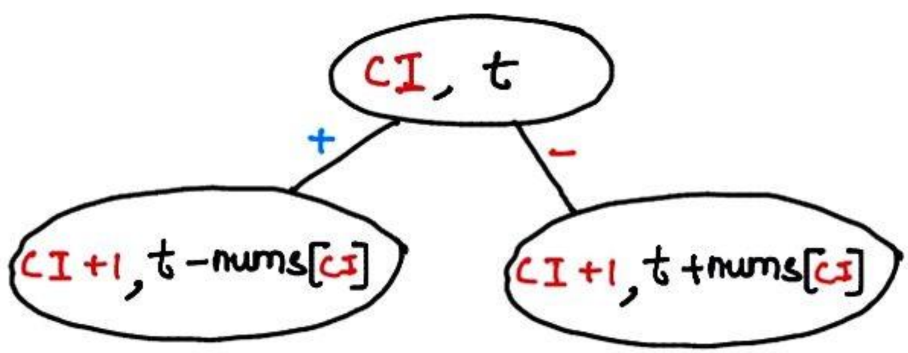
Eg

```
Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: There are 5 ways to assign symbols to make the sum
of nums be target 3.
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

→ at every index we can use + or − sign

if + then $t = t - (+\,nums[CI]) \Rightarrow t - nums[CI]$

if − then $t = t - (-\,nums[CI]) \Rightarrow t + nums[CI]$



→ at every node, return the sum of values from left & right. Because we need to find the total number of ways.

code →

```cpp
class Solution {
public:
    int totalWays(int currentIndex, vector<int>&nums, int target, unordered_map<string,int> &memo){

        if(target==0 and currentIndex==nums.size()){
            return 1;
        }

        if(currentIndex>=nums.size() and target!=0){
            return 0;
        }

        string key = to_string(currentIndex)+"_"+to_string(target);

        if(memo.find(key)!=memo.end()){
            return memo[key];
        }

        int plus = totalWays(currentIndex+1, nums, target-nums[currentIndex],memo);

        int minus = totalWays(currentIndex+1, nums, target+nums[currentIndex],memo);

        memo[key] = plus+minus;

        return plus+minus;
    }

    int findTargetSumWays(vector<int>& nums, int target) {
        unordered_map<string,int> memo;
        return totalWays(0,nums,target,memo);
    }
};
```

(10) Count number of subsets with given difference →

→ This is similar to Target Sum.

given the difference between two subsets, and an array find no. of subsets with the difference.

Approach →

Lets say $s1 - s2 =$ difference (given) — (1)

we can calculate sum of every element, say sum & it can be said that for 2 subsets $s1$ & $s2$

$$s1 + s2 = \text{sum} \quad \text{— (2)}$$

Now (1) + (2) $\Rightarrow$ $2(s1) =$ difference + sum

$$s1 = (\text{difference} + \text{sum})/2.$$

→ Implement Target Sum with target value = $s1$

(11) Delete and Earn →

You are given an integer array `nums`. You want to maximize the number of points you get by performing the following operation any number of times:

- Pick any `nums[i]` and delete it to earn `nums[i]` points. Afterwards, you must delete **every** element equal to `nums[i] - 1` and **every** element equal to `nums[i] + 1`.

Return *the* ***maximum number of points*** *you can earn by applying the above operation some number of times.*

Eg nums = [2, 2, 3, 3, 3, 4]

→ if we start deleting 2, then result = 2+2 = 4

then nums = [3, 3, 3, 4]

& we need to delete all 2+1 & 2−1 ⇒ nums = [4]

→ if we delete 4, then result = 4+4 = 8.

& nums = [ ]

(or)

→ if we start deleting 3, then result = 3+3+3 = 9.

then nums = [2, 2, 4]

& we need to delete all 3−1 & 3+1 ⇒ nums = [ ]

∴ result = 9.

(or)

→ if we start deleting 4, then result = 4

then nums = [2, 2, 3, 3, 3]

& we need to delete all 4+1 & 4−1 ⇒ nums = [2, 2]

→ if we delete 2, then result = 4+4 = 8.

& nums = [ ]

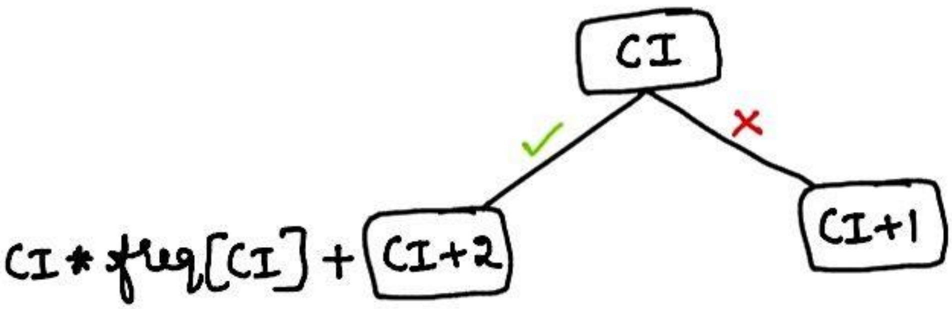
→ we can store frequency of each element and use the similar approach

nums = [2, 2, 3, 3, 3, 4]

freq =

0	1	2	3	4
0	0	2	3	1

↑
CI



code →

```cpp
class Solution {
public:

    int maxPoints(vector<int>& freq, int currentIndex, unordered_map<int,int>&memo){

        if(currentIndex >= freq.size()) return 0;

        int key = currentIndex;

        if(memo.find(key) != memo.end()) return memo[key];

        int Delete = currentIndex*freq[currentIndex] + maxPoints(freq, currentIndex+2, memo);
        int NotDelete = maxPoints(freq, currentIndex+1, memo);

        memo[key] = max(Delete, NotDelete);

        return memo[key];
    }

    int deleteAndEarn(vector<int>& nums) {

        int maxi = *max_element(nums.begin(), nums.end());
        vector<int> freq(maxi+1, 0);

        for(auto i: nums)   freq[i]++;

        unordered_map<int,int> memo;

        return maxPoints(freq, 0, memo);
    }
};
```

(12) Unbounded Knapsack → Similar to 0-1 Knapsack but allows us to choose an item more than once

Eg

wt = [2, 1] (indices 0, 1)

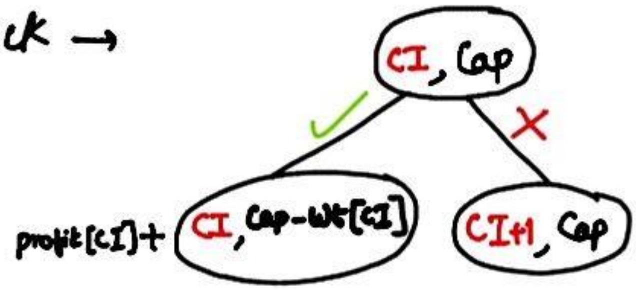
values = [1, 1]

capacity = 3

if bounded knapsack then profit = **2** (2, 1) → 1+1

if unbounded knapsack then profit = **3** (1, 1, 1) → 1+1+1

for unbounded Knapsack →



Code →

```cpp
class Solution{
public:
    int helper(int W, int wt[], int val[], int N, int curr, vector<vector<int>>&memo){

        if(W==0)    return 0;
        if(curr==N) return 0;

        if(memo[curr][W]!=-1)   return memo[curr][W];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, N, curr, memo);
        }

        int notSelected = helper(W, wt, val, N, curr+1, memo);

        memo[curr][W] = max(selected, notSelected);
        return memo[curr][W];
    }

    int knapSack(int N, int W, int val[], int wt[])
    {
        vector<vector<int>> memo( N , vector<int> (W+1, -1));
        return helper(W, wt, val, N, 0, memo);
    }
};
```

(13) Coin Change II → (Similar to unbounded knapsack)

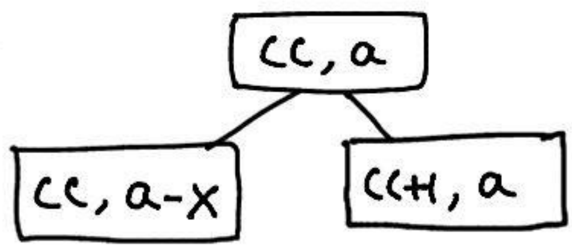
Given an array of coins & amount, find total number of ways / combinations to make up that amount.

Eg coins = [1, 2, 5]
amount = 5

4 ways
- 1+1+1+1+1
- 1+1+1+2
- 1+2+2
- 5

coins = [_ x _ _ _ _]   x = coins[cc]

current coin ← cc, a → amount



code →

```cpp
class Solution {
public:

    int totalWays(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0) return 1;    // amount==0 means that target is reached so return 1
        if(currentIndex >= coins.size()) return 0;  //if index is out of bounds then return 0

        if(memo[currentIndex][amount]!=-1) return memo[currentIndex][amount];

        int consider = 0;
        if(coins[currentIndex]<=amount){
            consider = totalWays(currentIndex,coins,amount-coins[currentIndex],memo);
        }
        int notConsider = totalWays(currentIndex+1,coins,amount,memo);

        memo[currentIndex][amount] = consider+notConsider;
        return memo[currentIndex][amount];
    }

    int change(int amount,vector<int>& coins) {
        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        return totalWays(0,coins,amount,memo);
    }
};
```

(14) Coin Change → (Similar to unbounded knapsack)

given an array of coins & amount, find fewest number of coins to make up that amount, return -1 if its not possible

Eg coins = [1,2,5] for 11 ⇒ $\overbrace{1+\ldots+1}^{11\text{ times}} = 11$

amount = 11

⋮

1+2+2+2+2+2 = 11

⋮

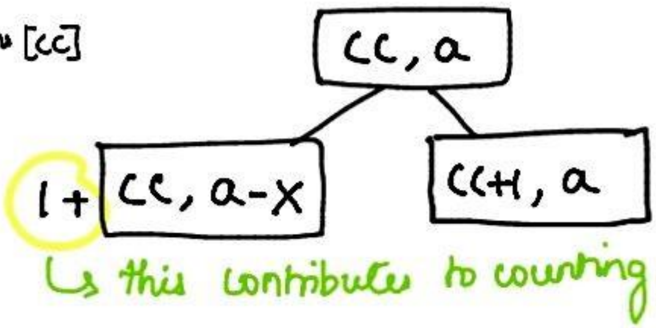
1+5+5 = 11

} out of all the ways last has min coins.

coins = [_ x _ _ _ _] x = coins[cc]

↑

current coin ← cc, a → amount



code →

```cpp
class Solution {
public:

    int minimumCoins(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0)    return 0;
        if(currentIndex >= coins.size())    return 100000;  //Any Max Value outside boundary

        if(memo[currentIndex][amount]!=-1)  return memo[currentIndex][amount];

        int consider = 100000;
        if(coins[currentIndex]<=amount){
            consider = 1 + minimumCoins(currentIndex, coins, amount-coins[currentIndex], memo);
        }

        int notConsider = minimumCoins(currentIndex+1, coins, amount, memo);

        memo[currentIndex][amount]= min(consider,notConsider);
        return memo[currentIndex][amount];
    }

    int coinChange(vector<int>& coins, int amount) {

        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        int ans = minimumCoins(0, coins, amount, memo);

        return (ans==100000)? -1 : ans;
    }
};
```

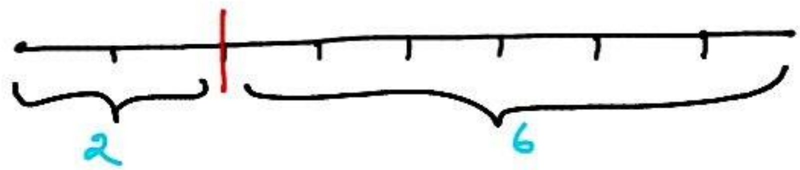
(15) Rod Cutting →

Given a rod of length N and array of prices find the max value that can be obtained by cutting rod.

Eg $N = 8$ prices = $\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline \end{array}$

$$\begin{array}{c} \text{length} \\ \text{prices} \\ \text{index} \end{array}\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1, & 5, & 8, & 9, & 10, & 17, & 17, & 20 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

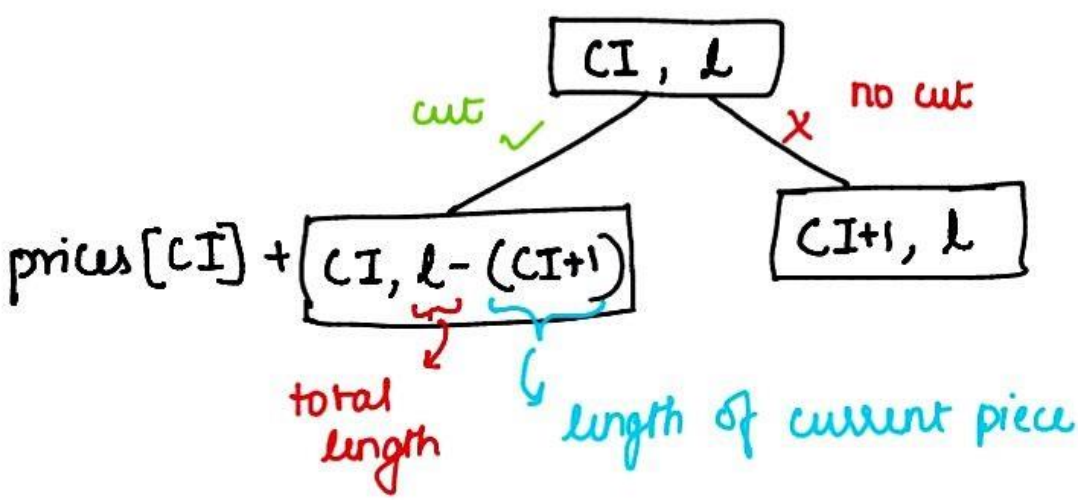
* price of a piece is prices[CI], whose length is CI+1



if we cut our rod into 2 pieces of length 2, 6 we get max value of 5 + 17 i.e. 22.

→ there might be other ways, but this particular configuration returns max value.

* At any instance length of current piece is CI+1



code →

```cpp

class Solution{
  public:
    int maxProfit(int price[],int currentIndex, int n, vector<vector<int>>&memo){
        if(n==0)   return 0;
        if(currentIndex>=n)   return 0;

        if(memo[currentIndex][n]!=-1)   return memo[currentIndex][n];

        int selected = 0;
        if(currentIndex+1<=n){
            selected = price[currentIndex]+maxProfit(price, currentIndex, n-(currentIndex+1), memo);
        }
        int notSelected = maxProfit(price, currentIndex+1, n, memo);

        memo[currentIndex][n] = max(selected, notSelected);
        return memo[currentIndex][n];
    }
    int cutRod(int price[], int n) {
        vector<vector<int>> memo(n+1, vector<int>(n+1,-1));
        return maxProfit(price,0,n,memo);
    }
};
```

Dynamic Programming - 2

- Karun Karthik

Contents

(16) Best time to Buy & Sell Stock →

Given an array of prices, find the max profit if we are allowed to do one transaction

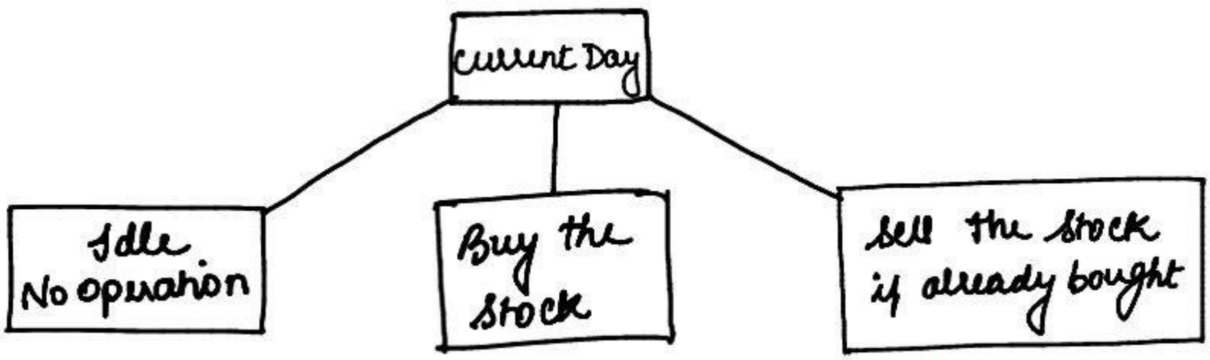
Eg

prices = [7, 1, 5, 3, 6, 4]
0 1 2 3 4 5

→ we get maxProfit when we buy at day 0 & sell on day 4

⇒ profit = 6 − 1 = 5.

lets look at choices we have,



→ to handle the case that transaction could occur once, we use a variable called transaction = 1.

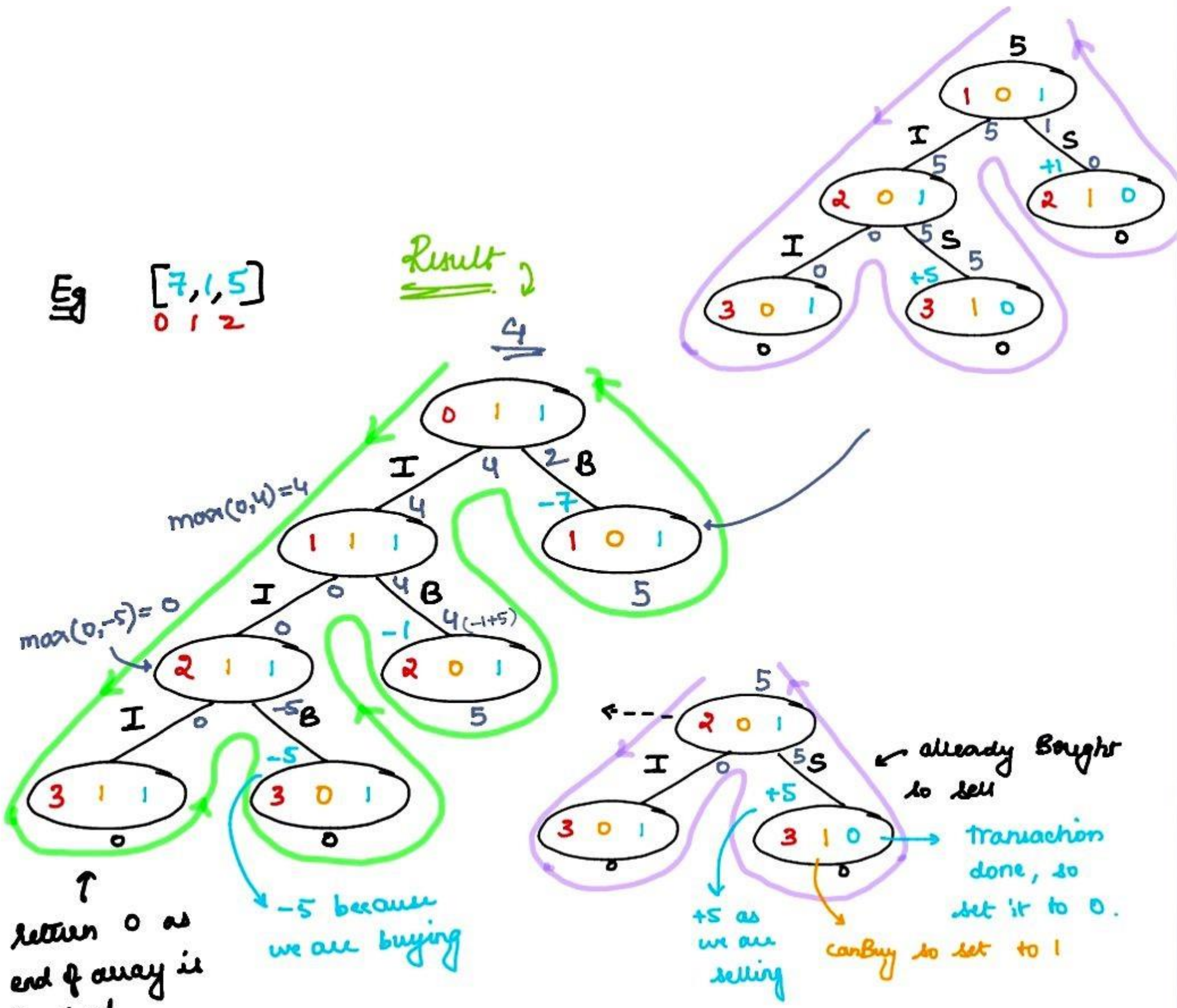
→ to handle these cases, we use a variable called canBuy.

→ once bought canBuy = false

→ once sold canBuy = true

∴ Our recursive structure would be as follows →

current Day, canBuy, transaction



Eg [7,1,5]
0 1 2
Result
4
max(0,4)=4
max(0,-5)= 0
4(-1+5)
return 0 as end of array is reached
-5 because we are buying
already bought so sell
transaction done, so set it to 0.
+5 as we are selling
canBuy so set to 1

code →

```
class Solution {
public:
    int find(vector<int> &prices, int currDay, int k, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size() || k<=0 ) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, k, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, k, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, k, canBuy, memo);
           int sell = prices[currDay] + find(prices, currDay+1, k-1, !canBuy, memo);
           return memo[currDay][canBuy] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true and transaction as k = 1
        return find(prices,0,1,true,memo);
    }
};
```

(17) Best time to Buy & Sell Stock - II →

→ In this we can have many transactions that can be done.

0 1 2 3 4 5

Eg → prices = [7, 1, 5, 3, 6, 4]

↳ Buy on 1 & sell on 2 profit = 5 - 1 = 4

Buy on 3 & sell on 4 profit = 6 - 3 = 3

Total Profit = 7 Ans

code →

Remove the parameter K i.e transaction limit.

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size()) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
            int idle = find(prices, currDay+1, canBuy, memo);
            int sell = prices[currDay] + find(prices, currDay+1, !canBuy, memo);
            return memo[currDay][canBuy] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true and transaction are infinite so ignore k
        return find(prices, 0, true, memo);
    }
};
```

(18) Best time to Buy & Sell Stock - III →

In this maximum profit has to be achieved by making atmost 2 transactions.

Eg prices = [3, 3, 5, 0, 0, 3, 1, 4]
(indices 0 1 2 3 4 5 6 7)

↳ Buy on 4 & sell on 5   profit = 3 − 0 = 3
Buy on 6 & sell on 7   profit = 4 − 1 = 3

Total Profit = 6 Ans

Code →

In the base condition is no. of transactions >= 2 then return 0.
(Line 6)

↳ ie possible transactions are when it is = 0, 1

```cpp
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, int transaction, bool canBuy,
4      vector<vector<vector<int>>> &memo){
5  
6          if(currDay >= prices.size() || transaction>=2) return 0;
7  
8          if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];
9  
10         if(canBuy)
11         {
12             int idle = find(prices, currDay+1, transaction, canBuy, memo);
13             int buy = -prices[currDay] + find(prices, currDay+1, transaction, !canBuy, memo);
14             return memo[currDay][canBuy][transaction] = max(buy, idle);
15         }
16         else
17         {
18            int idle = find(prices, currDay+1, transaction, canBuy, memo);
19            int sell = prices[currDay] + find(prices, currDay+1, transaction+1, !canBuy, memo);
20            return memo[currDay][canBuy][transaction] = max(sell, idle);
21         }
22     }
23     int maxProfit(vector<int>& prices) {
24         int n = prices.size();
25         vector<vector<vector<int>>> memo(n, vector<vector<int>>(2,vector<int>(2,-1)));
26         // canBuy = true and transactions are allowed 2 times
27         return find(prices, 0, 0, true, memo);
28     }
29 };
```

(19) Best time to Buy & Sell Stock - IV →

This is a generalised version of previous problem, instead of limiting it to 2 transactions, we need to allow atmost k transactions.

code →

Pass K as an argument & use it to limit transaction in base condition. (Line 6)

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, int transaction, int k, bool canBuy,
4      vector<vector<vector<int>>> &memo){
5
6          if(currDay >= prices.size() || transaction>=k) return 0;
7
8          if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];
9
10         if(canBuy)
11         {
12             int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
13             int buy = -prices[currDay] + find(prices, currDay+1, transaction, k, !canBuy, memo);
14             return memo[currDay][canBuy][transaction] = max(buy, idle);
15         }
16         else
17         {
18            int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
19            int sell = prices[currDay] + find(prices, currDay+1, transaction+1, k, !canBuy, memo);
20            return memo[currDay][canBuy][transaction] = max(sell, idle);
21         }
22     }
23     int maxProfit(int k, vector<int>& prices) {
24         int n = prices.size();
25         vector<vector<vector<int>>> memo(n ,vector<vector<int>>(2,vector<int>(k+1,-1)));
26         // canBuy = true and transactions are allowed atmost k times
27         return find(prices, 0, 0, k, true, memo);
28     }
29 };
```

(20) Best time to Buy & Sell Stock with Cooldown →

In this, cooldown means that we cannot buy a stock on the immediate day after it is sold.

=> the day after sold should be skipped.

code →

To skip day after sell, increment the currentDay by 2. (Line 18)

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){
4  
5          if(currDay >= prices.size()) return 0;
6  
7          if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8  
9          if(canBuy)
10         {
11             int idle = find(prices, currDay+1, canBuy, memo);
12             int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
13             return memo[currDay][canBuy] = max(buy, idle);
14         }
15         else
16         {
17            int idle = find(prices, currDay+1, canBuy, memo);
18            int sell = prices[currDay] + find(prices, currDay+2, !canBuy, memo);
19            return memo[currDay][canBuy] = max(sell, idle);
20         }
21     }
22     int maxProfit(vector<int>& prices) {
23         int n = prices.size();
24         vector<vector<int>> memo(n,vector<int> (2,-1));
25         // canBuy = true & transaction = infinite so ignore k & while sell, currDay +=2
26         return find(prices, 0, true, memo);
27     }
28 };
```

(21) Best time to Buy & Sell Stock with Transaction Fee →

In this variation, we donot have limit on transaction but while making a transaction ie selling it, some fee has to be paid ie transaction fee.

code →

Deduct the fee from the selling day's amount.

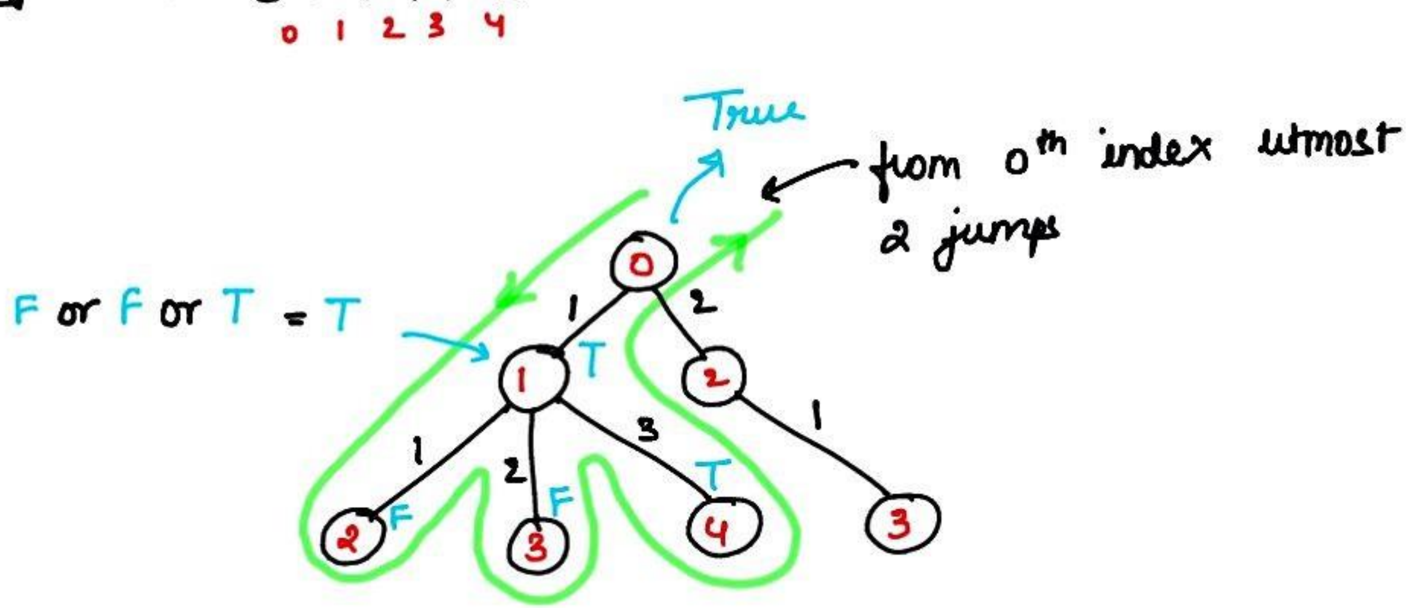
(Line 18)

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, int fee, bool canBuy, vector<vector<int>> &memo){
4  
5          if(currDay >= prices.size()) return 0;
6  
7          if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8  
9          if(canBuy)
10         {
11             int idle = find(prices, currDay+1, fee, canBuy, memo);
12             int buy = -prices[currDay] + find(prices, currDay+1, fee, !canBuy, memo);
13             return memo[currDay][canBuy] = max(buy, idle);
14         }
15         else
16         {
17            int idle = find(prices, currDay+1, fee, canBuy, memo);
18            int sell = (prices[currDay]-fee) + find(prices, currDay+1, fee, !canBuy, memo);
19            return memo[currDay][canBuy] = max(sell, idle);
20         }
21     }
22 
23     int maxProfit(vector<int>& prices, int fee) {
24         int n = prices.size();
25         vector<vector<int>> memo(n,vector<int> (2,-1));
26         // canBuy = true & transaction = infinite so ignore k & while selling deduce fee
27         return find(prices, 0, fee, true, memo);
28     }
29 };
```

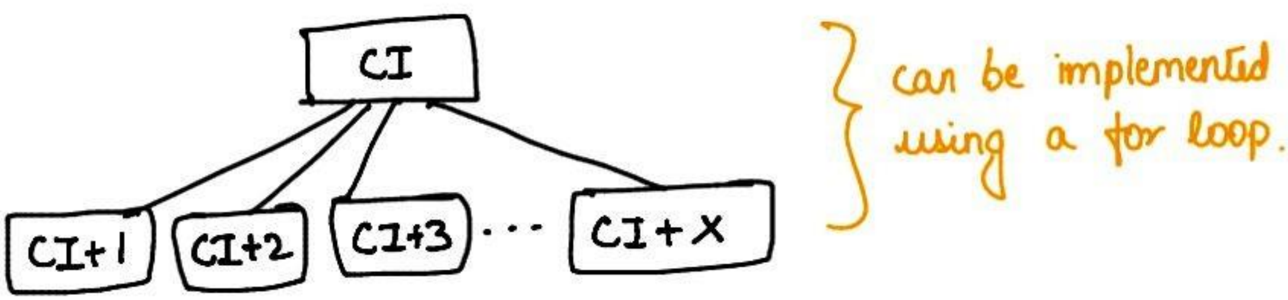
(22) Jump Game →

Given array of nums which indicate max number of jump from any index. Return true if you can reach last index.

Eg nums = [2, 3, 1, 1, 4]



Therefore, [_ _ $\frac{X}{CI}$ _ _ _]



Note: submitting DP solution gives TLE. This is just for understanding. Optimal solution involves Greedy approach.

$$Tc \rightarrow O\Big(\underbrace{max(nums[i])}_{\text{max time for forloop.}} * n\Big)$$

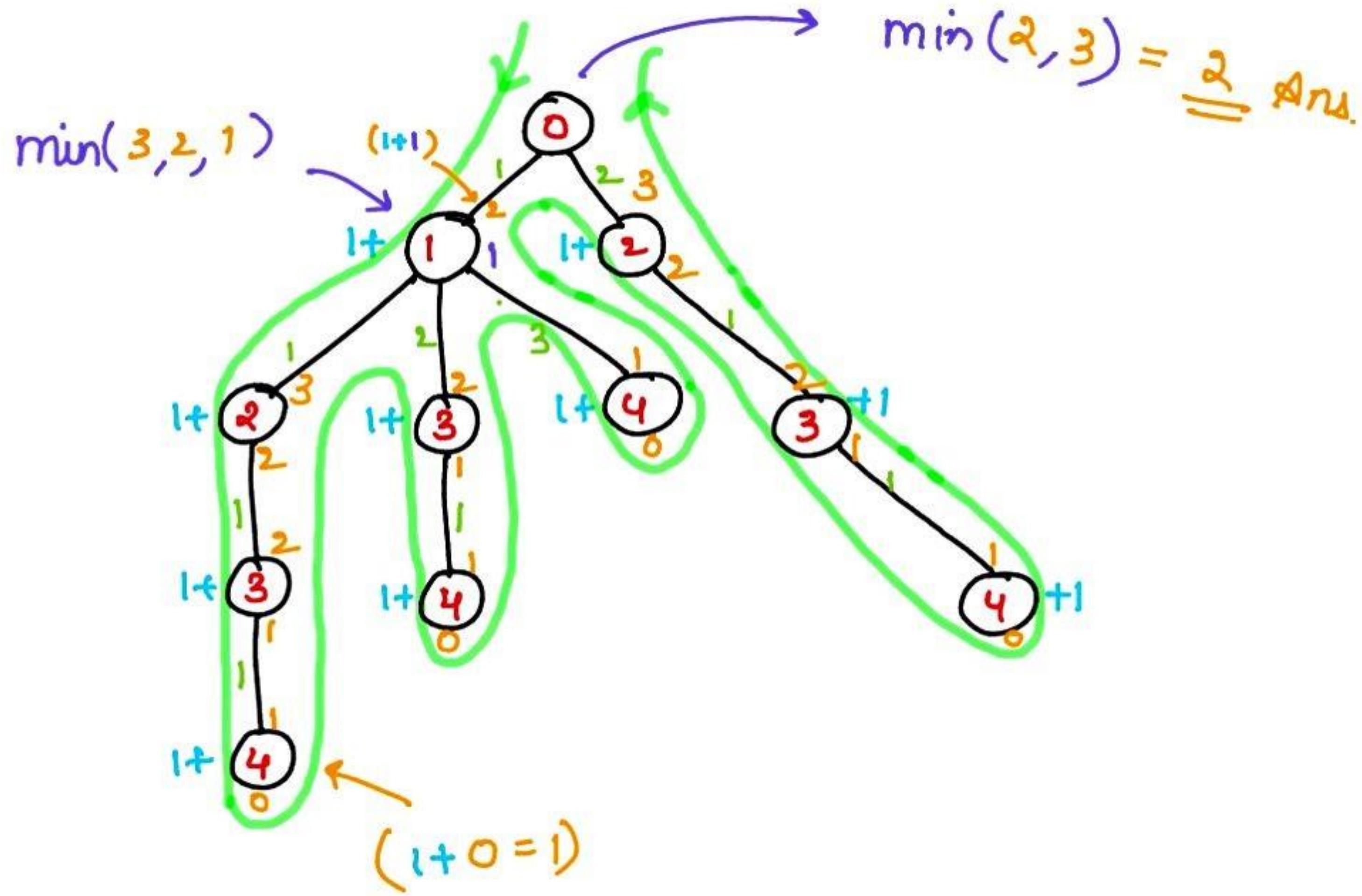
code →

```cpp
class Solution {
public:
    bool isPossible(vector<int>&nums, int curr, unordered_map<int,bool>&memo)
    {
        if(curr >= nums.size()-1) return true;

        int currKey = curr;

        if(memo.find(currKey)!=memo.end()) return memo[currKey];

        int currJump = nums[curr];

        if(currJump >= nums.size() - curr)  return true;

        bool ans = false;

        for(int i=1; i<=currJump; i++){
            bool tempAns = isPossible(nums,curr+i,memo);
            ans = ans || tempAns;
        }
        return memo[currKey] = ans;
    }

    bool canJump(vector<int>& nums){
        unordered_map<int,bool>memo;
        return isPossible(nums, 0, memo);
    }
};
```

(23) Jump Game II →

given array of nums which indicate max number of jump from any index. Reach last index in minimum number of moves.

Eg nums = [2, 3, 1, 1, 4]
0 1 2 3 4



→ If currentIndex >= lastIndex
then return 0.

while returning add 1 for counting ways!

Code →

```cpp
class Solution {
public:

    int minJumps(vector<int>& nums,int curr,vector<int>&memo)
    {
        if( curr >= nums.size()-1) return 0;

        int currKey = curr;
        if(memo[currKey]!=-1) return memo[currKey];

        int currJump = nums[curr];

        // some large value
        int ans = 10001;

        for(int i=1;i<=currJump;i++){
            int tempans = 1 + minJumps(nums,curr+i,memo);
            ans = min(ans, tempans);
        }
        return memo[currKey] = ans;
    }

    int jump(vector<int>& nums) {
        vector<int> memo(nums.size()+1,-1);
        return minJumps(nums, 0, memo);
    }
};
```

(24) Reach a given Score →

given 3 scores [3, 5, 10] & 'n'.
Return total number of ways to create n using the scores.

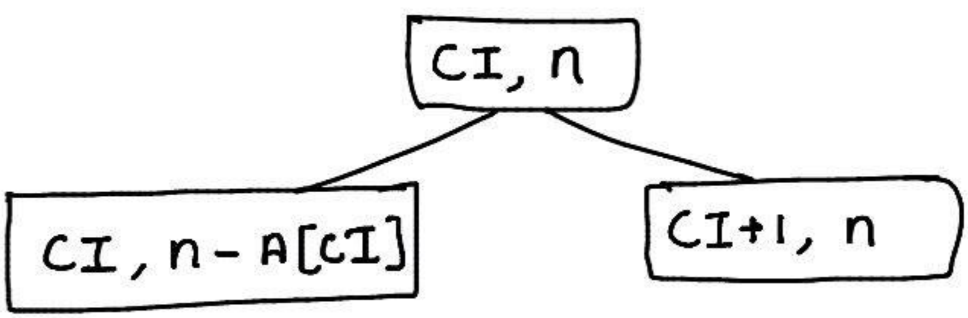
Eg $n = 8$ then no. of ways to create 8 from [3, 5, 10] is 1. (3+5)

$n = 13$ then no. of ways to create 13 from [3, 5, 10] is 2 (3+5+5) & (3+10)

$n = 20$ then no. of ways to create 20 from [3, 5, 10] is 4 (3+3+3+3+3+5) & (5+5+5+5) & (5+5+10) & (10+10)

∴ lets say $A = [3, 5, 10]$ then

↑
CI



code →

```cpp
typedef long long LL;

LL ways(int curr, LL n, vector<int>&score, vector<vector<int>>&vec)
{
    if(n==0) return 1;

    if(curr>=score.size()) return 0;

    if(vec[curr][n]!=-1) return vec[curr][n];

    LL consider = 0;

    if(score[curr]<=n)
      consider = ways(curr,n-score[curr],score,vec);

    LL notconsider = ways(curr+1,n,score,vec);

    return vec[curr][n] = consider + notconsider;
}

LL count(LL n)
{
    vector<int>score{3,5,10};
    vector<vector<int>>vec(score.size(),vector<int>(1001,-1));
    return ways(0,n,score,vec);
}
```

(25) Applications of Catalan Number →

Catalan Numbers are defined using the formula

$$C_n = \frac{(2n)!}{(n+1)!\, n!} = \prod_{k=2}^{n} \frac{n+k}{k} \quad \text{for } n \geq 0$$

this can be used recursively as follows,

$$C_{n+1} = \left.\sum_{i=0}^{n} C_i\, C_{i-1} \quad \right\} \; n \geq 0 \quad \& \quad C_0 = 1$$

→ $C_0 = 1.$

→ $C_1 = 1.$

→ $C_2 = C_0 \cdot C_1 + C_1 \cdot C_0 = 1 \cdot 1 + 1 \cdot 1 = 2.$

→ $C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = 5.$

→ $C_4 = C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0$

$= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 = 14.$

App^{n}s →

1. No. of possible BST with n keys.
2. No. of valid combinations for N pair of parenthesis.

26 N^{th} Catalan Number →

To find N^{th} catalan Number we can use formula

$$C_{n+1} = \sum_{i=0}^{n} C_i \, C_{i-1} \quad \Big\} \; n \geq 0 \;\; \& \;\; C_0 = 1$$

↳ this can be implemented by

i) having base condition for $n==0$ & $n==1$

ii) using a loop to sum values from $i=0$ to n

Code →

```cpp
class Solution
{
    public:
    cpp_int ncatalan(int n, vector<cpp_int>& memo) {
        if(n == 0 || n == 1) return 1;

        int curr = n;
        if(memo[curr]!=-1) return memo[curr];

        cpp_int catalan = 0;

        for(int i=0;i<n;i++) {
            catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
        }

        memo[curr] = catalan;
        return memo[curr];
    }

    cpp_int findCatalan(int n)
    {
        vector<cpp_int> memo(1001,-1);
        return ncatalan(n, memo);
    }
};
```

(27) Number of Valid Parenthesis Expression →

Given N, find total number of ways in which we can arrange N pair of parenthesis in a Balanced way.

Eg N = 6 ⇒ ( )( )( ), ( )(( )), (( ))( ), ((( ))) ∴ res = 4

( )( )( ) ⇒

( ) → ((( ))) , (( )( ))

nothing is outside

∴ C_0 & 2 types in ∴ C_2

⇒ $C_0 \cdot C_2$

\+

( ) → (( ))( ) , (( ))( )

identical &

one expression inside C_1

& one outside C_1

⇒ $C_1 \cdot C_1$

\+

( ) → ( )(( )) , ( )( )( )

both expressions outside ∴ C_2

& nothing inside ∴ C_0

⇒ $C_2 \cdot C_0$

⇒ $C_0 \cdot C_2 + C_1 \cdot C_1 + C_2 \cdot C_0 = C_3$ ⇒ for n we need to find ncatalan(n/2)

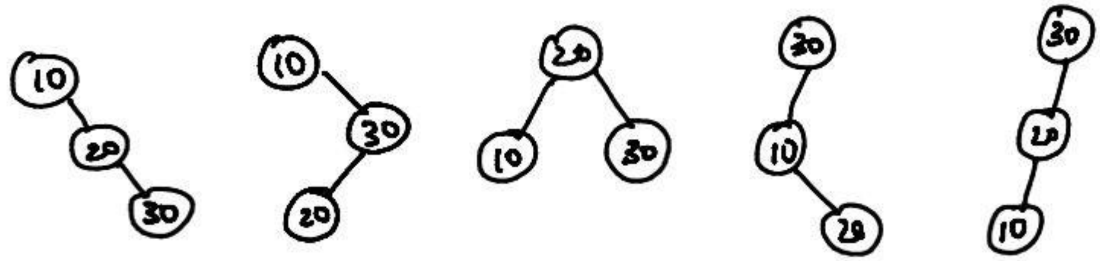
Code →

```cpp
#include<bits/stdc++.h>
using namespace std;

int ncatalan(int n, unordered_map<int,int>& memo) {
    if(n == 0 || n == 1) return 1;

    int curr = n;
    if(memo[curr]!=-1) return memo[curr];

    int catalan = 0;

    for(int i=0;i<n;i++) {
        catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
    }

    memo[curr] = catalan;
    return memo[curr];
}

int countValidParenthesis(int n)
{
    unordered_map<int,int> memo;
    return ncatalan(n/2, memo);
}

int main(){
    int n;
    cin>>n;
    cout<<countValidParenthesis(n);
}
```

(28) Unique Binary Search Trees →

given integer N, return no. of unique BST that can be formed.

Eg $n = 3$ & lets say elements are [10, 20, 30]

⇒ 

∴ For $n = 3$, the result is 5.

∴ The catalan number gives us the result.

code →

```cpp
class Solution {
public:

    int uniqueBST(int n, vector<int>& memo)
    {
        if(n==0||n==1) return 1;

        if( memo[n]!=-1)return memo[n];

        int ans = 0;
        for(int i=0;i<n;i++)
            ans += uniqueBST(i,memo)*uniqueBST(n-i-1,memo);

        return  memo[n] = ans;
    }

    int numTrees(int n) {
        vector<int> memo(n+1,-1);
        return uniqueBST(n, memo);
    }
};
```