# NeetCode Interview Questions List (Solved) 🔥✅

## Ft. Hritik Kumar

//contains Duplicate

->If the number has been seen previously ,then it has a duplicate, otherwise, insert it into the hash set.

```
        /*Time: O(n)
        Space: O(n)
        */
bool containsDuplicate (vector<int>&nums){
        int n = nums.size();
        unordered_set<int>st;

        for(int i=0;i<nums.size();i++){
                if(st.find(nums[i])!= st.end()){
                        return true;
                }
                st.insert(nums[i]);
        }
        return false;
}
```

//Valid Anagram

->hashmap solution

```
bool isAnagram(string s,string t){
        if(s.size()!=t.size())
        return false;

        unordered_map<char,int>smp;
        unordered_map<char,int>tmp;

        for(int i=0;i<s.size();i++){
                smp[s[i]]++;
                tmp[t[i]]++;
        }

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

```
            if(smp[i]!=tmp[i])
                return false;
        }
        return true;
}

//Two sum
->Given int array & target, return indices of 2 nums that add to target.
->At each num, calculate complement, if exists in hashmap then return.
/*  Time: O(n)
   Space: O(n)
*/
vector<int>twoSum(vector<int>&nums,int target){
        int n = nums.size();
        unordered_map<int,int>mp;

        for(int i=0;i<nums.size();i++){
            int sum = target-nums[i];
            if(mp.find(sum)!=mp.end()){
                return {mp[sum],i};
            }
            mp.insert({nums[i],i});
        }
        return {};
}

//Group Anagrams
->Given array of strings, group anagrams together (same letters, diff order)
Ex. strs = ["eat","tea","tan","ate","nat","bat"] -> [["bat"],["nat","tan"],["ate","eat","tea"]]
Hint:- Count chars, for each string use total char counts (naturally sorted) as key
/*  Time: O(n x l) -> n = length of strs, l = max length of a string in strs
   Space: O(n x l)
*/
vector<vector<string>>groupAnagrams(vector<string>&strs){
        unordered_map<string,vector<string>>mp;
        for(int i=0;i<strs.size();i++){
```

```
            string temp = getKey(strs[i]);
            mp[temp].push_back(strs[i]);
        }

        vector<vector<string>>ans;
        for(auto it = mp.begin();it!=mp.end();it++){
            ans.push_back(it->second);
        }
        return ans;
}
/*helper function*/
string getKey(string str){
        vector<string>count(26);
        for(int j=0;j<str.length();j++){
            count[str[j]-'a']++;
        }
        string temp = "";
        for(int i=0;i<count.size();i++){
            temp.append(to_string(count[i])+'#');
        }
        return temp;
}

//Top K Frequent Elements
Given an integer array nums & an integer k, return the k most frequent elements
   Ex. nums = [1,1,1,2,2,3] k = 2 -> [1,2], nums = [1] k = 1 -> [1]
Hint:- Heap -> optimize w/ freq map & bucket sort (no freq can be > n), get results from
end
/* Time: O(n log k)
  Space: O(n + k)
*/
vector<int>topKFrequent(vector<int>&nums,int k){
        unordered_map<int,int>mp;
        for(int i=0;i<nums.size();i++){
            mp[nums[i]]++;
        }
```

```
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        for(auto it = mp.begin();it!=mp.end();it++){
                pq.push({it->second,it->first});
                if(pq.size()>k){
                        pq.pop();
                }
        }
        vector<int>ans;
        while(!pq.empty()){
                ans.push_back(pq.top().second);
                pq.pop();
        }
        return ans;
}

/* Time: O(n) & Space: O(n)*/
vector<int>topKFrequent(vector<int>&nums,int k){
        int n = nums.size();
        unordered_map<int,int>mp;

        for(int i=0;i<n;i++){
                mp[nums[i]]++;
        }

        vector<vector<int>>buckets(n+1);
        for(auto it = mp.begin();it!=mp.end();it++){
                buckets[it->second].push_back(it->first);
        }
        vector<int>ans;

        for(int i=n;i>=0;i--){
                if(ans.size()>=k){
                        break;
                }
                if(!buckets[i].empty()){
                        ans.insert(ans.end(),buckets[i].begin(),buckets[i].end());
```

```
                }
        }
        return ans;
}
```

//Product of Array Except Self

->return an array such that:

answer[i] is equal to the product of all elements of nums except nums[i].

Hint:- Calculate prefix products forward, then postfix backwards in a 2nd pass.

```
/*
   Time: O(n)
   Space: O(1)
*/
vector<int>productExceptSelf(vector<int>&nums){
        int n = nums.size();
        vector<int>ans(n,1);

        int prefix = 1;
        for(int i=0;i<n;i++){
                ans[i] = prefix;
                prefix = prefix*nums[i];
        }
        int postfix = 1;
        for(int i=n-1;i>=0;i--){
                ans[i] = ans[i]*postfix;
                postfix = postfix*nums[i];
        }
        return ans;
}
```

//Valid Sudoku

-> Determine if a 9x9 Sudoku board is valid (no repeats)

Hint:- Boolean matrices to store seen values. Check rows, cols, 3x3 sub-boxes.

```
/*  Time:  O(cnt^2)
   Space: O(cnt^2)
*/
```

```
bool isValidSudoko(vector<vecotor<char>>&board){
        const int cnt = 9;
        bool row[cnt][cnt] = {false};
        bool col[cnt][cnt] = {false};
        bool sub[cnt][cnt] = {false};

        for(int r=0;r<cnt;r++){
                for(int c=0;c<cnt;c++){
                        if(board[r][c]=='.')
                                continue; //if not number pass

                        int idx = board[r][c]- '0' -1; //char to num idx
                        int area = (r/3)*3 + (c/3);

                        //if number already exists
                        if(row[r][idx] || col[c][idx] || sub[area][idx]){
                                return false;
                        }

                        row[r][idx] = true;
                        col[c][idx] = true;
                        sub[area][idx] = true;
                }
        }
        return true;
}

//Encode and Decode Strings
->Design algorithm to encode/decode: list of strings <-> string
Hint:- Encode/decode w/ non-ASCII delimiter: {len of str, "#", str}
/*
   Time: O(n)
   Space: O(1)
*/

/*Encodes a list of strings to a single string*/
```

```
string encode(vector<string>&strs){
        string res = "";

        for(int i=0;i<strs.size();i++){
                string str = strs[i];
                res += to_string(str.size()) + "#"+ str;
        }
        return res;
}
/*Decodes a single string to a list of strings*/
vector<string>decode(string s){
        vector<string>ans;

        int i=0;
        while(i<s.size()){
                int j=i;
                while(s[j]!='#'){
                        j++;
                }
                int length = stoi(s.substr(i,j-i));
                string str = s.substr(j+1,length);
                res.push_back(str);
                i = j+1+length;
        }
        return res;
}

//Longest Consecutive Sequence
->Given unsorted array, return length of longest consecutive sequence.
Hint:-  Store in hash set, only check for longer seq if it's the beginning.
/*  Time: O(n)
   Space: O(n)
*/
int longestConsecutive(vector<int>&nums){
        unordered_set<int>st(nums.begin(),nums.end());
        int longest = 0;
```

```
        for(auto &n:st){
                /*if this is the start of the sequence*/
                if(!st.count(n-1)){
                        int length = 1;
                        while(st.count(n+length))
                                length++;
                        longest = max(longest,length);
                }
        }
        return longest;
}
```

//Valid Palindrome

->Given a string s, return true if it's a palindrome

Ex. s = "A man, a plan, a canal: Panama" -> true

Hint: 2 pointers, outside in, skip non-letters & compare

```
/*  Time: O(n)
    Space: O(1)
*/
bool isPalindrome(string s){
        int i=0,j=s.size()-1;
        while(i<j){
                while(!isalnum(s[i])&& i<j){
                        i++;
                }
                while(!isalnum(s[j])&& i<j){
                        j++;
                }
                if(tolower(s[i]) != tolower[s[j]]){
                        return false;
                }
                i++,j--;
        }
        return true;
}
```

//Two Sum II Input Array Is Sorted

->Given a 1-indexed sorted int array & target: Return indices (added by 1) of 2 nums that add to target

Hint:- 2 pointers, outside in, iterate i/j if sum is too low/high

```
/*
   Time: O(n)
   Space: O(1)
*/
vector<int>twoSum(vector<int>nums,int target){
        int i=0,j=nums.size()-1;

        vector<int>ans;
        while(i<j){
                int sum = nums[i]+nums[j];
                if(sum<target){
                        i++;
                }
                else if(sum>target){
                        j--;
                }
                else{
                        ans.push_back(i+1);
                        ans.push_back(j+1);
                        break;
                }
        }
        return ans;
}
```

//3Sum

->Given int array, return all unique triplets that sum to 0.

   Ex. nums = [-1,0,1,2,-1,-4] -> [[-1,-1,2],[-1,0,1]]

Hint:- Sort,for each i, have j & k go outside in, check for 0 sums

```
/*  Time: O(n^2)
   Space: O(n)
```

```
*/
vector<vector<int>>3Sum(vector<int>&nums){
	vector<vector<int>>ans;
	int n = nums.size();
	if(n<3){
		return ans;
	}

	sort(nums.begin(),nums.end());

	for(int i=0;i<n-2;i++){
		if(nums[i]>0){
			break;
		}
		if(i>0 && nums[i-1]==nums[i]){
			continue;
		}

		int j=i+1,k = n-1;
		while(j<k){
			int sum = nums[i]+nums[j]+nums[k];

			if(sum<0){
				j++;
			}
			else if(sum>0){
				k--;
			}
			else{
				ans.push_back({nums[i],nums[j],nums[k]});
				while(j<k && nums[j]==nums[j+1]){
					j++;
				}
				j++;
				while(j<k && nums[k-1]==nums[k]){
					k--;
```

```
                        }
                        k--;
                    }
                }
            }
            return ans;
}

//Container With Most Water
-> Given array of heights, find max water container can store
Ex. height = [1,8,6,2,5,4,8,3,7] -> 49, (8 - 1) x min(8, 7)
Hint:- 2 pointers outside in, greedily iterate pointer w/ lower height
/*  Time: O(n)
   Space: O(1)
*/
int maxArea(vector<int>&height){
        int i=0,j=height.size()-1;
        int curr = 0;
        int ans =0;

        while(i<j){
            curr = (j-i)*min(height[i],height[j]);
            ans = max(ans,curr);

            if(height[i]<=height[j]){
                i++;
            }
            else{
                j--;
            }
        }
        return ans;
}

//Trapping Rain Water
->Given elevation map array, compute trapped water
```

Ex. height = [0,1,0,2,1,0,1,3,2,1,2,1] -> 6

Hint: - 2 pointers, outside in, track max left/right

For lower max, curr only dependent on that one

Compute height of these, iterate lower one

```
/*  Time: O(n)
   Space: O(1)
*/
int trap(vector<int>&height){
        int i=0,j= height.size()-1;
        int maxLeft = height[i];
        int maxRight = height[j];

        int ans =0;
        while(i<j){
                if(maxLeft<=maxRight){
                        i++;
                        maxLeft = max(maxLeft,height[i]);
                        ans+= maxLeft-height[i];
                }
                else{
                        j--;
                        maxRight = max(maxRight,height[j]);
                        ans+= maxRight-height[j];
                }
        }
        return ans;
}
```

//Best Time to Buy And Sell Stock

->Given array prices, return max profit w/ 1 buy & 1 sell

Hint:- For each, get diff b/w that & min value before, store max

```
/*  Time: O(n)
   Space: O(1)
*/
int maxProfit(vector<int>*prices){
        int maxProfit = 0;
```

```
        int l =0,r =0;

        while(r<prices.size()){
                if(prices[r]>prices[l])
                        maxProfit = max(maxProfit,prices[r]-prices[l]);
                else
                         l= r;
                r++;
        }
        return maxProfit;
}
```

//Longest Substring Without Repeating Characters

->Given string, find longest substring w/o repeating chars

Ex. s = "abcabcbb" -> 3 "abc", s = "bbbbb" -> 1 "b"

Hint:- Sliding window, expand if unique, contract if duplicate

```
/*   Time: O(n)
    Space: O(n)
*/
int lengthOfLongestSubstring(string s){
        unordered_set<char>st;

        int i=0,j=0;
        int ans =0;

        while(j<s.size()){
                if(st.find(s[j])==st.end()){
                        st.insert(s[j]);
                        ans = max(ans,j-i+1);
                        j++;
                }
                else{
                        st.erase(s[i]);
                        i++;
                }
```

```
        }
        return ans;
}

/***2nd method***/
->Below: inner loop increasing "i", outer loop increasing "j".
int lengthOfLongestSubstring(string s){
        unordered_set<char>st;
        int maxSize =0;
        int i=0,j=0;
        while(j<s.size()){
                while(st.find(s[j])!= st.end()){
                        st.erase(s[i]);
                        i++;
                }
                maxSize = max(maxSize,j-i+1);
                st.insert(s[j]);
                j++;
        }
        return maxSize;
}

//Longest Repeating Character Replacement
->Given a string s & an int k, can change any char k times: Return length of longest
substring containing same letter
  Ex. s = "ABAB" k = 2 -> 4 "AAAA", s = "AABABBA" k = 1 -> 4 "BBBB"
Hint:-  Sliding window, expand if can change char, contract if > k
/*  Time:  O(n)
   Space: O(26)
*/
int characterReplacement(string s,int k){
        vector<int>count(26);
        int maxCount =0;

        int i=0,j=0;
        int ans =0;
```

```
        while(j<s.size()){
                count[s[j]-'A']++;
                maxCount = max(maxCount,count[s[j]-'A']);
                if(j-i+1-maxCount>k){
                        count[s[i]-'A']--;
                i++;
        }
        ans = max(ans,j-i+1);
        j++;
}
        return ans;
}

//Permutation In String
->Given 2 strings, return true if s2 contains permutation of s1
   Ex. s1 = "ab", s2 = "eidbaooo" -> true, s2 contains "ba"
Hint:- Sliding window, expand + count down char, contract + count up char
/*
   Time: O(n)
   Space: O(1)
*/

bool checkInclusion(string s1,string s2){
        int m = s1.size();
        int n = s2.size();

        if(m>n){
                return false;
        }

        vector<int>count(26);
        for(int i=0;i<m;i++){
                count[s1[i]-'a']++;
                count[s2[i]-'a']--;
        }
```

```
        if(isPermutation(count)){
                return true;
        }

        for(int i=m;i<n;i++){
                count[s2[i]-'a']--;
                count[s1[i-m]-'a']++;
                if(isPermutation(count)){
                        return true;
                }

                return false;
        }
/*helper function*/
        bool isPermutation(vector<int>&count){
                for(int i=0;i<26;i++){
                        if(count[i]!=0){
                                return false;
                        }
                }
                return true;
        }

//Minimum Window Substring
->Given 2 strings s & t, return min window substring of s such that all chars in t are
included in window
  Ex. s = "ADOBECODEBANC" t = "ABC" -> "BANC"
Hint:- Sliding window + hash map {char -> count}
 Move j until valid, move i to find smaller
/*    Time: O(m + n)
     Space: O(m + n)
*/
string minWindow(string s,string t){
        // count of char in t
        unordered_map<char,int>mp;
        for(int i=0;i<t.size();i++){
```

```
            mp[t[i]]++;
        }

        int i=0,j=0;
        // # of chars in t that must be in s
        int count = t.size();

        int minStart =0;
        int minLength = INT_MAX;

        while(j<s.size()){
            //if char in s exists in t, decrease
            if(mp[s[j]]>0){
                count--;
            }
            // if char doesn't exist in t, will be -'ve
            mp[s[j]]--;
            j++;

        // when window found, move i to find smaller
        while(count==0){
            if(j-i<minLength){
                minStart = i;
                minLength = j-i;
            }

            mp[s[i]]++;
            // when char exists in t, decrease
            if(mp[s[i]]>0){
                count++;
            }
            i++;
        }
}
if(minLength()!=INT_MAX){
        return s.substr(minStart,minLength);
```

```
}
        return "";
}

//Sliding Window Maximum
-> Given int array & sliding window size k, return max sliding window
 Ex. nums = [1,3,-1,-3,5,3,6,7] k = 3 -> [3,3,5,5,6,7]
Hint:-  Sliding window deque, ensure monotonic decr, leftmost largest
/*  Time: O(n)
   Space: O(n)
*/
vector<int>maxSlidingWindow(vector<int>&nums,int k){
        deque<int>dq;
        vector<int>ans;

        int i=0,j=0;

        while(j<nums.size()){
                while(!dq.empty() && nums[dq.back()<nums[j]]){
                        dq.pop_back();
                }
                dq.push_back(j);

                if(i>dq.front()){
                        dq.pop_front();
                }

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

//Valid Parentheses

-> Given s w/ '(, ), {, }, [, ]', determine if valid

Ex. s = "()[]{}" -> true, s = "(]" -> false

Hint:- Stack of opens, check for matching closes & validity

```
/*  Time: O(n)
    Space: O(n)
*/
bool isValid(string s){
        stack<char>st;

        unordered_map<char,char>mp = {
        {')','('},
        {']','['},
        {'{','}'},

        };

        for(const auto &c:s){
                if(mp.find(c)!=mp.end()){
                        if(st.empty()){
                                return false;
                        }

                        if(st.top()!=mp[c]){
                                return false;
                        }
                        st.pop();
                }
                        else{
                                st.push(c);
                        }
                }
        return st.empty();
}
```

//Min Stack

-> Design stack that supports push, pop, top, & retriving min element

Hint:- 2 stacks, 1 normal & 1 monotonic decr, only push if lower than top

```
/*  Time: O(1)
   Space: O(n)
*/

class MinStack{
        Public:
                MinStack(){

                }

        void push(int val){
                st.push(val);

                if(stk.empty() || val<stk.top().first){
                        stk.push({val,1});
                }
                else if(val==stk.top().first){
                        stk.top().second++;
                }
        }

        void pop(){
                if(st.top()==stk.top().first){
                        stk.top().second--;
                        if(stk.top().second==0){
                                stk.pop();
                        }
                }
                st.pop();
        }

        int top(){
                return st.top();
        }
```

```
        int getMin(){
                return stk.top().first;
        }
        private:
                stack<int>st;
                stack<pair<int,int>>stk;
};

//Evaluate Reverse Polish Notation
->Evaluate RPN, valid operators: +, -, *, /
Hint:- Stack, if num push, if operator apply to top 2 nums
/*  Time: O(n)
   Space: O(n)
*/
int evalRPN(vector<string>&tokens){
        stack<int>st;

        for(int i=0;i<tokens.size();i++){
                string token = tokens[i];

                if(tokens.size()>1 && isDigit(token[0])){
                        st.push(stoi(token));
                        continue;
                }

                int num2 = st.top();
                st.pop();
                int num1 = st.top();
                st.pop();

                int ans =0;
                if(token=='+'){
                        ans = num1+num2;
                }
                else if(token=="-"){
```

```
                ans = num1-num2;
            }
            else if(token=="*"){
                ans = num1*num2;
            }
            else{
                ans = num1/num2;
            }
            st.push(ans);
        }
        return st.top();
}
```

//Generate Parentheses

-> Given n pairs of parentheses, generate all combos of well-formed parentheses.

Ex. n = 3 -> ["((()))","(()())","(())()","()(())","()()()"], n = 1 -> ["()"]

Hint:- Backtracking, keep valid, favor trying opens, then try closes if still valid

```
/*  Time: O(2^n)
   Space: O(n)
*/
vector<string>generateParanthesis(int n){
        vector<string>ans;
        generate(n,0,0,"",ans);
        return ans;
}
void generate(int n,int open,int close,string str,vector<string>&ans){
        if(open == n && close==n){
            ans.push_back(str);
        return;
}
if(open<n){
        generate(n,open+1,close,str+'(',ans);
}
if(open>close){
        generate(n,open,close+1,str+')',ans);
        }
```

```
}

//Daily Temperatures
->Given array of temps, return an array w/ # of days until warmer
   Ex. temperature = [73,74,75,71,69,72,76,73] -> [1,1,4,2,1,1,0,0]
   Hint:- Monotonic decr stack, at each day, compare incr from prev days
   /*   Time: O(n)
        Space: O(n)
   */
vectot<int>dailyTemp(vector<int>&temp){
        int n = temp.size();
        // pair: [index, temp]
        stack<pair<int,int>>st;
        vector<int>ans(n);

        for(int i=0;i<n;i++){
                int currDay = i;
                int currTemp = temp[i];

                while(!st.empty() && st.top().second< currTemp){
                        int prevDay = st.top().first;
                        int prevTemp = st.top().second;
                        st.pop();

                        ans[prevDay] = currDay - prevDay;
                }
                st.push({currDay,currTemp});
        }
        return ans;
}

//Car Fleet
->A car fleet is some non-empty set of cars driving at the same position and same speed.
Return the number of car fleets that will arrive at the destination.
-> n cars 1 road, diff pos/speeds, faster cars slow down -> car fleet, return # fleets
Ex. target = 12, pos = [10,8,0,5,3], speeds = [2,4,1,1,3] -> 3 (10 & 8, 0, 5 & 3)
```

Hint:- Sort by start pos, calculate time for each car to finish, loop backwards
If car behind finishes faster then catches up to fleet, else creates new fleet

```
/*  Time: O(n log n)
   Speed: O(n)
*/

int carFleet(int target,vector<int>&position,vector<int>&speed){
        int n = position.size();
        vector<pair<int,double>>cars;
        for(int i=0;i<n;i++){
                double time = (double)(target-position[i])/speed[i];
                cars.push_back({position[i],time});
        }
        sort(cars.begin(),cars.end());

        double maxTime = 0.0;
        int ans =0;

        for(int i=n-1;i>=0;i--){
                double time = cars[i].second;
                if(time>maxTime){
                        maxTime = time;
                        ans++;
                }
        }
        return ans;
}
```

//Largest Rectangle In Histogram

->Given array of heights, return area of largest rectangle.

Ex. heights = [2,1,5,6,2,3] -> 10 (5 x 2 at index 2 and 3)

Hint:- Monotonic incr stack, if curr height lower extend back, find max area

```
/* Time: O(n)
  Space: O(n)
*/
```

```
int largestRectangleArea(vector<int>&heights){
        //pair: [index,height]
        stack<pair<int,int>>st;
        int ans =0;

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

                while(!st.empty() && st.top().second>heights[i]){
                        int index = st.top().first;
                        int height = st.top().second;
                        int width = i-index;
                        st.pop();

                        ans = max(ans,height*width);
                        start = index;
                }
                st.push({start,heights[i]});
        }
        while(!st.empty()){
                int width = heights.size() - st.top().first;
                int height = st.top().second;
                st.pop();
                ans = max(ans,height*width);
        }
        return ans;
}

// Given sorted int array, search for a target value
   Ex. nums = [-1,0,3,5,9,12], target = 9 -> 4 (index)
Hint:- since array is sorted, perform binary search
/*  Time: O(log n)
   Space: O(1)
*/

int search(vector<int>&nums,int target){
```

```
        int l =0,h= nums.size()-1;
        while(l<=h){
                int mid = l+(h-l)/2;
                if(nums[mid]<target){
                        l = mid+1;
                }
                else if(nums[mid]>target){
                        h = mid-1;
                }
                else{
                        return mid;
                }
        }
        return -1;
}

//Search for target value in matrix where every row & col is (sorted)
Hint:- Perform 2 binary searches:- 1 to find "Row", then another to find "Col"
/*
   Time: O(log m + log n)
   Space: O(1)
*/
bool searchMatrix(vector<vector<int>>&matrix,int target){
        int lRow = 0;
        int hRow = matrix.size()-1;

        while(lRow<hRow){
                int mid = lRow+(hRow-lRow)/2;
                if(matrix[mid][0]==target){
                        return true;
                }

                if(matrix[mid][0] <target && target<matrix[mid+1][0]){
                        lRow = mid;
                        break;
                }
```

```
            if(matrix[mid][0]<target){
                lRow = mid+1;
            }
            else{
                hRow = mid-1;
            }
        }
        int lCol =0;
        int hCol = matrix[0].size()-1;

        while(lCol<=hCol){
            int mid = lCol + (hCol-lCol)/2;
            if(matrix[lRow][mid]==target){
                return true;
            }
            if(matrix[lRow][mid]<target){
                lCol = mid+1;
            }
            else{
                hCol = mid-1;
            }
        }
        return false;
}
```

//Koko Eating Bananas

->Given array of banana piles, guards are gone for h hours ,Return min int k such that can eat all banans within h

Ex. piles = [3,6,7,11] h = 8 -> 4 (1@3, 2@6, 2@7, 3@11)

Hint:- Binary search, for each k count hours needed, store min

```
/* Time: O(n x log m) -> n = # of piles, m = max # in a pile
   Space: O(1)
*/

int minEatingSpeed(vector<int>&piles,int h){
```

```
        int n = pile.size();
        int low = 1,high = 0;

        for(int i=0;i<n;i++){
                high = max(high,piles[i]);
        }
        int ans = high;

        while(low<=high){
                int k = low+(high-low)/2;
                long int hours =0;
                for(int i=0;i<n;i++){
                        hours+= ceil((double)piles[i]/k);
                }
                if(hours<=h){
                        ans = min(ans,k);
                        high = k-1;
                }
                else{
                        low = k+1;
                }
        }
        return ans;
}

//Find Minimum In Rotated Sorted Array
Hint :- implement Binary Search
/* Ologn ->time  O(1) ->space*/
int findMin(vector<int>&nums){
        int ans = nums[0];
        int l =0,r = nums.size()-1;

        while(l<=r){
                if(nums[l]<nums[r]){
                        ans= min(ans,nums[l]);
                        break;
```

```
                }
                int mid = l+(r-l)/2;
                ans = min(ans,nums[mid]);

                if(nums[mid]>=nums[l]){
                        l = mid+1;
                }
                else{
                        r = mid-1;
                }
        }
        return ans;
}

//Search In Rotated Sorted Array
-> Given array after some possible rotation, find if target is in nums
Ex. nums = [4,5,6,7,0,1,2] target = 0 -> 4 (value 0 is at index 4)
Hint:- Modified binary search, if low <= mid left sorted, else right sorted
/*  Time: O(log n)
   Space: O(1)
*/
int search(vector<int>&nums,int target){
        int low =0;
        int high = nums.size()-1;

        while(low<=high){
                int mid = low+(high-low)/2;
                if(nums[mid]==target){
                        return mid;
                }
                if(nums[low]<=nums[mid]){
                        if(nums[low]<=target && target <=nums[mid]){
                                high = mid-1;
                }
                        else{
                           low = mid+1;
```

```
                }
        }
        else{
                if(nums[mid]<=target && target<=nums[high]){
                        low = mid+1;
                }
                else{
                        high = mid-1;
                }
        }
}
        return -1;
}

//Time Based Key Value Store
->Design time-based key-value structure, multiple vals at diff times.
Hint:- Use Hashmap ,since timestamps are naturally in order, follow binary search
/*  Time: O(log n)
   Space: O(n)
*/

class TimeMap(){
        public:
                TimeMap(){

                }

                void set(string key,string value,int timestamp){
                        mp[key].push_back({timestamp,value});
                }
                string get(string key,int timeStamp){
                        if(mp.find(key)==mp.end()){
                                return "";
                        }

                        int low = 0;
```

```
                int high = mp[key].size()-1;

                while(low<=high){
                        int mid = low+(high-low)/2;

                        if(mp[key[mid].first<timestamp){
                                low = mid+1;
                        }
                        else if(mp[key][mid].first>timestamp){
                                high = mid-1;
                        }
                        else{
                                return mp[key][mid].second;
                        }
                }

                if(high>=0){
                        return mp[key][high].second;
                }
                return "";
        }
private:
        unordered_map<string,vector<pair<int,string>>mp;
};
```

//Median of Two Sorted Arrays

->Given 2 sorted arrays of size m & n, return the median of these arrays.

Ex. nums1 = [1,3] nums2 = [2] -> 2, nums1 = [1,2] nums2 = [3,4] -> 2.5

Hint:- Binary search, partition each array until partitions are correct, get median

```
        [1,2,3,4,5]
   |  a|b    |
   [1,2,3,4,5,6,7,8]    -->    a <= d ? yes, c <= b ? no, so need to fix
   |      c|d      |
```

/*Time: O(log min(m, n))

```
   Space: O(1)
*/

double findMedianSortedArrays(vector<int>&nums1,vector<int>&nums2){
        int m = nums1.size();
        int n = nums2.size();

        if(m>n){
                return findMedianSortedArrays(nums2,nums1);
        }

        int total = m+n;

        int low=0,high = m;

        double ans = 0.0;

        while(low<=high){
                int i = low+(high-low)/2; //nums1
                int j = (total+1)/2-i; //nums2

                int left1 = (i>0)?nums1[i-1]:INT_MIN;
                int right1 = (i<m)?nums1[i]:INT_MAX;

                int left2 = (j>0)?nums2[j-1]:INT_MIN;
                int right2 = (j<n)?nums2[j]:INT_MAX;

                //partition is correct
                if(left1<=right2 && left2<=right1){
                        //even
                        if(total%2==0){
                                ans = (max,(left1,left2)+min(right1,right2))/2.0;
                        }
                        //odd
                        else{
                                ans = max(left1,left2);
```

```
                        }
                        break;
                }
                else if(left1>right2){
                        high = i-1;
                }
                else{
                        low = i+1;
                }
        }
        return ans;
}

//Implement Trie Prefix Tree
-> Implement trie (store/retrieve keys in dataset of strings)
Hint:- Each node contains pointer to next letter & is word flag
/*  Time: O(n) insert, O(n) search, O(n) startsWith
   Space: O(n) insert, O(1) search, O(1) startsWith
*/

class Trie{
        public:
                Trie(){
                        root = new TrieNode();
                }

                void insert(string word){
                        TrieNode*node = root;
                        int curr =0;

                        for(int i=0;i<word.size();i++){
                                curr = word[i]-'a';
                                if(node->children[curr]==NULL){
                                        node->children[curr] = new TrieNode();
                                }
                                node = node->children[curr];
```

```
            }
            node->isWord = true
        }

        bool search(string word){
            TrieNode*node = root;
            int curr =0;

            for(int i=0;i<word.size();i++){
                curr = word[i]-'a';
                if(node->children[curr]==NULL){
                    return false;
                }
                node = node->children[curr];
            }
            return node->isWord;
        }

        bool startsWith(string prefix){
            TrieNode*node = root;
            int curr =0;

            for(int i=0;i<prefix.size();i++){
                curr = prefix[i]-'a';
                if(node->children[curr]==NULL){
                    return false;
                }
                node = node->children[curr];
            }
            return true;
        }
    Private:
        TrieNode*root;
};
```

//Design Add And Search Words Data Structure

Hint:-Implement trie, handle wildcards: traverse all children & search substrings.

```
/*
   Time: O(m x 26^n) -> m = # of words, n = length of words
   Space: O(n)
*/

class TrieNode(){
        public:
                TrieNode*children[26];
                bool isWord;

                TrieNode(){
                        for(int i=0;i<26;i++){
                                children[i]==NULL;
                        }
                        isWord = false;
                }
};

class WordDictionary{
        public:
                wordDictionary(){
                        root = new TrieNode();
                }

                void addWord(string word){
                        TrieNode*node = root;
                        int curr =0;

                        for(int i=0;i<word.size();i++){
                                curr = word[i]-'a';
                                if(node->children[curr]==NULL){
                                        node->children[curr] = new TrieNode();
                                }
                                node = node->children[curr];
```

```
                }
                node->isWord = true;
        }

    bool search(string word){
        TrieNode*node = root;
        return searchInNode(word,0,node);
    }
    private:
        TrieNode*root;

        bool searchInNode(string &word,int i,TrieNode*node){
            if(node==NULL){
                return false;
            }
            if(i==word.size()){
                return node->isWord;
            }

            if(word[i]!='.'){
                return searchInNode(word,i+1,node->children[word[i]-'a']);
            }

            for(int j=0;j<26;j++){
                if(searchInNode(word,i+1,node->children[j])){
                    return true;
                }
            }
            return false;
        }
};

//Word Search II
-> Given a board of characters & a list of words, return all words on the board
Hint:- Implement trie, for search: iterate through children until isWord, add to result
/* Time: O(m x (4 x 3^(l - 1))) -> m = # of cells, l = max length of words
```

```
   Space: O(n) -> n = total number of letters in dictionary (no overlap in Trie)
*/
Class TrieNode{
		TrieNode*children[26];
		bool isWord;

		TrieNode(){
			for(int i=0;i<26;i++){
				children[i] = NULL;
		}
		isWord = false;
	}
};
class Solution{
public:
	vector<string>findWords(vector<vector<char>>&board,vector<string>&words){
		for(int i=0;i<words.size();i++){
			insert(words[i]);
		}
		int m = boards.size();
		int n = boards[0].size();

		TrieNode*node = root;
		vector<string>ans;

		for(int i=0;i<m;i++){
			for(int j=0;j<n;j++){
				search(board,i,j,m,n,node,"",ans);
			}
		}
		return ans;
	}
private:
	TrieNode*root = new TrieNode();

	void insert(string word){
```

```
        TrieNode*node = root;
        int curr =0;

        for(int i=0;i<word.size();i++){
                curr = word[i]-'a';
                if(node->children[curr]==NULL){
                        node->children[curr] = new TrieNode();
                }
                node = node->children[curr];
        }
        node->isWord = true;
    }

    void search(vector<vector<char>>&word,int i,int j,int m,int n,TrieNode*node,string
word,vector<string>&ans){
        if(i<0 || i>=m || j<0 || j>=n || board[i][j]=='#'){
        return;
        }

        char c = board[i][j];
        node = node->children[c-'a'];
        if(node==NULL){
                return;
        }

        word+= board[i][j];
        if(node->isWord){
                ans.push_back(word);
                node->isWord = false;
        }

        board[i][j] = '#';

        search(board,i-1,j,m,n,node,word,ans);
        search(board,i+1,j,m,n,node,word,ans);
        search(board,i,j-1,m,n,node,word,ans);
```

```
                search(board,i,j+1,m,n,node,word,ans);

                board[i][j] = c;
        }
};
```

//Search a 2D Matrix

-> Search for target value in matrix where every row & col is 'sorted'.

Hint:- Perform 2 binary searches: 1 to find row, then another to find col

```
/*Time: O(log m + log n)
  Space: O(1)
*/

bool searchMatrix(vector<vectot<int>>&matrix,int target){
        int lRow = 0;
        int hRow - matrix.size()-1;

        while(lRow<hRow){
                int mid = lRow+(hRow-lRow)/2;
                if(matrix[mid][0]==target){
                        return true;
                }
                if(matrix[mid][0]<target && target<matrix[mid+1][0]){
                        lRow = mid;
                        break;
                }

                if(matrix[mid][0]<target){
                        lRow  = mid+1;
                }
                else{
                        hRow = mid-1;
                }
        }
```

```
        int lCol =0;
        int hCol = matrix[0].size()-1;

        while(lCol<=hCol){
                int mid = lCol + (hCol-lCol)/2;
                if(matrix[lRow][mid]==target){
                        return true;
                }
                if(matrix[lRow][mid]<target){
                        lCol = mid+1;
                }
                else{
                        hCol = mid-1;
                }
        }
        return false;
}

//Kth Largest Element In a Stream
->Design a class to find the kth largest element in a stream
Hint:- Min heap & maintain only k elements, top will always be kth largest.
Ex. nums = [6,2,3,1,7], k = 3 -> [1,2,3,6,7] -> [3,6,7]
/*  Time: O(n log n + m log k) -> n = length of nums, m = add calls
   Space: O(n)
*/

class KthLargest{
        public:
                kthLargest(vector<int>&nums,int k){
                        this->k = k;
                        for(int i=0;i<nums.size();i++){
                                pq.push(nums[i]);
                        }
                while(pq.size()>this->k){
                        pq.pop();
                        }
```

```
        }

    int add(int val){
        pq.push(val);
        if(pq.size()>k){
            pq.pop();
        }
        return pq.top();
    }
private:
    int k;
    priority_queue<int,vector<int>,greater<int>>pq;
};
```

//Last Stone Weight

-> Given array of stones to smash, return smallest possible weight of last stone.

-> If x == y both stones destroyed, if x != y ->stone x destroyed, stone y = y - x.

-> At the end of the game, there is at most one stone left.

-> Return the weight of the last remaining stone. If there are no stones left, return 0.

Ex:- stones = [2,7,4,1,8,1] -> 1, [2,4,1,1,1], [2,1,1,1], [1,1,1], [1]

Hint:- Max heap, pop 2 biggest, push back difference until no more 2 elements left

```
/* Time: O(n log n)
   Space: O(n)
*/
int lastStoneWeight(vector<int>&nums){
    priority_queue<int>pq;
    for(int i=0;i<stones.size();i++){
        pq.push(stones[i]);
    }

    while(pq.size()>k){
        int y = pq.top();
        pq.pop();
        int x = pq.top();
        pq.pop();
        if(y>x){
```

```
                        pq.push(y-x);
                }
        }
        if(pq.empty()){
                return 0;
        }
        return pq.top();
}
```

//K Closest Points to Origin

->Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k.

->distance between two points on the X-Y plane is Euclidean distance (√(x1 - x2)2 + (y1 - y2)2).

->return the k closest points to the origin (0, 0).

```
/*Time: O(k log n)
  Space: O(n)
*/
vector<vector<int>>kClosest(vector<vector<int>>&points,int k){
        vector<vector<int>>triples;
        for(auto &p:points){
                triples.push_back({p[0]*p[0]+p[1]*p[1],p[0],p[1]});
        /*Min heap of vectors (triples). This constructor takes O(n) time (n = len(v)) */

priority_queue<vector<int,vector<vector<int>>,greater<vector<int>>>pq(triples.begin(),triples
.end());
        vector<vector<int>>ans;
        while(k--){
                vector<int>el = pq.top();
                pq.pop();
                ans.push_back({el[1],el[2]});
                }
                return ans;
        }
}
```

//Kth Largest Element In An Array

->Given array and int k, return kth largest element in array

Ex:- Ex. nums = [3,2,1,5,6,4], k = 2 -> 5

/*Time: O(n) -> optimized from O(n log k)

Space: O(1)

*/

Hint:- Min Heap solution

```
int findKthLargest(vector<int>&nums,int k){
        priority_queue<int,vector<int>,greater<int>>pq;
        for(int i=0;i<nums.size();i++){
                pq.push(nums[i]);
                if(pq.size()>k){
                        pq.pop();
                }
        }
        return pq.top();
}
```

//Task Scheduler

-> Given array of tasks & cooldown b/w same tasks, return minimum number of intervals required to complete all tasks.

Ex. tasks = ["A","A","A","B","B","B"] n = 2 -> 8 (A->B->idle->A->B->idle->A->B)

Hint:- Key is to determine # of idles, greedy: always arrange task w/ most freq first

```
/*  Time: O(n*cooldown)
   Space: O(1)
*/
int leastInterval(vector<char>&tasks,int n){
        priority_queue<int>pq;
        queue<vector<int>>q;
        vector<int>counter(26);

        for(int i=0;i<tasks.size();i++)
                ++counter[tasks[i]-'A'];
        for(int i=0;i<26;i++){
                if(counter[i])
                 pq.push(counter[i]);
```

```
    }

        int time =0;
        while(!q.empty()|| !pq.empty()){
            ++time;
            if(!pq.empty()){
                if(pq.top()-1)
                    q.push({pq.top()-1,time+n});
                pq.pop();
            }
            if(!q.empty() && q.front()[1]==time){
                pq.push(q.front()[0]);
                q.pop();
            }
        }
        return time;
    }
```

//Design Twitter

->Design a simplified version of Twitter where users can post tweets, follow/unfollow another user, and is able to see the 10 most recent tweets in the user's news feed.

Hint:- Maintain user -> tweet pairs & hash map {user -> ppl they follow}

```
/* Time: O(n)
   Space: O(n)
*/
class Twitter{
    public:
        Twitter(){

        }

    void postTweet(int userId,int tweetId){
        posts.push_back({userId,tweetId});
    }

    vector<int>getNewsFeed(int userId){
```

```
            int count = 10; //10 tweets
            vector<int>ans;

        // since postTweet pushes to the back, looping from back gets most recent
            for(int i= posts.size()-1;i>=0;i--){
                if(count==0){
                    break;
                }

                int followingId = posts[i].first;
                int tweetId = posts[i].second;

                unordered_set<int>following = followMap[userId];

        // add to result if they're following them or it's a tweet from themself
            if(following.find(followingId)!= following.end() || followingId == userId){
                ans.push_back(tweetId);
                count--;
                }
            }
            return ans;
        }

        void follow(int followerId,int followeeId){
            followMap[followerId].insert(followeeId);
        }

        void unfollow(int followerId,int followeeId){
            followMap[followerId].erase(followeeId);
        }
 private:
        //pairs: [user, tweet]
        vector<pair<int,int>>posts;
        // hash map: {user -> people they follow}
        unordered_map<int,unordered_set<int>>followMap;
};
```

//Find Median From Data Stream

->Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

For example, for arr = [2,3,4], the median is 3.

For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

->Implement data structure that gets the median from a data stream

Hint:- Max heap of lower values & min heap of higher values, access to mids

```
/*  Time: O(log n) + O(1)
   Space: O(n)
*/
class MedianFinder{
        public:
                MedianFinder(){

                }

        void addNum(int num){
                if(lower.empty()){
                        lower.push_back(num);
                        return;
                }

                if(lower.size()>higher.size()){
                        if(lower.top()>num){
                                higher.push(lower.top());
                                lower.pop();
                                lower.push(num);
                        }
                        else{
                                higher.push(num);
                        }
                }
                else{
                        if(num>higher.top()){
                                lower.push_back(higher.top());
```

```
                        higher.top();
                        higher.push(num);
                }
                else{
                        lower.push(num);
                }
        }
    }

    double findMedian(){
        double ans = 0.0;

    if(lower.size()== higher.size()){
        ans = lower.top() +(higher.top()-lower.top())/2.0;
    }
    else{
        if(lower.size()>higher.size()){
                ans = lower.top();
        }
        else{
                ans = higher.top();
        }
    }
    return ans;
    }
private:
    priority_queue<int>lower;
    priority_queue<int,vector<int>,greater<int>>higher;
};
```

//Insert Interval

-> Given array of non-overlapping intervals & a new interval, insert & merge if necessary

Ex. intervals = [[1,3],[6,9]], newInterval = [2,5] -> [[1,5],[6,9]]

Hint:- To merge: while intervals are still overlapping the new one, take the larger bounds

```
/*  Time: O(n)
   Space: O(n)
```

```
*/
vector<vector<int>>insert(vector<vector<int>>&intervals,vector<int>&newInterval){
        vector<vector<int>>ans;
        int newStart = newInterval[0];
        int newEnd = newInterval[1];

        int n = intervals.size();
        for(int i=0;i<n;i++){
                /***case 1: Non Overlapping interval***/
                /* If new interval is before the current interval*/
                if(intervals[i][0]>newEnd){
                        ans.push_back(newInterval);
                        copy(intervals.begin()+i,intervals.end(),back_inserter(ans));
                        return ans;
                }
                /* if new interval is after the current interval*/
                else if(intervals[i][1]<newStart){
                        ans.push_back(intervals[i]);
                }

                /***case 2: Overlapping interval***/
                else{
                        newInterval[0] = min(newInterval[0],intervals[i][0]);
                        newInterval[1] = max(newInterval[1],intervals[i][1]);
                }
        }
        ans.push_back(newInterval);
        return ans;
}

//Merge Intervals
-> Given an array of intervals, merge all overlapping intervals
Ex. intervals = [[1,3],[2,6],[8,10],[15,18]] -> [[1,6],[8,10],[15,18]]
Hint:- Sort by earliest start time, merge overlapping intervals (take longer end time)
/* Time: O(n log n)
   Space: O(n)
```

```
*/

vector<vector<int>>merge(vector<vector<int>>&intervals){
        int n = intervals.size();
        if(n==1)
                return intervals;

        /*lambda function used as a custom comparator for sorting*/
        /*This is the comparison criterion used by the lambda function. It compares the
first element of each interval*/
        sort(intervals.begin(),intervals.end(),[](const auto& a ,const auto& b){
                return a[0]<a[b];
        });

        vector<vector<int>>ans;

        int i=0;
        while(i<n-1){
                if(intervals[i][1] >= intervals[i+1][0]){
                        intervals[i+1][0] = intervals[i][0];
                        intervals[i+1][1] = max(intervals[i][1],intervals[i+1][1]);
                }
                else{
                        ans.push_back(intervals[i]);
                }
                i++;
        }
        ans.push_back(intervals[i]);
        return ans;
}
```

//Non Overlapping Intervals

-> Given array of intervals where intervals[i] = [starti, endi], return min no. of intervals to remove for all non-overlapping

Ex. intervals = [[1,2],[1,3],[2,3],[3,4]] -> 1, remove [1,3] for all non-overlapping

Hint:- Remove interval wrt. longer end point, since will always overlap more or = vs shorter one

```
/*  Time: O(n log n)
   Space: O(1)
*/

int eraseOverlapIntervals(vector<vector<int>>&intervals){
        int n = intervals.size();

        if(n==1)
                return intervals;

        /*lambda function used as a custom comparator for sorting*/
        /*This is the comparison criterion used by the lambda function*/
        sort(intervals.begin(),intervals.end(),[](const auto& a,const auto& b){
                return a[0]<b[0];
        });

        int ans =0;
        int i=0;

        while(i<n-1){
                /* if the end point of the current interval intervals[i] is greater than the start point of the next interval intervals[i + 1]*/
                if(intervals[i][1]>intervals[i+1][0]){
                /* if the end point of the current interval intervals[i] is less than the end point of the next interval intervals[i + 1]*/
                        if(intervals[i][1]<intervals[i+1][1]){
                /* In this case, we update the next interval intervals[i + 1] to be the same as the current interval intervals[i]. This effectively merges the two intervals*/
                                intervals[i+1] = intervals[i];
                        }
                        ans++;
                }
                i++;
        }
```

```
        return ans;
}

//Meeting Rooms
->Given array of time intervals, determine if can attend all meetings
Ex. intervals = [[0,30],[5,10],[15,20]] -> false
Hint:-  Sort by start time, check adj meetings, if overlap return false
/*  Time: O(n log n)
   Space: O(1)
*/

bool canAttendMeetings(vector<vector<int>>&intervals){
        if(intervals.empty()){
                return true;
        }

        sort(intervals.begin(),intervals.end());
        for(int i=0;i<intervals.size()-1;i++){
                if(intervals[i][1]>intervals[i+1][0]){
                        return false;
                }
        }
        return true;
}

//Meeting Rooms II
->Given array of time intervals, determine min no. of meeting rooms required
Ex. intervals = [[0,30],[5,10],[15,20]] -> 2
Hint:- Min heap for earliest end times, most overlap will be heap size
/*  Time: O(n log n)
   Space: O(n)
*/
int minMeetingRooms(vector<vector<int>>&intervals){
        int n = intervals.size();
        // sort intervals by start time
        sort(intervals.begin(),sort(intervals.end());
```

```
        // min heap to track min end time of merged intervals
        priority_queue<int,vector<int>,greater<int>>>pq;
        pq.push(intervals[0][1]);

        for(int i=1;i<intervals.size();i++){
        // compare curr start wrt/ earliest end time, if no overlap then pop
        if(intervals[i][0]>=pq.top()){
                pq.pop();
        }

        //add new room
        pq.push(intervals[i][1]);
        }
        return pq.size();
}

//Minimum Interval to Include Each Query
->Given intervals array & queries array, ans to a query is min interval containing it
 Ex. intervals = [[1,4],[2,4],[3,6],[4,4]], queries = [2,3,4,5] -> [3,3,1,4]
Hint:- Min heap & sort by size of intervals, top will be min size,
/*  Time: O(n log n + q log q) -> n = number of intervals, q = number of queries
   Space: O(n + q)
*/
vector<int>minInterval(vector<vector<int>>&intervals,vector<int>&queries){
        vector<int>sortedQueries = queries;

        //[size of interval, end of interval]
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        // {query -> size of interval}
        unordered_map<int,int>mp;

        //also need only valid intervals so sort by start time & sort queries
        sort(intervals.begin(),intervals.end());
        sort(sortedQueries.begin(),sortedQueries.end());
```

```
        vector<int>ans;

        int i=0;
        for(int j=0;j<sortedQueries.size();j++){
                int query = sortedQueries[j];

                while(i<intervals.size() && intervals[i][0]<=query){
                        int left = intervals[i][0];
                        int right = intervals[i][1];
                        pq.push({right-left+1,right});
                        i++;
                }

                while(!pq.empty() && pq.top().second<query){
                        pq.pop();
                }

                if(!pq.empty()){
                        mp[query] = pq.top().first;
                }
                else{
                        mp[query] = -1;
                }
        }

        for(int j=0;j<queries.size();j++){
                ans.push_back(mp[queries[j]]);
        }
        return ans;
}
```

//Maximum Subarray

-> Given int array, find contiguous subarray wrt/ max sum

Ex. nums = [-2,1,-3,4,-1,2,1,-5,4] -> 6, [4,-1,2,1]

Hint:- At each point, determine if it's better to add to curr sum or start over.

```
/*  Time: O(n)
   Space: O(1)
*/
int maxSubarray(vector<int>&nums){
        int n = nums.size();
        int curr = nums[0];
        int ans = nums[0];

        for(int i=1;i<nums.size();i++){
                curr = max(curr+nums[i],nums[i]);
                ans = max(ans,curr);
        }
        return ans;
}
```

//Jump Game

-> Given int array, return true if can reach last index

Ex. nums = [2,3,1,1,4] -> true, index 0 to 1 to last

Hint:- Greedy Technique: At each point, determine furthest reachable index

```
/*  Time: O(n)
   Space: O(1)
*/
bool canJump(vector<int>&nums){
        int n = nums.size();
        int reach =0;

        for(int i=0;i<n;i++){
                if(i>reach)
                        return false;

                reach = max(reach,i+nums[i]);
                if(reach>=n-1)
                        break;
        }
        return true;
```

```
}
```

//Jump Game II

->Given int array, determine min jumps to reach last index

Ex. nums = [2,3,1,1,4] -> 2, index 0 to 1 to last

Hint:- Greedy technique: At each point, determine furthest reachable, jump to it

```
/*  Time: O(n)
   Space: O(1)
*/
int jump(vector<int>&nums){
        int n = nums.size();
        int ans =0;

        int i=0;
        while(i<n-1){
                if(i+nums[i]>=n-1){
                        ans++;
                        break;
                }
                int maxIndex = i+1;
                int maxValue = 0;
                for(int j=i+1;j<i+1+nums[i];j++){
                        if(j+nums[j]>maxValue){
                                maxIndex = j;
                                maxValue = j+nums[j];
                        }
                }
                i = maxIndex;
                ans++;
        }
        return ans;
}
```

//Gas Station

->Gas stations along circular route, return where to start to complete 1 trip.

Ex. gas = [1,2,3,4,5] cost = [3,4,5,1,2] -> index 3 (station 4), tank = 4,8,7,6,5

Hint:- At a start station, if total ever becomes negative won't work, try next station

```
/*
   Time: O(n)
   Space: O(1)
*/
int canCompleteCircuit(vector<int>&gas,vector<int>&cost){
        int n = gas.size();
        int totalGas = 0,totalCost =0;

        for(int i=0;i<n;i++){
                totalGas += gas[i];
                totalCost += cost[i];
        }

        if(totalGas<totalCost)
                return -1;

        int total =0;
        int ans = 0;

        for(int i=0;i<n;i++){
                total += gas[i]-cost[i];
                if(total<0){
                        total = 0;
                        ans = i+1;
                }
        }
        return ans;
}
```

//Hand of Straights

-> Given int array, return true if can rearrange cards into "groupSize consecutive"

Ex. hand = [1,2,3,6,2,3,4,7,8], groupSize = 3 -> true, [1,2,3],[2,3,4],[6,7,8]

Hint:- Loop through ordered map, for a value, check groupSize consecutive & remove

```
/*Time: O(n log n)
  Space: O(n)
*/
bool isNstraightHand(vector<int>&hand,int groupSize){
        int n = hand.size();

        if(n % groupSize!=0){
                return false;
        }

        //map {card value -> count}
        map<int,int>mp;
        for(int i=0;i<n;i++){
                mp[hand[i]]++;
        }

        while(!mp.empty()){
                int curr = mp.begin().first;
                for(int i=0;i<groupSize();i++){
                        if(mp[curr+i]==0)
                                return false;

                        mp[curr+i]--;
                        if(mp[curr+i]<1)
                                mp.erase(curr+1);
                }
        }
        return true;
}
```

//Merge Triplets to Form Target Triplet

->Update: [max(ai,aj), max(bi,bj), max(ci,cj)], return if possible to obtain target

Ex. triplets = [[2,5,3],[1,8,4],[1,7,5]] target = [2,7,5] -> true, Update "1st/3rd"

Hint:- Skip all "bad" triplets (can never become target), if match add to "good" set.

```
 /*Time: O(n)
   Space: O(1)
*/
bool mergeTriplets(vector<vector<int>>&triplets,vector<int>&target){
        int n = triplets.size();
        unordered_set<int>st;

        for(int i=0;i<n;i++){
                if(triplets[i][0]>target[0] || triplets[i][1]>target[1] ||
triplets[i][2]>target[2]){
                        continue;
                }

                for(int j=0;j<3;j++){
                        if(triplets[i][j]==target[j]){
                                st.insert(j);
                        }
                }
        }
        return st.size()==3;
}
```

//Partition Labels

->Partition string so each letter appears in at most 1 part, return sizes

Ex. s = "ababcbacadefegdehijhklij" -> [9,7,8]

Hint:- Greedy technique: determine last occurrence of each char, then loop through & get sizes

```
/*  Time: O(n)
    Space: O(1)
*/
vector<int>partitionLables(string s){
        int n = s.size();
        //{char -> last index in s}
        vector<int>lastIndex(26);
```

```
        for(int i=0;i<n;i++){
                lastIndex[s[i]-'a'] = i;
        }

        int size =0;
        int end  =0;
        vector<int>ans;

        for(int i=0;i<n;i++){
                size++;
        // constantly checking for further indices if possible
                end = max(size,lastIndex[s[i]-'a']);
                if(i==end)
                        ans.push_back(size);
                        size = 0;
        }
        return ans;
}

//Valid Parenthesis String
->To check the valid condition of paranthesis occurences in a string
bool checkValid (string s){
        int n = s.size();
        int balanced = 0;

        for(int i=0;i<n;i++){
                if(s[i]=='('|| s[i]=='*')
                        balanced++;
                else
                        balanced--;

                if(balanced<0)
                        return false;
        }

        if(balanced==0)
```

```
                return true;

        balanced = 0;
        for(int i=n-1;i>0;i--){
                if(s[i]==')'|| s[i]=='*')
                        balanced++;
                else
                        balanced--;

                if(balanced<0)
                        return false;
        }
        return true;
}
```

//Subsets

-> Given an integer array of unique elements, return all possible subsets (the power set)

Ex. nums = [1,2,3] -> [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Hint:- Backtracking, generate all combinations, push/pop + index checking to explore new combos

```
/*Time: O(n x 2^n)
  Space: O(n)
*/
vector<vector<int>>subsets(vector<int>&nums){
        int n = nums.size();
        vector<int>curr;
        vector<vector<int>>ans;
        dfs(nums,0,curr,ans);
        return ans;
}
void dfs(vector<int>&nums,int start,vector<int>&curr,vector<vector<int>>&ans){
        ans.push_back(curr);

        for(int i= start;i<nums.size();i++){
                curr.push_back(nums[i]);
```

```
                dfs(nums,i+1,curr,ans);
                curr.pop_back();
        }
}
```

//Combination Sum

->Given distinct int array & a target, return list of all unique combos that sum to target

Ex. candidates = [2,3,6,7] target = 7 -> [[2,2,3],[7]]

Hint:- Backtracking, generate all combo sums, push/pop + index checking to explore new combos

```
/*Time: O(2^target)
  Space: O(target)
*/
vector<vector<int>>combinations(vector<int>&candidates,int target){
        int n = candidates.size();
        vector<int>curr;
        vector<vector<int>>ans;
        dfs(candidates,target,0,curr,ans);
        return ans;
}
void dfs(vector<int>candidates,int target,int
indx,vector<int>&curr,vector<vector<int>>&ans){
        if(i>=candidates.size() || target<0)
                return;

        if(target==0)
                ans.push_back(curr);
                return;

        curr.push_back(candidates[i]);

        dfs(candidates,target-candidates[i],indx,curr,ans);
        curr.pop_back();

        dfs(candidates,target,indx+1,curr,ans);
```

```
}
```

//Permutations

-> Given array of distinct integers, return all the possible permutations

Ex. nums = [1,2,3] -> [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Hint:- Permute by swapping i/start, DFS from this point, backtrack to undo swap

```
/*
   Time: O(n x n!)
   Space: O(n!)
*/
vector<vector<int>>permute(vector<int>&nums){
        int n = nums.size();
        vector<vector<int>>ans;
        dfs(nums,0,ans); //indexing is 0-based
        return ans;
}
void dfs(vector<int>&nums,int indx,vector<vector<int>>&ans){
        if(indx==nums.size())
                ans.push_back(nums);
                return;

        for(int i=indx;i<nums.size();i++){
                swap(nums[i],nums[indx]);
                dfs(nums,indx+1,ans);
                swap(nums[i],nums[indx]);
        }
}
```

//Subsets II (without duplicates(skip))

-> Given an integer array of unique elements, return all possible subsets (the power set)

Ex. nums = [1,2,2] -> [[],[1],[1,2],[1,2,2],[2],[2,2]]

Hint:- Backtracking, generate all combos, push/pop + to explore new combos, skip duplicates

```
/*
```

```
   Time: O(n x 2^n)
   Space: O(n)
*/
vector<vector<int>>subsetsWithDuplicate(vector<int>&nums){
        int n = nums.size();
        sort(nums.begin(),nums.end());
        dfs(nums,0,curr,ans);
        vector<int>curr;
        vector<vector<int>>ans;
        return ans;
}
void dfs(vector<int>&nums,int indx,vector<int>&curr,vector<vector<int>>&ans){
        ans.push_back(curr);
        for(int i=indx;i<nums.size();i++){
                if(i>indx && nums[i]==nums[i-1]){
                        continue;
        }
        curr.push_back(nums[i]);
        dfs(nums,indx+1,curr,ans);
        curr.pop_back();
        }
}
```

//Combination Sum II

-> Given array & a target, find all unique combos that sum to target, nums can only be used once

Ex. candidates = [10,1,2,7,6,1,5], target = 8 -> [[1,1,6],[1,2,5],[1,7],[2,6]]

Hint:- Backtracking, generate all combo sums, push/pop + index checking to explore new combos

```
/*Time: O(2^n)
  Space: O(n)
*/
vector<vector<int>>combinations(vector<int>&candidates,int target){
        int n = candidates.size();
        sort(candidates.begin(),candidates.end());
```

```
        vector<int>curr;
        vector<vector<int>>ans;
        dfs(candidates,0,target,curr,ans,0); /*indexing = 0 & sum = 0 initially*/
        return ans;
}
void dfs(vector<int>&candidates,int indx,int
target,vector<int>&curr,vector<vector<int>>&ans,int sum){
        if(sum>target)
                return;

        if(sum==target)
                ans.push_back(curr);
                return;

        for(int i= indx;i<candidates.size();i++){
                if(i>indx && candidates[i]==candidates[i-1]){
                        continue;
                }

                curr.push_back(candidates[i]);
                dfs(candidates,indx+1,target,curr,ans,sum+candidates[i]);
                curr.pop_back();
        }
}

//Word Search
->Given a char board & a word, return true if word exists in the grid
Hint:- Use DFS traversal, set visited cells to '#', search in 4 directions, Backtrack
/*  Time: O(n x 3^l) -> n = # of cells, l = length of word
   Space: O(l)
*/
bool wordExist(vector<vector<char>>&board,string word){
        int n = board.size();
        int m = board[0].size();

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

```
                for(int j=0;j<m;j++){
                        if(board[i][j]== word[0]){
                                if(dfs(board,word,0,i,j,n,m)){
                                        return true;
                                }
                        }
                }
        }
        return false;
}
void dfs(vector<vector<char>>&board,string word,int indx,int i,int j,int n,int m){
        /***edge case***/
        if(i<0 ||i>=n || j<0|| j>=m || board[i][j]!= word[indx]){
                return false;
        }
        /***edge case***/
        if(indx==word.size()-1){
                return false;
        }

        board[i][j] = '#';

        if(dfs(board,word,indx+1,i-1,j,n,m)  ||
                dfs(board,word,indx+1,i+1,j,n,m) ||
                dfs(board,word,indx+1,i,j-1,n,m) ||
                dfs(board,word,indx+1,i,j+1,n,m)
         ){
                return true;
         }
         board[i][j] = word[indx];
         return false;
}
```

//Palindrome Partitioning

->Given a string, partition such that every substring is a palindrome, return all possible ones

Ex. s = "aab" -> [["a","a","b"],["aa","b"]], s = "a" -> [["a"]]

Hint:- Generate all possible substrings at idx, if palindrome potential candidate, backtrack after

```
/*  Time: O(n x 2^n)
   Space: O(n)
*/
vector<vector<string>>partition(string s){
        int n = s.size();
        vector<string>curr;
        vector<vector<string>>ans;
        dfs(s,0,curr,ans);
        return ans;
}
void dfs(string s, int indx,vector<string>&curr,vector<vector<string>>&ans){
        if(indx==s.size()){
                ans.push_back(curr);
                return;
        }
        for(int i=indx;i<s.size();i++){
                if(isPalindrome(s,indx,i)){
                        string str = s.substr(indx,i-indx+1);
                        curr.push_back(str);
                        dfs(s,i+1,curr,ans);

                        curr.pop_back();
                }
        }
}
bool isPalindrome(string s,int left,int right){
        while(left<right){
                if(s[left]!=s[right]){
                        return false;
                }
                left++,right--;
        }
}
```

```
        return true;
}

//Letter Combinations of a Phone Number
-> Given cell phone pad, return all possible letter combos that the number could represent
  Ex. digits = "23" -> ["ad","ae","af","bd","be","bf","cd","ce","cf"]
Hint:- Hash map all digits to letters, add 1 letter at a time for each digit, then backtrack undo
/*   Time: O(n x 4^n)
    Space: O(n x 4^n)
*/
vector<string>letterCombinations(string digits){
        int n = digits.size();
        if(digits.empty()){
                return {};
        }

        unordered_map<char,string>mp = {
                {'2',"abc"},
                {'3',"def"},
                {'4',"ghi"},
                {'5',"jkl"},
                {'6',"mno"},
                {'7',"pqrs"},
                {'8',"tuv"},
                {'9',"xyz"};
        };

        string curr = "";
        vector<string>ans;
        dfs(digits,0,mp,curr,ans);
        return ans;
}
void dfs(string digits,int indx,unordered_map<char,string>mp,string &curr,vector<string>&ans){
        if(indx==digits.size()){
```

```
                ans.push_back(curr);
                return;
        }
        string str = mp[digits[indx]];
        for(int i=0;i<str.size();i++){
                curr.push_back(str[i]);
                dfs(digits,indx+1,mp,curr,ans);

                curr.pop_back();
        }
}
```

//N Queens

->N-Queens: place n queens such that no 2 queens atk each other, return all soln's

Hint:- Place queens per row, try all possibilities & validate for further rows, backtrack

```
/*  Time: O(n!)
   Space: O(n^2)
*/
vector<vector<string>solveNQueens(int n){
        vector<vector<string>>ans;
        vector<string>board(n,string(n,'.'));
        backtrack(n,0,ans,board);
        return ans;
}
//helper function
unordered_set<int>cols; //for Columns
unordered_set<int>negDiag; //for negative diagnals R-C
unordered_set<int>posDiag; //for positive diagnals R+C

void backtrack(int n, int row,vector<vector<string>>&ans,vector<string>&board){
        if(row==n){
                ans.push_back(board);
                return;
        }

        for(int col = 0;col<n;col++){  //Shifting through each col
```

```
                if(cols.find(col)!= cols.end() //if queen alread placed in this col
                || negDiag.find(row-col) != negDiag.end() //if queen in negDiag
                || posDiag.find(row+col) != posDiag.end()) //if queen in posDiag
                        continue;

                cols.erase(col);
                negDiag.erase(row-col);
                posDiag.erase(row+col);
                board[row][col] = 'Q';

                backtrack(n,row+1,ans,board);

                cols.erase(col);
                negDiag.insert(row-col);
                posDiag.insert(row+col);
                board[row][col] = '.';
        }
}
```

//Reverse Linked List

->Given the head of a singly linked list, reverse list & return

Ex. head = [1,2,3,4,5] -> [5,4,3,2,1], head = [1,2] -> [2,1]

Hint:- Maintain prev, curr pointers, iterate through & reverse

```
/*  Time: O(n)
   Space: O(1)
*/
ListNode*reverseList(ListNode*head){
        if(head==NULL || head->next==NULL)
                return head;

        ListNode*prev = NULL;
        ListNode*curr = head;

        while(curr!=NULL){
                ListNode*temp = curr->next;
                curr->next = prev;
```

```
            prev = curr;
            curr = temp;
        }
        return prev;
}
```

//Merge Two Sorted Lists

->Given heads of 2 sorted linked lists, merge into 1 sorted list

Ex. list1 = [1,2,4], list2 = [1,3,4] -> [1,1,2,3,4,4]

Hint:- Create curr pointer, iterate through, choose next to be lower one

```
/*  Time: O(m + n)
   Space: O(1)
*/
ListNode*mergeLists(ListNode*l1,ListNode*l2){
        if(l1==NULL && l2==NULL){
            return NULL;
        }

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

        ListNode*dummy = new ListNode();
        ListNode*curr = dummy;

        while(l1!=NULL && l2!=NULL){
            if(l1->val <=l2->val){
                curr->next = l1;
                l1 = l1->next;
            }
            else{
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
```

```
        }

        if(l1==NULL)
                curr->next = l2;
        else
                curr->next = l1;

        return dummy->next;
}
```

//Reorder List

-> Given head of linked-list, reorder list alternating outside in

Ex. head = [1,2,3,4] -> [1,4,2,3], head = [1,2,3,4,5] -> [1,5,2,4,3]

Hint:- Find middle node, split in half, reverse 2nd half of list, merge

```
/*  Time: O(n)
   Space: O(1)
*/
void reorderList(ListNode*head){
        if(head->next==NULL)
                return;

        ListNode*prev = NULL;
        ListNode*slow = head;
        ListNode*fast = head;

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

        prev->next = NULL;

        ListNode*l1 = head;
        ListNode*l2 = reverse(slow);
```

```
        merge(l1,l2);
}
private:
        ListNode*reverse(ListNode*head){
                ListNode*prev = NULL;
                ListNode*curr = head;
                ListNode*next = curr->next;

                while(curr!=NULL){
                        next = curr->next;
                        curr->next = prev;
                        prev = curr;
                        curr = next;
                }
                return prev;
        }
        void merge(ListNode*l1,ListNode*l2){
                while(l1!=NULL){
                        ListNode*p1 = l1->next;
                        ListNode*p2 = l2->next;

                l1->next = l2;
                if(p1==NULL)
                        break;

                l2->next = p1;

                l1 = p1;
                l2 = p2;
                }
        }
```

//Remove Nth Node From End of List

-> Given head of a linked list, remove nth node from end of list

Ex. head = [1,2,3,4,5], n = 2 -> [1,2,3,5]

Hint:- Create 2 pointers "n" apart, iterate until end, will be at nth

```
/*  Time: O(n)
   Space: O(1)
*/
ListNode*removeNthFrontEnd(ListNode*head,int n){
        if(head->next==NULL)
                return NULL;

        ListNode*slow = head;
        ListNode*fast = head;

        while(n>0){
                fast = fast->next;
                n--;
        }

        if(fast==NULL){
                return head->next;
        }

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

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

//Copy List With Random Pointer

-> Given linked list w/ also a random pointer, construct deep copy

Hint:-  Hash map {old -> new}, O(n) space

Optimize interweave old and new nodes, O(1) space

A -> A' -> B -> B' -> C -> C', A'.random = A.random.next

```
/* Time: O(n)
   Space: O(n) -> can optimize to O(1)
*/
```

```
Node*copyRandomList(Node*head){
        unordered_map<Node*,Node*>mp;
        Node*h = head;

        while(h){
                mp[h] = new Node(h->val);
                h = h->next;
        }
        h = head;
        while(h){
                Node*newNode = mp[h];
                newNode->next = mp[h->next];
                newNode->random = mp[h->random];
                h = h->next;
        }
        return mp[head];
}
```

//Add Two Numbers

-> Given 2 linked lists, digits stored in reverse order, add them

Ex. l1 = [2,4,3] l2 = [5,6,4] -> [7,0,8] (342 + 465 = 807)

Hint:- Sum digit-by-digit + carry, handle if one list becomes null

```
/*  Time: O(max(m, n))
   Space: O(max(m, n))
*/
ListNode*addNumbers(ListNode*l1, ListNode*l2){
        ListNode*dummy = new ListNode();
        ListNode*curr = dummy;
        int carry =0;

        while(l1!=NULL || l2!=NULL){
                int val1 = (l1!=NULL)?l1->val:0;
                int val2 = (l2!=NULL)?l2->val:0;

                int sum = val1+val2+carry;
```

```
            carry = sum/10;

            curr->next = new ListNode(sum%10);
            curr = curr->next;

            if(l1!=NULL)
                l1 = l1->next;
            if(l2!=NULL)
                l2 = l2->next;
        }

        if(carry ==1){
            curr->next = new ListNode(1);
        }

        return dummy->next;
}
```

//Linked List Cycle

-> Given head of a linked list, determine if it has a cycle in it

Hint:- Slow/fast pointers, if they ever intersect then there's a cycle

```
/*  Time: O(n)
   Space: O(1)
*/
bool hasCycle(ListNode*head){
        if(head==NULL)
            return false;

        ListNode*slow = head;
        ListNode*fast = head;

        while(fast->next!=NULL && fast->next->next!=NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow==fast){
                return true;
```

```
                }
        }
        return false;
}

//Find The Duplicate Number
-> Given int array, return the one repeated number
Ex. nums = [1,3,4,2,2] -> 2, nums = [3,1,3,4,2] -> 3
Hint:- If there's duplicate, must be a cycle, find meeting point
   Take 1 back to start, they'll intersect at the duplicate
/*  Time: O(n)
   Space: O(1)
*/
int findDuplicate(vector<int>&nums){
        int n = nums.size();
        int slow = nums[0];
        int fast = nums[nums[0]];

        while(slow!=fast){
                slow = nums[slow];
                fast = nums[nums[fast]];
        }
        slow = 0;
        while(slow!=fast){
                slow = nums[slow];
                fast = nums[fast];
        }
        return slow;
}

//LRU Cache (IMP)
->Design data structure that follows constraints of an LRU cache
Hint:-  Hash map + doubly linked list, left = LRU, right = MRU
 get: update to MRU, put: update to MRU, remove LRU if full
/*   Time: O(1)
    Space: O(capacity)
```

```
*/
class Node{
	int k;
	int val;
	Node*prev;
	Node*curr;

	Node(int key,int value){
		k = key;
		val = value;
		prev = NULL, next = NULL;
	}
};

class LRUCache{
	LRUCache(int capacity){
		cap = capacity;

		left = new Node(0,0);
		right = new Node(0,0);

		left->next = right;
		right->prev = left;
	}

	int get(int key){
		if(cache.find(key)!=cache.end()){
			remove(cache[key]);
			insert(catch[key]);
			return cache[key]->val;
		}
		return -1;
	}

	void put(int key,int value){
		if(cache.find(key)!=cache.end()){
```

```
                remove(cache[key]);

            //Free allocated memory for the removed node
            delete cache[key];
        }
            cache[key] = new Node(key,value);
            insert(cache[key]);

            if(cache.size()>cap){
                // remove from list & delete LRU from map
                Node*lru = left->next;
                remove(lru);
                cache.erase(lru->k);

                delete lru;
            }
        }

private:
        int cap;
        unordered_map<int,Node*>cache;  // {key -> node}
        Node*left,Node*right;

        // remove node from list
        void remove(Node*node){
            Node*prev = node->prev;
            Node*next = node->next;

            prev->next = next;
            next->prev = prev;
        }

        // insert node at right
        void insert(Node*node){
            Node*prev = right->prev;
            Node*next = right;
```

```
                prev->next = node;
                next->prev = node;

                node->prev = prev;
                node->next = next;
        }
};

//Merge K Sorted Lists
-> Given array of k sorted linked-lists, merge all into 1 sorted list
  Ex. lists = [[1,4,5],[1,3,4],[2,6]] -> [1,1,2,3,4,4,5,6]
Hint:- Min heap -> optimize space w/ divide-and-conquer, merge 2 each time
/*  Time: O(n log k)
    Space: O(n) -> O(1)
*/
ListNode*mergeKLists(vector<ListNode*>&lists){
        int n = lists.size();
        if(n==0){
                return NULL;
        }

        while(n>1){
                for(int i=0;i<n/2;i++){
                        lists[i] = mergeLists(lists[i],lists[n-i-1]);
                }
                n = (n+1)/2;
        }
        return lists.front();
}
ListNode*mergeLists(ListNode*l1,ListNode*l2){
        if(l1==NULL && l2==NULL){
                return NULL;
        }

        if(l1==NULL){
```

```
        return l2;
    }
    if(l2==NULL){
        return l1;
    }

    ListNode*head = NULL;
    if(l1->val<=l2->val){
        head = l1;
        l1 = l1->next;
    }
    else{
        head = l2;
        l2 = l2->next;
    }
    ListNode*curr = head;

    while(l1!=NULL && l2!=NULL){
        if(l1->val<=l2->val){
            curr->next = l1;
            l1 = l1->next;
        }
        else{
            curr->next = l2;
            l2 = l2->next;
        }
        curr = curr->next;
    }

    if(l1==NULL)
        curr->next = l2;
    else
        curr->next = l1;

    return head;
}
```

//Reverse Nodes in K Group

->Given head of linked list, reverse nodes of list k at a time

Ex. head = [1,2,3,4,5], k = 2 -> [2,1,4,3,5]

Hint:- Maintain prev, curr, & temp pointers to reverse, count k times

```
/*  Time: O(n)
   Space: O(1)
*/
ListNode*reverseKGroup(ListNode*head,int k){
        ListNode*dummy = new ListNode();
        dummy->next = head;

        ListNode*prev = dummy;
        ListNode*curr = dummy->next;
        ListNode*temp = NULL;

        int count = k;

        while(curr!=NULL){
                if(count>1){
                        temp = prev->next;
                        prev->next = curr->next;
                        curr->next = curr->next->next;
                        prev->next->next = temp;
                        count--;
                }
                else{
                        prev = curr;
                        curr = curr->next;
                        count = k;

                        ListNode*end = curr;
                        for(int i=0;i<k;i++){
                                if(end==NULL){
                                        return dummy->next;
                                }
```

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

/***Trees***/
//Invert Binary Tree
->  Given the root of a binary tree, invert the tree, and return its root
Ex. root = [4,2,7,1,3,6,9] -> [4,7,2,9,6,3,1], [2,1,3] -> [2,3,1]
Hint:- Preorder traversal, at each node, swap it's left and right children
/*  Time: O(n)
   Space: O(n)
*/
TreeNode*invertTree(TreeNode*root){
        if(root==NULL)
                return;

        swap(root->left,root->right);
        invertTree(root->left);
        invertTree(root->right);
        return root;
}

//Maximum Depth of Binary Tree
->Given root of binary tree, return max depth (# nodes along longest path from root to leaf)
Hint:-  At every node, max depth is the max depth between its left & right children + 1
/*  Time: O(n)
   Space: O(n)
*/
int maxDepth(TreeNode*root){
        if(root==NULL)
                return;
```

```
        return 1+max(maxDepth(root->left),maxDepth(root->right));
}
/*another method*/
int maxDepth(TreeNode*root){
        if(root==NULL)
                return;

        queue<TreeNode*>q;
        q.push(root);
        int ans =0;
        while(!q.empty()){
                int count = q.size();
                for(int i=0;i<count;i++){
                        TreeNode*node = q.front();
                        q.pop();
                        if(node->left!=NULL){
                                q.push(node->left);
                        }
                        if(node->right!=NULL){
                                q.push(node->right);
                        }
                }
                ans++;
        }
        return ans;
}

//Diameter of Binary Tree
-> Given root of binary tree, return length of diameter of tree (longest path b/w any 2 nodes)
Hint:- Max path b/w 2 leaf nodes, "1 +" to add path
/*  Time: O(n)
   Space: O(n)
*/
int diameterOfTree(TreeNode*root){
        int ans =0;
```

```
        dfs(root,ans);
        return ans;
}
int dfs(TreeNode*root,int &ans){
        if(root==NULL)
                return 0;

        int left = dfs(root->left,ans);
        int right = dfs(root->right,ans);

        ans = max(ans,left+right);
        return 1+max(left,right);
}
```

//Balanced Binary Tree

Given binary tree, determine if height-balanced (all left & right subtrees height diff <= 1)

Hint:- Check if subtrees are balanced, if so, use their heights to determine further balance

```
/* Time: O(n)
   Space: O(n)
*/
bool isBalanced(TreeNode*root){
        int height =0;
        dfs(root,height);
        return;
}
bool dfs(TreeNode*root,int &height){
        if(root==NULL){
                height =-1;
                return true;
        }

        int left =0,right =0;
        /**edge cases**/
        if(!dfs(root->left,left)|| !dfs(root->right,right)){
                return false;
```

```
        }
        if(abs(left-right)>1){
                return false;
        }

        height = 1+max(left,right);
        return true;
}
```

//Same Tree (same structure & values)

->Given roots of 2 binary trees, check if they're the same or not (same structure & values)

Ex. p = [1,2,3] q = [1,2,3] -> true, p = [1,2] q = [1,null,2] -> false

Hint:- Check: (1) matching nulls, (2) non-matching nulls, (3) non-matching values

```
/*  Time: O(n)
   Space: O(n)
*/
bool isSameTree(TreeNode*p,TreeNode*q){
        if(p==NULL && q==NULL) //Check: matching nulls
                return true;
        if(p==NULL || q==NULL) //Check: non-matching nulls
                return false;

        if(p->val!=q->val) //Check: non-matching values
                return false;

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

//Subtree of Another Tree

-> Given the roots of 2 binary trees, return true if a tree has a subtree of the other tree

Hint:- Check at each node of the root tree if it's the same as the subRoot tree (structure + values)

```
/* Time: O(m x n)
   Space: O(m)
*/
bool isSubTree(TreeNode*root,TreeNode*subRoot){
```

```
        if(root==NULL)
         return false;

        if(isSame(root,subTree))
                return true;

        return isSubTree(root->left,subRoot) || isSubTree(root->right,subRoot);
}
bool isSame(TreeNode*root,TreeNode*subRoot){
        if(root==NULL && subRoot == NULL)
                return true;
        if(root==NULL || subRoot==NULL)
                return false;

        if(root->val != subRoot->val)
                return false;

        return isSame(root->left,subRoot->left) && isSame(root->right,subRoot->right);
}
```

//Lowest Common Ancestor of a Binary Search Tree

-> Given a binary search tree (BST), find the LCA of 2 given nodes in the BST

Hint:- Use BST property: if curr > left & right go left, else if < go right, else done

```
/*  Time: O(n)
   Space: O(n)
*/
TreeNode*lowestCommonAncestor(TreeNode*root,TreeNode*p,TreeNode*q){
        if(p->val<root->val && q->val<root->val){
                return lowestCommonAncestor(root->left,p,q);
        }
        else if(p->val>root->val && q->val>root->val){
                return lowestCommonAncestor(root->right,p,q);
        }
        else{
                return root;
        }
```

```
}

/***another soln***/
TreeNode*lowestCommonAncestor(TreeNode*root,TreeNode*p,TreeNode*q){
        while(root!=NULL){
                if(p->val < root->val && q->val<root->val){
                        root = root->left;
                }
                else if(p->val>root->val && q->val>root->val){
                        root = root->right;
                }
                else{
                        return root;
                }
        }
        return NULL;
}

//Binary Tree Level Order Traversal
->Given root of binary tree, return level order traversal of its nodes (left to right)
Ex. root = [3,9,20,null,null,15,7] -> [[3],[9,20],[15,7]]
Hint:- Standard BFS traversal, at each level, push left & right nodes if they exist to queue
/*   Time: O(n)
    Space: O(n)
*/
vector<vector<int>>levelOrder(TreeNode*root){
        vector<vector<int>>ans;

        if(root==NULL){
                return ans;
        }

        queue<TreeNode*>q;
        q.push(root);

        while(!q.empty()){
```

```
            int count = q.size();
            vector<int>curr;

            for(int i=0;i<count;i++){
                TreeNode*node = q.front();
                q.pop();

                curr.push_back(node->val);

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

        ans.push_back(curr);
}
        return ans;
}
```

//Binary Tree Right Side View

-> Given root of binary tree, return values that can only be seen from the right side(top to bottom).

Hint:- BFS traversal, push right first before left, store only first value

```
/*  Time: O(n)
   Space: O(n)
*/
vector<int>rightSideView(TreeNode*root){
        if(root==NULL)
            return{};

        queue<TreeNode*>q;
        q.push(root);

        vector<int>ans;

        while(!q.empty()){
```

```
                int count = q.size();

                for(int i=count;i>0;i--){
                        TreeNode*node = q.front();
                        q.pop();

                        if(i==count){
                                ans.push_back(node->val);
                        }
                        if(node->right!=NULL){
                                q.push(node->right);
                        }
                        if(node->left!=NULL){
                                q.push(node->left);
                        }
                }
        }
        return ans;
}
```

//Count Good Nodes In Binary Tree

->Given binary tree, node is "good" if path from root has no nodes > X, return # of "good"

->A node X in the tree is named "good" if in the path from root to X there are no nodes with a value greater than X.

->Return the number of good nodes in the binary tree.

Hint:- Maintain greatest value seen so far on a path, if further node >= this max, "good" node

```
/*  Time: O(n)
   Space: O(n)
*/

int goodNodes(TreeNode*root){
        int ans =0;
        dfs(root,root->val,ans);
        return ans;
}
```

```
void dfs(TreeNode*root,int maxSoFar,int &ans){
        if(root==NULL)
                return{};

        if(root->val>=maxSoFar){
                ans++;
        }
        dfs(root->left,max(maxSoFar,root->val),ans);
        dfs(root->right,max(maxSoFar,root->val),ans);
}
```

//Validate Binary Search Tree

->Given root of binary tree, determine if it's valid (left all < curr, right all > curr)

Hint:- Inorder traversal & check if prev >= curr, use recursive/iterative solutions

```
/*  Time: O(n)
   Space: O(n)
*/
```

//DFS soln.

```
bool isValidBST(TreeNode*root){
        return help(root,LONG_MIN,LONG_MAX);
}
bool help(TreeNode*root,long left,long right){
        if(!root){
                return true;
        }

        if(root->val<right && root->val>left){
                return help(root->left,left,root->val) && help(root->right,right,root->val);
        }
        return false;
}
```

//Inorder Traversal soln.

```
bool isValidBST(TreeNode*root){
        TreeNode*prev = NULL;
        return inorder(root,prev);
}
```

```
bool inorder(TreeNode*root,TreeNode*prev){
        if(root==NULL){
                return true;
        }

        if(!inorder(root->left,prev)){
                return false;
        }
        if(prev!= NULL && prev->val>= root->val){
                return false;
        }

        prev = root;

        if(!inorder(root->right,prev)){
                return false;
        }
        return true;
}

//Using Stack soln.
bool isValidBST(TreeNode*root){
        stack<TreeNode*>st;
        TreeNode*prev = NULL;

        while(!st.empty()|| root!=NULL){
                while(root!=NULL){
                        st.push(root);
                        root = root->left;
                }
                root = st.top();
                st.pop();

                if(prev!=NULL && prev->val>=root->val){
                        return false;
                }
```

```
                prev = root;
                root = root->right;
        }
        return true;
}
```

//Kth Smallest Element In a BST

->Given root of BST & int k, return kth smallest value (1-indexed) of all values in tree

Ex. root = [3,1,4,null,2] k = 1 -> 1, root = [5,3,6,2,4,null,null,1] k = 3 -> 3

Hint:- Inorder traversal, each visit decrement k, when k = 0 return, works because inorder

```
/*  Time: O(n)
   Space: O(n)
*/
int kthSmallest(TreeNode*root,int k){
        int ans =0;
        inorder(root,k,ans);
        return ans;
}
void inorder(TreeNode*root,int &k,int &ans){
        if(root==NULL)
                return;

 inorder(root->left,k,ans);
        k--;
 if(k==0){
        ans = root->val;
        return;
}
  inorder(root->right,k,ans);
}
```

//Construct Binary Tree From Preorder And Inorder Traversal

-> Given 2 integer arrays preorder & inorder, construct & return the binary tree

Ex. preorder = [3,9,20,15,7], inorder = [9,3,15,20,7] -> [3,9,20,null,null,15,7]

Hint:- Preorder dictates "nodes", inorder dictates "subtrees" (preorder values, inorder positions)

```
/*
   Time: O(n)
   Space: O(n)
*/
TreeNode*buildTree(vector<int>&preorder,vector<int>&inorder){
        int indx =0;
        return build(preorder,inorder,index,0,inorder.size()-1);
}

TreeNode*build(vector<int>&preorder,vector<int>&inorder,int &index,int i,int j){
        if(i>j){
                return NULL;
        }

        TreeNode*root = new TreeNode(preorder[index]);
        int split =0;
        for(int i=0;i<inorder.size();i++){
                if(preorder[index]==indorder[i]){
                        split = i;
                        break;
                }
        }
        index++;

        root->left = build(preorder,inorder,index,i,split-1);
        root->right = build(preorder,inorder,index,split+1,j);
        return root;
}
```

//Binary Tree Maximum Path Sum

->Given root of binary tree, return max path sum (seq. of adj node values added together)

Hint:- Path can only have <= 1 split point, assume curPath has it, so return can't split again

```
/*  Time: O(n)
   Space: O(n)
```

```
*/
int maxPathSum(TreeNode*root){
        int maxPath = INT_MIN;
        dfs(root,maxPath);
        return maxPath;
}
int dfs(TreeNode*root,int &maxPath){
        if(root==NULL){
                return 0;
        }

        int left = max(dfs(root->left,maxPath),0);
        int right = max(dfs(root>right,maxPath),0);

        int currPath = root->val+left+right;
        maxPath = max(maxPath,currPath);

        return root->val + max(left,right);
}

//Serialize And Deserialize Binary Tree (IMP)
->Design an algorithm to serialize & deserialize a binary tree
Hint:- Use "stringstream" to concisely handle negatives, nulls, etc.
/* Time: O(n) serialize, O(n) deserialize
   Space: O(n) serialize, O(n) deserialize
*/
class codec{
        //Encodes a tree to a single string
        string serialize(TreeNode*root){
                ostringstream out;
                encode(root,out);
                return out.str();
        }
        //Decodes your encoded data to tree
        TreeNode*deserialize(string data){
                istringstream in(data);
```

```
        return decode(in);
    }

    void encode(TreeNode*root,ostringstream &out){
        if(root==NULL){
            out<<"N";
            return;
        }

        out<<root->val<<" ";

        encode(root->left,out);
        encode(root->right,out);
    }

    TreeNode*decode(istringstream &in){
        string value = "";
        in>>value;

        if(value == "N"){
            return NULL;
        }

        TreeNode*root = new TreeNode(stoi(value));
        root->left = decode(in);
        root->right = decode(in);

        return root;
    }
};

/***Graphs Problem**/
//Number of Islands
->Given grid where '1' is land & '0' is water, return no. of islands
Hint:-  DFS, set visited land to '0' to not visit it again, count islands
/*  Time: O(m x n)
```

```
   Space: O(m x n)
*/
int numIslands(vector<vector<int>>&grid){
        int n = grid.size();
        int m = grid[0].size();

        int ans =0;
        for(int i=0;i<m;i++){
                for(int j=0;j<n;j++){
                        if(grid[i][j]=='1'){
                                dfs(grid,i,j,n,m);
                                ans++;
                        }
                }
        }

        return ans;
}
void dfs(vector<vector<int>>&grid,int i,int j,int n,int m){
        if(i<0 ||i>=n ||j<0 ||j>=m ||grid[i][j]=='0'){
                return;
        }
        grid[i][j] = 0;

        dfs(grid,i,j-1,n,m);
        dfs(grid,i,j+1,n,m);
        dfs(grid,i+1,j,n,m);
        dfs(grid,i-1,j,n,m);
}

//Clone Graph
->Given ref of a node in connected undirected graph, return deep copy
Hint:- Both BFS & DFS work, map original node to its copy
/* Time: O(m + n)
   Space: O(n)
*/
```

```
unordered_map<Node*,Node*>mp;
Node* cloneGraph(Node*node){
        if(node==NULL)
                return NULL;

        Node*copy = new node(node->val);
        mp[node] = copy;

        queue<Node*>q;
        q.push(node);

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

                for(int i=0;i<curr->neighbors.size();i++){
                        Node*neighbor = curr->neighbors[i];

                        if(mp.find(neighbor)==mp.end()){
                                mp[neighbor] = new Node(neighbor->val);
                                q.push(neighbor);
                        }
                        mp[curr]->neighbours.push_back(mp[neighbour]);
                }
        }
        return copy;
}

//Max Area of Island
-> Given grid where '1' is land & '0' is water, return largest island
Hint:- DFS, set visited land to '0' to not visit it again, store biggest
/*  Time: O(m x n)
   Space: O(m x n)
*/
int maxAreaIsland(vector<vector<int>>&grid){
        int n = grid.size();
```

```
        int m = grid[0].size();

        int ans =0;
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        if(grid[i][j]==1){
                                ans = max(ans,dfs(grid,i,j,n,m));
                        }
                }
        }
        return ans;
}
void dfs(vector<vector<int>&grid,int i,int j,int n,int m){
        if(i<0||i>=n|| j<0||j>=m || grid[i][j]==0){
                return;
        }
        grid[i][j] =0;

        return 1+grid(i-1,j,n,m)+grid(i+1,j,n,m)+grid(i,j-1,n,m)+grid(i,j+1,n,m);
}

//Surrounded Regions
->Given a matrix, capture ('X') all regions that are surrounded ('O')
Hint:- Distinguish captured vs escaped, 'X' vs 'O' vs 'E'
/*  Time: O(m x n)
   Space: O(m x n)
*/
void Surrounded(vector<vector<char>>&board){
        int n = board.size();
        int m = board[0].size();

        // marking escaped cells along the border
        for(int i=0;i<n;i++){
                dfs(board,i,0,n,m);
                dfs(board,i,m-1,n,m);
        }
```

```
        for(int j=0;j<m;j++){
                dfs(board,0,j,n,m);
                dfs(board,n-1,j,n,m);
        }

        // flip cells to correct final states
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        if(board[i][j]=='O'){
                                board[i][j] = 'X';
                        }
                        if(board[i][j]=='E'){
                                board[i][j] = 'O';
                        }
                }
        }
}
void dfs(vector<vector<char>>&board,int i,int j,int n,int m){
        if(i<0||i>=n|| j<0||j>=m ||board[i][j]!='O'){
                return;
        }
        board[i][j] = 'E';

        dfs(board,i-1,j,n,m);
        dfs(board,i+1,j,n,m);
        dfs(board,i,j-1,n,m);
        dfs(board,i,j+1,n,m);
}
```

//Rotting Oranges

->Given grid: 0 empty cell, 1 fresh orange, 2 rotten orange

->Return min no. of minutes until no cell has a fresh orange

Hint:- Use BFS: rotten will contaminate neighbors first, then propagate out

```
/*  Time: O(m x n)
   Space: O(m x n)
```

```
*/
vector<vector<int>>dirs = {{-1,0},{1,0},{0,-1},{0,1}};
int orangesRotting(vector<vector<int>>&grid){
        int m = grid.size();
        int n = grid[0].size();

        // build initial set of rotten oranges
        queue<pair<int,int>q;
        int fresh =0;
        for(int i=0;i<m;i++){
                for(int j=0;j<n;j++){
                        if(grid[i][j]==2){
                                q.push({i,j});
                        }
                        else if(grid[i][j]==1){
                                fresh++;
                        }
                }
        }
        //mark the start of a minute
        q.push({-1,-1});
        int ans = -1;

        // start rotting process via BFS
        while(!q.empty()){
                int row = q.front().first;
                int col = q.front().second;
                q.pop();

                if(row==-1){
                        // finish 1 minute of processing, mark next minute
                        ans++;
                        if(!q.empty()){
                                q.push({-1,-1});
                        }
                }
```

```
                else{
                        // rotten orange, contaminate its neighbors
                        for(int i=0;i<dirs.size();i++){
                                int x = row+dirs[i][0];
                                int y = col+dirs[i][1];

                                if(x<0||x>=m|| y<0||y>=n){
                                        continue;
                                }
                                if(grid[x][y]==1){
                                         //contaminate
                                        grid[x][y] = 2;
                                        fresh--;
                                    //this orange will now contaminate others
                                        q.push({x,y});
                                }
                        }
                }
        }
        if(fresh==0){
                return ans;
        }
        return -1;
}

//Walls And Gates
->Given grid: -1 wall, 0 gate, INF empty, fill each empty w/ dist to nearest gate
Hint:- BFS traversal, shortest path from each gate to all empty rooms
Each gate only looks at within 1 space, then next gate, guarantees shortest
/*  Time: O(m x n)
   Space: O(m x n)
*/
vector<vector<int>>dirs = {{-1,0},{1,0},{0,-1},{0,1}};
void wallAndGates(vector<vector<int>>&rooms){
        int n = rooms.size();
        int m = rooms[0].size();
```

```
        queue<pair<int,int>q;
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        if(rooms[i][j]==0){
                                q.push({i,j});
                        }
                }
        }

        while(!q.empty()){
                int row = q.front().first;
                int col = q.front().second;
                q.pop();

                for(int i=0;i<4;i++){
                        int x = row+dirs[i][0];
                        int y = col+dirs[i][1];

                        if(x<0||x>=n|| y<0||y>=m||rooms[x][y]!=INT_MAX){
                                continue;
                        }

                        rooms[x][y] = rooms[row][col]+1;
                        q.push({x,y});
                }
        }
}
```

//Redundant Connection

->Given undirected graph, return an edge that can be removed to make a tree

Ex. edges = [[1,2],[1,3],[2,3]] -> [2,3]

Hint:- If n nodes & n edges, guaranteed a cycle

How to know creating cycle? When connecting a node already connected

"Union Find": can find this redundant edge, track parents & ranks

```
/*  Time: O(n)
```

```
   Space: O(n)
*/
vector<int>findRedundantConnection(vector<vector<int>>&edges){
        int n = edges.size();

        vector<int>parent,ranks;
        for(int i=0;i<n;i++){
                parents.push_back(i);
                ranks.push_back(1);
        }

        vector<int>ans;
        for(int i=0;i<n;i++){
                int n1 = edges[i][0];
                int n2 = edges[i][1];
                if(!doUnion(parents,ranks,n1,n2)){
                        ans = {n1,n2};
                        break;
                }
        }
        return ans;
}
int doFind(vector<int>&parents,int n){
        int p = parents[n];
        while(p!=parents[n]){
                parents[p] = parents[parents[p]];
                p = parents[p];
        }
        return p;
}
bool doUnion(vector<int>&parents,vector<int>&ranks,int n1,int n2){
        int p1 = doFind(parents,n1);
        int p2 = doFind(parents,n2);
        if(p1==p2){
                return false;
        }
```

```
        if(ranks[p1]>ranks[p2]){
                parents[p2] = p1;
                ranks[p1]+= ranks[p2];
        }
        else{
                parents[p1] = p2;
                ranks[p2]+= ranks[p1];
        }
        return true;
}

//Course Schedule
->Courses & prerequisites, return true if can finish all courses
Ex. numCourses = 2, prerequisites = [[1,0]] -> true
Hint:-  All courses can be completed if there's no cycle (visit already visited)
/*  Time: O(V + E)
   Space: O(V + E)
*/
bool canFinish(int numCourses,vector<vector<int>>&preq){
        // map each course to prereq list
        unordered_map<int,vector<int>mp;
        for(int i=0;i<pre.size();i++){
                mp[pre[i][0]].push_back(pre[i][1]);
        }
        // all courses along current DFS path
        unordered_set<int>visited;
        for(int course=0;course<numCourses;course++){
                if(!dfs(course,mp,visited)){
                        return false;
                }
        }
        return true;
}
bool dfs(int course,unordered_map<int,vector<int>>&mp,unordered_set<int>&visited){
        if(visited.find(course)!= visited.end()){
                return false;
```

```
        }
        if(mp[course].empty()){
                return true;
        }
        visited.insert(course);
        for(int i=0;i<mp[courses].size();i++){
                int nextCourse = mp[course][i];
                if(!dfs(nextCourse,mp,visited)){
                        return false;
                }
        }
        mp[course].clear();
        visited.erase(course);
        return true;
}

//Course Schedule II
-> Courses & prerequisites, return ordering of courses to take to finish all courses
 Ex. numCourses = 2, prerequisites = [[1,0]] -> [0,1], take course 0 then 1
Hint:- All courses can be completed if there's no cycle, check for cycles
/* Time: O(V + E)
   Space: O(V + E)
*/
vector<int>findOrder(int numCourses,vector<vector<int>>&preq){
        unordered_map<int,vector<int>mp;
        // build adjacency list of prereqs
        for(int i=0;i,pre.size();i++){
                mp[pre[i][0]].push_back(pre[i][1]);
        }

        unordered_set<int>visit;
        unordered_set<int>cycle;

        vector<int>ans;
        for(int course =0; course<numCourses;course++){
                if(!dfs(course,mp,visit,cycle,ans)){
```

```
                return false;
            }
        }
        return ans;
}
/* a course has 3 possible states:-
   visited -> course added to result
   visiting -> course not added to result, but added to cycle
   unvisited -> course not added to result or cycle
*/
bool dfs(int
course,unordered_map<int,vector<int>>&mp,unordered_set<int>&visit,unordered_set<int>&c
ycle,vector<int>&ans){
        if(cycle.find(course)!=cycle.end()){
            return false;
        }

        if(visit.find(course)!= visit.end()){
            return true;
        }
        cycle.insert(course);
        for(int i=0;i<mp[course].size();i++){
            int nextCourse = mp[course][i];
            if(!dfs(nextCourse,mp,visit,cylce,ans)){
                return false;
            }
        }
        cycle.erase(course);
        visit.insert(course);
        ans.push_back(course);
        return ans;
}

//Number of Connected Components In An Undirected Graph
->Graph of n nodes, given edges array, return # of connected components
 Ex. n = 5, edges = [[0,1],[1,2],[3,4]] -> 2
```

Hint:- "Union find", for each edge combine, if already in same set keep traversing,
If not in same set, decrement count by 1, count will store no. of components.

```
/*  Time: O(n)
   Space: O(n)
*/
int countComponents(int n,vector<vector<int>>edges){
        vector<int>parents;
        vector<int>ranks;

        for(int i=0;i<n;i++){
                parents.push_back(i);
                ranks.push_back(1);
        }
        int ans = n;
        for(int i=0;i<edges.size();i++){
                int n1 = edges[i][0];
                int n2 = edges[i][1];
                ans -= doUnion(parents,ranks,n1,n2);
        }
        return ans;
}

int doFind(vector<int>&parents,int n){
        int p = parents[n];
        while(p!=parents[n]){
                parents[p] = parents[parents[p]];
                p = parents[p];
        }
        return p;
}
int doUnion(vector<int>&parents,vector<int>&ranks,int n1,n2){
        int n1 = doFind(parents,p1);
        int n2 = doFind(parents,p2);
        if(p1==p2){
                return 0;
        }
```

```
        if(ranks[p1]>ranks[p2]){
                parents[p2] = p1;
                ranks[p1] += ranks[p2;]
        }
        else{
                parents[p1] = p2;
                ranks[p2]+=ranks[p1];
        }
        return 1;
}

//Graph of nodes, list of edges, determine if edges make valid tree
 Ex. n = 5, edges = [[0,1],[0,2],[0,3],[1,4]] -> true
 Hint:-
(1) For graph to be a valid tree, must have exactly n - 1 edges
(2) If graph has "fully connected & n - 1 edges", can't contain cycle
/*
   Time: O(n)
   Space: O(n)
*/
bool validTree(int n,vector<vector<int>>&edges){
        vector<vector<int>>adj(n);
        for(int i=0;i<edges.size();i++){
                vector<int>edge = edges[i];
                adj[edge[0].push_back(edge[1]);
                adj[edge[1]].push_back(edge[0]);
        }

        vector<bool>visited(n);
        if(){
                return false;
        }

        for(int i=0;i<visited.size();i++){
                if(!visited[i]){
```

```
                return false;
            }
        }
        return true;
}

bool hasCycle(vector<vector<int>>&adj,vector<bool>&visited,int parent,int child)[
        if(visited[child]){
            return true;
        }

        visited[child] = true;
        /*checking for cycles and connectedness*/
        for(int i=0;i<adj[child].size();i++){
            int curr = adj[child][i];
            if(curr != parent && hasCycle(adj,visited,child,curr)){
                return true;
            }
        }
        return false;
}
```

//Word Ladder (V.IMP)

->A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

- Every adjacent pair of words differs by a single letter.
- Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- sk == endWord

->return the number of words in the shortest transformation sequence from beginWord to endWord, or 0.

Hint:- Use "BFS", change 1 letter at a time (neighbors), if in dict add to queue, else skip

```
/*  Time: O(m^2 x n) -> m = length of each word, n = # of words in input word list
   Space: O(m^2 x n)
*/
int ladderLength(string beginword,string endWord,vector<string>&wordList){
```

```
unordered_set<string>dict;
for(int i=0;i<wordList.size();i++){
        dict.insert(wordList[i]);
}
queue<string>q;
q.push(beginword);

int ans = 1;

while(!q.empty()){
        int count = q.size();
        for(int i=0;i<count;i++){
                string word = q.front();
                q.pop();

                if(word==endWord){
                        return ans;
                }
                dict.erase(word);

                for(int j=0;j<word.size();j++){
                        char c = word[j];
                        for(int k= 0;k<26;k++){
                                word[j] = k+'a';
                                if(dict.find(word)!= dict.end()){
                                        q.push(word);
                                        dict.erase(word);
                                }
                                word[j] = c;
                        }
                }
        }
        ans++;
}
        return 0;
}
```

//Min Cost to Connect All Points

->Given array of points, return min cost to connect all points, All points have 1 path b/w them, cost is "Manhattan distance"

Hint:- MST problem, Prim's, greedily pick node not in MST & has smallest edge cost Add to MST, & for all its neighbors, try to update min dist values, repeat

```
/*  Time: O(n^2)
   Space: O(n)
*/
int minCostConnectPoints(vector<vector<int>>& points) {
        int n = points.size();

        int edgesUsed = 0;
        // track visited nodes
        vector<bool> inMST(n);
        vector<int> minDist(n, INT_MAX);
        minDist[0] = 0;

        int result = 0;

        while (edgesUsed < n) {
            int currMinEdge = INT_MAX;
            int currNode = -1;

            // greedily pick lowest cost node not in MST
            for (int i = 0; i < n; i++) {
                if (!inMST[i] && currMinEdge > minDist[i]) {
                    currMinEdge = minDist[i];
                    currNode = i;
                }
            }

            result += currMinEdge;
            edgesUsed++;
            inMST[currNode] = true;
```

```
            // update adj nodes of curr node
            for (int i = 0; i < n; i++) {
                int cost = abs(points[currNode][0] - points[i][0])
                    + abs(points[currNode][1] - points[i][1]);

                if (!inMST[i] && minDist[i] > cost) {
                    minDist[i] = cost;
                }
            }
        }

        return result;
    }
```

//Network Delay Time

->Signal sent from node k to network of n nodes, return time for all nodes to receive it

Ex. times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2 -> 2

u,v,w -> u = source node, v = target node, w = signal travel time

Hint:- Shortest path from node k to every other node, "Dijkstra's to find fastest path"

```
/*
    Time: O(V + E log V)
    Space: O(V + E)
*/
int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<pair<int, int>> adj[n + 1];
        for (int i = 0; i < times.size(); i++) {
            int source = times[i][0];
            int dest = times[i][1];
            int time = times[i][2];
            adj[source].push_back({time, dest});
        }

        vector<int> signalReceiveTime(n + 1, INT_MAX);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        pq.push({0, k});
```

```
    // time for start node is 0
    signalReceiveTime[k] = 0;

    while (!pq.empty()) {
        int currNodeTime = pq.top().first;
        int currNode = pq.top().second;
        pq.pop();

        if (currNodeTime > signalReceiveTime[currNode]) {
            continue;
        }

        // send signal to adjacent nodes
        for (int i = 0; i < adj[currNode].size(); i++) {
            pair<int, int> edge = adj[currNode][i];
            int time = edge.first;
            int neighborNode = edge.second;

            // fastest signal time for neighborNode so far
            if (signalReceiveTime[neighborNode] > currNodeTime + time) {
                signalReceiveTime[neighborNode] = currNodeTime + time;
                pq.push({signalReceiveTime[neighborNode], neighborNode});
            }
        }
    }

    int result = INT_MIN;
    for (int i = 1; i <= n; i++) {
        result = max(result, signalReceiveTime[i]);
    }

    if (result == INT_MAX) {
        return -1;
    }
    return result;
}
```

//Swim In Rising Water

->Given an integer elevation matrix, rain falls, at time t, depth everywhere is t

Can swim iff elevation at most t, return least time get from top left to bottom right

Hint:- Shortest path w/ min heap: at every step, find lowest water level to move forward

```
/*  Time: O(n^2 log n)
    Space: O(n^2)
*/
vector<vector<int>> dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
  int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size();
        if (n == 1) {
            return 0;
        }

        vector<vector<bool>> visited(n, vector<bool>(n));
        visited[0][0] = true;

        int result = max(grid[0][0], grid[n - 1][n - 1]);

        priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
        pq.push({result, 0, 0});

        while (!pq.empty()) {
            vector<int> curr = pq.top();
            pq.pop();

            result = max(result, curr[0]);

            for (int i = 0; i < 4; i++) {
                int x = curr[1] + dirs[i][0];
                int y = curr[2] + dirs[i][1];

                if (x < 0 || x >= n || y < 0 || y >= n || visited[x][y]) {
                    continue;
                }
```

```
            if (x == n - 1 && y == n - 1) {
                return result;
            }

            pq.push({grid[x][y], x, y});
            visited[x][y] = true;
        }
    }

    return -1;
}

//Cheapest Flights Within K Stops
->Bellman-Ford algorithm to find the cheapest price from source (src) to destination (dst)with at most k stops allowed.
->final result is the minimum cost to reach the destination, or -1 if the destination is not reachable within the given constraints.
/*  Space Complexity: O(n) - space used for the prices array.
   Time Complexity: O(k * |flights|) - k iterations, processing all flights in each iteration.
*/
int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int k) {
    vector<int> prices(n, INT_MAX);
    prices[src] = 0;

    // Perform k+1 iterations of Bellman-Ford algorithm.
    for (int i = 0; i < k + 1; i++) {
        vector<int> tmpPrices(begin(prices), end(prices));

        for (auto it : flights) {
            int s = it[0];
            int d = it[1];
            int p = it[2];

            if (prices[s] == INT_MAX) continue;
```

```
            if (prices[s] + p < tmpPrices[d]) {
                tmpPrices[d] = prices[s] + p;
            }
          }
          prices = tmpPrices;
       }
       return prices[dst] == INT_MAX ? -1 : prices[dst];
    }
```

//Reconstruct Itinerary

-> Given airline tickets, find valid itinerary (use all tickets once)

Ex. tickets = [["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]

output = ["JFK","MUC","LHR","SFO","SJC"]

Hint:- Greedy DFS, build route backwards when retreating, merge cycles into main path

```
/*  Time: O(E log (E / V)) -> E = # of flights, V = # of airports, sorting
    Space: O(V + E) -> store # of airports & # of flights in hash map
*/

vector>string>findIterinary(vector<vector<string>>&tickets){
        unordered_map<string,multiset<string>>mp;
        for(int i=0;i<tickets.size();i++){
                mp[tickets[i][0]].insert(tickets[i][1]);
        }

        vector<string>ans;
        dfs(mp,"JFK",ans);
        reverse(ans.begin(),ans.end());
        return ans;
}
void dfs(unordered_map<string,multiset<string>>&mp,string airport,vector<string>&ans){
        while(!mp[airport].empty()){
                string next = *mp[airport].begin();
                mp[airport].erase(mp[airport].begin());
                dfs(mp,next,ans);
        }
        ans.push_back(airport);
```

```
}

/*** 1-D Dynamic Programming***/
//Climbing Stairs
->Climbing stairs, either 1 or 2 steps, distinct ways to reach top
Ex. n = 2 -> 2 (1 + 1, 2), n = 3 -> 3 (1 + 1 + 1, 1 + 2, 2 + 1)
Hint:- Recursion w/ memoization -> DP, why DP? Optimal substructure
Recurrence relation: dp[i] = dp[i - 1] + dp[i - 2]
Reach ith step in 2 ways: 1) 1 step from i-1, 2) 2 steps from i-2.

/*  Time: O(n)
   Space: O(1)
*/
int climbStairs(int n){
        if(n==1){
                return 1;
        }
        if(n==2){
                return 2;
        }

        int first = 1,second = 2;
        int ans= 0;
        for(int i=2;i<n;i++){
                ans = first+second;
                first = second;
                second = ans;
        }
        return ans;
}

//Min Cost Climbing Stairs
-> Given cost array, ith step is cost[i], can climb 1 or 2 steps
-> Return min cost to reach top floor, can start at index 0 or 1
 Ex. cost = [10,15,20] -> 15, start at idx 1, pay 15, climb 2
```

Hint:- Recursion w/ memoization -> DP, min cost to reach 1/2 steps below curr step

Recurrence relation: minCost[i] = min(minCost[i-1] + cost[i-1], minCost[i-2] + cost[i-2])

```
/*  Time: O(n)
    Space: O(1)
*/
int minCostClimbingStairs(vector<int>&cost){
        int n = cost.size();
        int downOne = 0, donwnTwo =0;

        for(int i=2;i<=cost.size();i++){
                int temp = downOne;
                downOne = min(downOne+cost[i-1],downTwo+cost[i-2]);
                downTwo = temp;
        }
        return downOne;
}
```

//House Robber

-> Given int array, return max amount can rob (can't rob adjacent houses)

Ex. nums = [1,2,3,1] -> 4, rob house 1 then house 3: 1 + 3 = 4

Hint:-  Recursion w/ memoization -> DP, rob either 2 away + here, or 1 away

Recurrence relation: robFrom[i] = max(robFrom[i-2] + nums[i], robFrom[i-1])

```
/* Time: O(n)
    Space: O(1)
*/
int rob(vector<int>&nums){
        int prev =0;
        int curr = 0;
        int next =0;

        for(int i=0;i,nums.size();i++){
                next = max(prev+nums[i],curr);
                prev = curr;
                curr = next;
        }
        return curr;
```

```
}

//House Robber II
-> Given int array in a circle, return max amount can rob (can't rob adj houses)
 Ex. nums = [2,3,2] -> 3, can't rob house 1 & 3 b/c circular adj, so rob 2
Hint:-  Recursion w/ memo -> DP, rob either 2 away + here, or 1 away, try both ranges
 Recurrence relation: robFrom[i] = max(robFrom[i-2] + nums[i], robFrom[i-1])

/*  Time: O(n)
   Space: O(1)
*/
int rob(vector<int>&nums){
        int n = nums.size();

        if(n==1)
                return nums[0];

        int range1 = robber(nums,0,n-2);
        int range2 = robber(nums,0,n-1);

        return max(range1,range2);
}
int robber(vector<int>&nums,int start,int end){
        int prev =0;
        int curr =0;
        int next =0;

        for(int i=start;i<=end;i++){
                next = max(prev+nums[i],curr);
                prev = curr;
                curr = next;
        }
        return curr;
}

//Longest Palindromic Substring
```

-> Given a string s, return the longest palindromic substring in s

Ex. s = "babad" -> "bab", s = "cbbd" -> "bb"

Hint:- Expand around center, extend as far as possible, store max length

```
/* Time: O(n^2)
   Space: O(1)
*/
string longestPalindrome(string s){
        int maxStart = 0;
        int maxLength = 1;

        for(int i=0;i<s.size()-1;i++){
                middleOut(s,i,i,maxStart,maxLength);
                middleOut(s,i,i+1,maxStart,maxLength);
        }
        return s.substr(maxStart,maxLength);
}
void middleOut(string s,int i,int j, int &maxStart, int &maxLength){
        while(i>=0 && j<=s.size()-1 && s[i]==s[j]){
                i--;
                j++;
        }
        if(j-i+1>maxLength){
                maxStart = i+1;
                maxLength = j-i+1;
        }
}
```

//Palindromic Substrings

->Given a string, return # of palindromic substrings in it.

Ex. s = "babad" -> "bab", s = "cbbd" -> "bb"

Hint:- 2 pointers, middle out, check both odd & even sized strings.

```
/*  Time: O(n^2)
   Space: O(1)
*/
int countSubstrings(string s){
        int ans = 0;
```

```
        for(int i=0;i<s.size();i++){
                middleOut(s,i,i,ans);
                middleOut(s,i,i+1,ans);
        }
        return ans;
}
void middleOut(string s,int i,int j,int &ans){
        while(i>=0 && j<s.size() && s[i]==s[j]){
                ans++;
                i--,j++;
        }
}
```

//Decode Ways

->Given a string with only digits, return no. of ways to decode it (letter -> digit)

Ex. s = "12" -> 2 (AB 1 2 or L 12), s = "226" -> 3 (2 26 or 22 6 or 2 2 6)

Hint:- DP: At each digit, check validity of ones & tens, if valid add to no. of ways

Recurrence relation: dp[i] += dp[i-1] (if valid) + dp[i-2] (if valid)

```
/*  Time: O(n)
   Space: O(n)
*/
int numDecodings(string s){
        if(s[0]=='0'){
                return 0;
        }

        int n = s.size();

        vector<int>dp(n+1);
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2;i<=n;i++){
                int ones = stoi(s.substr(i-1,1));
                if(ones>= 1 && ones<=9){
                        dp[i] += dp[i-1];
```

```
                }
                int tens = stoi(s.substr(i-2,2));
                if(tens>=10 && tens<=26){
                        dp[i] += dp[i-2];
                }
        }
        return dp[n];
}

//Coin Change
->Given array of coins & an amount, return fewest coins to make that amount
Ex. coins = [1,2,5], amount = 11 -> 3, $11 = $5 + $5 + $1

/*  Time: O(m x n) -> m = # of coins, n = amount
    Space: O(n)
*/

Hint:- Compute all min counts for amounts up to i, "simulate" use of a coin
/* Time: O(m x n) -> m = # of coins, n = amount
   Space: O(n)
*/

int coinChange(vector<int>&coins,int amount){
        vector<int>dp(amount+1,amount+1);
        dp[0] = 0;

        for(int i=1;i<amount+1;i++){
                for(int j=0;j<coins.size();i++){
                        if(i-coins[j]>=0){
                                dp[i] = min(dp[i],1+dp[i-coins[j]]);
                        }
                }
        }
        if(dp[amount]==amount+1){
                return -1;
        }
```

```
        return dp[amount];
}

//Maximum Product Subarray
int maxProduct(vector<int>&nums){
        int ans = nums[0];
        int currMin = 1;
        int currMax = 1;

        for(int i=0;i<nums.size();i++){
                int n = nums[i];
                int temp = currMax*n;
                currMax = max(max(n*currMax,n*currMin),n);
                currMin = min(min(temp,n*currMin),n);
                ans = max(ans,currMax);
        }
        return ans;
}

//Word Break
->Given a string & dictionary, return true if: Can segment string into 1 or more dictionary
words
Hint: Use DP, at each loop, substring, check if in dict, & store
/*
   Time: O(n^3)
   Space: O(n)
*/
bool wordBreak(string s,vector<string>&wordDict){
        unordered_set<string>st;

        for(int i=0;i<wordDict.size();i++){
                words.insert(wordDict[i]);
        }

        int n = s.size();
        vector<bool>dp(n+1);
```

```
        dp[0] = true;

        for(int i=1;i<=n;i++){
                for(int j=i-1;j>=0;j--){
                        if(dp[j]){
                                string word = s.substr(j,i-j);
                                if(words.find(word)!= words.end()){
                                        dp[i] = true;
                                        break;
                                }
                        }
                }
        }
        return dp[n];
}

//Longest Increasing Subsequence
-> Given int array, return length of longest increasing subsequenc
 Ex. nums = [10,9,2,5,3,7,101,18] -> 4, [2,3,7,101]
 Framework to solve DP:
    1) Need some function or array that represents ans to the problem (dp array)
    2) Way to transition b/w states (recurrence relation), depends on question
    3) Need a base case (initial solution for every subproblem)
Hint:- Recurrence relation: dp[i] = max(dp[j] + 1)
      Base case: dp[i] = 1, since every element on its own has an LIS of 1
/*  Time: O(n^2)
   Space: O(n)
*/

int lengthOfLIS(vector<int>&nums){
        int n = nums.size();
        vector<int>dp(n,1);

        int ans = 1;
        for(int i=1;i<n;i++){
                for(int j=0;j<i;j++){
```

```
                        if(nums[j]<nums[i]){
                                dp[i] = max(dp[i],dp[j]+1);
                        }
                }
                ans = max(ans,dp[i]);
        }
        return ans;
}
```

//Partition Equal Subset Sum

->Given non-empty, non-negative integer array nums, find if: Can be partitionined into 2 subsets such that sums are equal

Ex. nums = [1,5,11,5] -> true, [1,5,5] & [11], both add to 11

Hint:- Maintain DP set, for each num, check if num in set + curr = target

If not, add curr to every num in set we checked & iterate

```
/* Time: O(n x sum(nums))
   Space: O(sum(nums))
*/
bool canPartition(vector<int>&nums){
        int result =0;
        for(int i=0;i<nums.size();i++){
                result += nums[i];
        }
        if(result %2 !=0)
                return false;

        result /=2;

        unordered_set<int>dp;
        dp.insert(0);

        for(int i=0;i<nums.size();i++){
                unordered_set<int>st;
                for(auto it = dp.begin();it!=dp.end();it++){
                        if(*it+nums[i]==result){
```

```
                        return true;
                    }

                    st.insert(*it+nums[i]);
                    st.insert(*it);
                }
                dp = st;
            }
            return false;
}

//Unique Paths
->Given grid, return no. of unique paths from top-left to bottom-right
Ex. m = 3, n = 2 -> 3 unique paths (R->D->D, D->D->R, D->R->D)
Hint:- Use DP: edges have 1 unique path, inner cells consider where it comes from
  Recurrence relation: grid[i][j] = grid[i-1][j] + grid[i][j-1]
/*  Time: O(m x n)
    Space: O(m x n)
*/

int uniquePaths(int m,int n){
        vector<vector<int>>grid(m,vector<int>(n,0));

        for(int i=0;i<m;i++){
                grid[i][0] = 1;
        }
        for(int j=0;j<n;j++){
                grid[0][j] = 1;
        }

        for(int i=1;i<m;i++){
                for(int j=1;j<n;j++){
                        grid[i][j] = grid[i-1][j] + grid[i][j-1];
                }
        }
        return grid[m-1][j-1];
```

```
}

//Longest Common Subsequence
-> Given 2 strings, return length of longest common subsequence
   Ex. text1 = "abcde", text2 = "ace" -> 3, "ace" is LCS
Hint:- build DP bottom-up
/*  Time: O(m x n)
   Space: O(m x n)
*/

int longestCommonSubsequence(string text1, string text2){
        int m = text1.size();
        int n = text2.size();

        vector<vector<int>>dp(m+1,vector<int>(n+1),0);

        for(int i=m-1;i>=0;i--){
                for(int j=n-1;j>=0;j--){
                        if(text1[i]==text2[j]){
                                dp[i][j] = 1+dp[i+1][j+1];
                        }
                        else{
                                dp[i][j] = max(dp[i+1][j],dp[i][j+1]);
                        }
                }
        }
        return dp[0][0];
}

//Best Time to Buy And Sell Stock With Cooldown
->Array of stock prices, find max profit
->After a sell cooldown of 1 day, can't engage in multiple transactions
Ex. prices = [1,2,3,0,2] -> 3, transactions = [buy,sell,cd,buy,sell]

Hint:- DP + state machine: held ---> sold ---> reset ---> held
                        sell      rest       buy
```

```
/*  Time: O(n)
    Space: O(1) -> optimized from O(n) since only need i - 1 prev state
*/

int maxProfit(vector<vector<int>>&prices){
        int n = prices.size();
        int sold =0;
        int hold = INT_MIN;
        int rest =0;

        for(int i=0;i<n;i++){
                int prevSold = sold;
                sold = sold+prices[i];
                hold = max(hold,rest-prices[i]);
                rest = max(rest,prevSold);
        }
        return max(sold,rest);
}
```

//Coin Change II

->Given array of coins & an amount, return no of combos that make up this amount

Ex. amount = 5, coins = [1,2,5] -> 4 (5, 2+2+1, 2+1+1+1, 1+1+1+1+1)

Hint:- "DFS + memoization", 2 choices: either try coin & stay at indx, or don't try & proceed.

```
/*
    Time: O(m x n)
    Space: O(m x n)
*/
int change(int amount,vector<int>&coins){
        return dfs(amount,coins,0,0);
}
private:
/* {(index, sum) -> No. of combos that make up this amount} */
map<pair<int,int>,int>dp;
int dfs(int amount,vector<int>&coins,int i,int sum){
        if(sum==amount){
```

```
                return 1;
        }
        if(sum>amount){
                return 0;
        }

        if(i==coins.size()){
                return 0;
        }

        if(dp.find({i,sum})!= dp.end()){
                return dp({i,sum});
        }

        dp[{i,sum}] = dfs(amount,coins,i,sum+coins[i]) + dfs(amount,coins,i+1,sum);
        return dp[{i,sum}];
}
```

//Target Sum

->Given int array & a target, want to build expressions w/ '+' & '-'

->Return number of different expressions that evaluates to target

Hint:- "Recursion with memoization", cache on (index, total), which stores no. of ways

If total ever reaches the target, return 1 (this is a way), else 0

```
/*    Time: O(n x target)
      Space: O(n x target)
*/
int findTargetSum(vector<int>&nums,int target){
        return backtrack(nums,target,0,0);
}
private:
        /* {(index, total) -> no of ways} */
        map<pair<int,int>,int>dp;

int backtrack(vector<int>&nums,int target,int i,int total){
        if(i==nums.size()){
```

```
                return total==target?1:0;
        }
        if(dp.find({i,total})!= dp.end()){
                return dp[{i,total}];
        }

        dp[{i,total}] = backtrack(num,target,i+1,total+nums[i]) +
backtrack(num,target,i+1,total-nums[i]);

        return dp[{i,total}];
}
```

//Interleaving String

->Given 3 strings, find if s3 is formed by interleaving of s1 & s2

Ex. s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac" -> true

Hint:- "DFS + memo", cache on s1 & s2 indices i & j

2 choices: either take s1 & iterate i, or take s2 & iterate j

```
/*  Time: O(m x n)
   Space: O(m x n)
*/
bool isInterleave(string s1,string s2,string s3){
        if(s3.size()!=s1.size()+s2.size()){
                return false;
        }
        return dfs(s1,s2,s3,0,0);
}

private:
        map<pair<int,int>,bool>dp;
bool dfs(string s1,string s2,string s3, int i,int j){
        if(i==s1.size() && j==s2.size()){
                return true;
        }
        if(dp.find({i,j})!= dp.end()){
                return dp[{i,j}];
```

```
        }

        if(i<s1.size() && s1[i]==s3[i+j] && dfs(s1,s2,s3,i+1,j)){
                return true;
        }
        if(j<s2.size() && s2[j]==s3[i+j] && dfs(s1,s2,s3,i,j+1)){
                return true;
        }

        dp[{i,j}] = false;
        return dp[{i,j}];
}
```

//Edit Distance (V.IMP)

->Given 2 strings, return minimum number of operations to convert word1 to word2

Hint:- Naive appr: check all possible edit sequences & choose shortest one

Optimal appr: Use DP, if chars at i & j same, no operations needed, else 3 cases:

(1) replace (i - 1, j - 1),

(2) delete (i - 1, j),

(3) insert (i, j - 1)

```
/*  Time: O(m x n)
   Space: O(m x n)
*/
int minDist(string word1, string word2){
        if(word1.empty() && word2.empty()){
                return 0;
        }
        if(word1.empty() || word2.empty()){
                return 1;
        }

        int m = word1.size();
        int n = word2.size();

        vector<vector<int>>dp(m+1,vector<int>(n+1,0));
```

```
        for(int i=1;i<=m;i++){
                dp[i][0] = i;
        }
        for(int j=1;j<=n;j++){
                dp[0][j] = j;
        }

        for(int i=1;i<=m;i++){
                for(int j=1;j<=n;j++){
                        if(word1[i-1] == word2[j-1]){
                                dp[i][j] = dp[i-1][j-1];
                        }
                        else{
                                dp[i][j] = min(dp[i-1][j-1],min(dp[i-1][j],dp[i][j-1]))+1;
                        }
                }
        }
        return dp[m][n];
}

/*Since we only need at most dp[i - 1][j - 1], can space optimize to O(n) */
int minDistance(string word1,string word2){
        int m = word1.size();
        int n = word2.size();

        int prev = 0;

        vector<int>curr(n+1);
        for(int j=1;j<=n;j++){
                curr[j] = j;
        }

        for(int i=1;i<=m;i++){
                prev = curr[0];
                curr[0] = i;
                for(int j=1;j<=n;j++){
```

```
                int temp = curr[j];
                if(word1[i-1] == word2[j-1]){
                        curr[j] = prev;
                }
                else{
                        curr[j] = min(prev,min(curr[j-1],curr[j]))+1;
                }
                prev = temp;
            }
        }
        return curr[n];
}

//Longest Increasing Path In a Matrix (IMP)
-> Given matrix, return length of longest increasing path
   Ex. matrix = [[9,9,4],[6,6,8],[2,1,1]] -> 4, [1,2,6,9]
Hint:-  "DFS + memo", cache on indices, compare to prev for increasing check
/*  Time: O(m x n)
   Space: O(m x n)
*/

int longestIncreasingPath(vector<vector<int>>&matrix){
        int m = matrix.size();
        int n = matrix[0].size();

        int ans =0;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                ans = max(ans,dfs(matrix,-1,i,j,m,n));
            }
        }
        return ans;
}
private:
map<pair<int,int>,int>dp;
int dfs(vector<vector<int>>&matrix,int prev,int i,int j,int m,int n){
```

```
        if(i<0||i>=m||j<0|j>=n || matrix[i][j]<=prev){
                return 0;
        }

        if(dp.find({i,j})!= dp.end()){
                return dp({i,j});
        }

        int ans =1;
        ans = max(ans,1+dfs(matrix,matrix[i][j],i-1,j,m,n));
        ans = max(ans,1+dfs(matrix,matrix[i][j],i+1,j,m,n));
        ans = max(ans,1+dfs(matrix,matrix[i][j],i,j-1,m,n));
        ans = max(ans,1+dfs(matrix,matrix[i][j],i,j+1,m,n));
        dp[{i,j}] = ans;
                return dp[{i,j}];
}

//Distinct Subsequences (IMP)
-> Given 2 strings s & t: Return no. of distinct subsequences of s which equals t
 Ex. s = "rabbbit", t = "rabbit" -> 3, RABBbIT, RAbBBIT, RABbBIT
 Hint:- "DFS + memo", cache on i & j indices to the no. of distinct subseq
  2 choices: if chars equal, look at remainder of both s & t
           if chars not equal, only look at remainder of s
/*  Time: O(m x n)
   Space: O(m x n)
*/
int numDistinct(string s,string t){
        return dfs(s,t,0,0);
}
{(i, j) -> # of distinct subsequences
map<pair<int,int>,int>dp;

int dfs(string &s,string &t,,int i,int j){
        if(j==t.size())
                return 1;
        if(i==s.size())
```

```
                return 0;

        if(dp.find({i,j})!= dp.end()){
                return dp[{i,j}];
        }

        if(s[i]==t[j]){
                dp[{i,j}] = dfs(s,t,i+1,j+1) + dfs(s,t,i+1,j);
        }
        else{
                dp[{i,j}] = dfs(s,t,i+1,j);
        }
        return dp[{i,j}];
}
```

//Burst Balloon (V.IMP)
->Given array of balloons with coins, if burst ith, get (i-1) + i + (i+1) coins
->Return max coins can collect by bursting the balloons wisely.
Hint:- Use DP to return max coins obtainable in each interval [left, right]
Divide & conquer left & right depends on previous bursts, so think "backwards"
Instead of which one to burst first, need to think which one to burst last

```
/*  Time: O(n^3) -> O(n^2) states, for each states, determining max coins is O(n)
   Space: O(n^2) -> O(n^2) to store all states
*/
int maxCoins(vector<int>&nums){
        int n = nums.size();

        nums.insert(nums.begin(),1);
        nums.insert(nums.end(),1);

        //cache results of dp
        vector<vector<int>>memo(n,vector<int>(n,0));

        // 1 & n - 2 since we can't burst our fake balloons
        return dp(nums,memo,1,n-2);
```

```
}

int dp(vector<int>&nums,vector<vector<int>&memo,int left,int right){
        // base case interval is empty, yields 0 coins
        if(right-left<0){
                return 0;
        }

        //we've already seen this, return from cache
        if(memo[left][right]>0){
                return memo[left][right];
        }

        int ans =0;
        for(int i=left;i<= right;i++){
                // nums[i] is the last burst
                int curr = nums[left-1] * nums[i]*nums[right+1];
                // nums[i] is fixed, recursively call left & right sides
                int remaining = dp(nums,memo,left,i-1)+dp(nums,memo,right,i+1);
                ans = max(ans,curr+remaining);
        }
        // add to cache
        memo[left][right] = ans;
        return ans;
}

//Regular Expression Matching (V.IMP)
->Given string & pattern, implement RegEx matching
   '.' -> matches any single character
   '*' -> matches zero or more of the preceding element
Matching should cover the entire input string (not partial)
Ex. s = "aa", p = "a" -> false, "a" doesn't match entire string "aa"
Hint:- "DFS + memo", 2 choices at a *: either use it, or don't use it
/*   Time: O(m x n)
    Space: O(m x n)
*/
```

```
bool isMatch(string s,string p){
        return dfs(s,p,0,0);
}
private:
map<pair<int,int>,bool>dp;
bool dfs(string &s,string &p, int i,int j){
        if(dp.find({i,j})!= dp.end()){
                return dp[{i,j}];
        }

        if(i>=s.size() && j>=p.size()){
                return true;
        }

        if(j>=p.size()){
                return false;
        }

        bool match = i<s.size() && (s[i]== p[j] || p[j]=='.');
        if(j+1<p.size() && p[j+1]=='*'){
                // choices: either (1) don't use *, or (2) use *
                dp[{i,j}]= dfs(s,p,i,j+2) || (match && dfs(s,p,i+1,j));
                return dp[{i,j}];
        }

        if(match){
                dp[{i,j}] = dfs(s,p,i+1,j+1);
                return dp[{i,j}];
        }

        dp[{i,j}] = false;
        return dp[{i,j}];
}

/***Maths**/
//Rotate Image
```

-> Given a 2D image matrix, rotate image 90 degree Clock-wise

Hint:- Transpose + reflect (rev on diag then rev left to right)

```
/*  Time: O(n^2)
    Space: O(1)
*/
void rotate(vector<vector<int>>&matrix){
        int n = matrix.size();

        for(int i=0;i<n;i++){
                for(int j=i;j<n;j++){
                        swap(matrix[i][j],matrix[j][i]);
                }
                reverse(matrix[i].begin(),matrix[i].end());
        }
}
```

//Spiral Matrix

-> Given a matrix, return all elements in spiral order

Hint:- Set up boundaries, go outside in Clock-wise: "top->right->bottom->left"

```
/*  Time: O(m x n)
    Space: O(m x n)
*/
vector<int>spiralOrder(vector<vector<int>>&matrix){
        int left = 0, top = 0;
        int right = matrix[0].size()-1, bottom = matrix.size()-1;

        vector<int>ans;

        while(top<=bottom && left<=right){
                for(int j=left;j<=right;j++){
                        ans.push_back(matrix[top][j]);
                }
                top++;

                for(int i=top;i<=bottom;i++){
                        ans.push_back(matrix[i][right]);
```

```
                }
                right--;

                if(top<=bottom){
                        for(int j=right;j>=left;j--){
                                ans.push_back(matrix[bottom][j]);
                        }
                }
                bottom--;

                if(left<=right){
                        for(int i=bottom;i>=top;i--){
                                ans.push_back(matrix[i][left]);
                        }
                }
                left++;
        }
        return ans;
}

//Set Matrix Zeroes
->Given matrix, if element 0, set entire row/col to 0
Hint:- Use 1st row/col as flag to determine if entire row/col 0
/*  Time: O(mn)
   Space: O(1)
*/
void setZeroes(vector<vector<int>>&matrix){
        int m = matrix.size();
        int n = matrix[0].size();

        bool isFirstRowZero = false;
        bool isFirstColZero = false;

        for(int i=0;i<m;i++){
                if(matrix[i][0]==0){
                        isFirstColZero = true;
```

```
                break;
        }
}

for(int j=0;j<n;j++){
        if(matrix[0][j]==0){
                isFirstRowZero = true;
                break;
        }
}

for(int i=1;i<m;i++){
        for(int j=1;j<n;j++){
                if(matrix[i][j] == 0){
                        matrix[i][0] = 0;
                        matrix[0][j] = 0;
                }
        }
}

for(int i=1;i<m;i++){
        for(int j=1;j<n;j++){
                if(matrix[i][0]==0 || matrix[0][j]==0){
                        matrix[i][j] = 0;
                }
        }
}

if(isFirstColZero){
        for(int i=0;i<m;i++){
                matrix[i][0] = 0;
        }
}
if(isFirstRowZero){
        for(int j=0;j<n;j++){
                matrix[0][j] = 0;
```

```
                }
        }
}
```

//Happy Number

-> Given num, replace by sum of squares of its digits, Repeat until 1 or endless loop, determine if ends in 1

Ex. n = 19 -> true, 1^2 + 9^2 = 82, 8^2 + 2^2 = 68 ... 1

Hint:- Detect cycle with "slow/fast pointer technique"

If happy will eventually be 1, else pointers will meet

```
/*Time: O(log n)
  Space: O(1)
*/

bool isHappy(int n){
        int slow = n;
        int fast = getNext(n);

        while(slow!= fast && fast !=1){
                slow = getNext(slow);
                fast = getNext(getNext(fast));
        }

        if(fast==1){
                return true;
        }
        return false;
}
int getNext(int n){
        int sum =0;
        while(n>0){
                int digit = n%10;
                n/=10;
                sum += pow(digit,2);
        }
```

```
        return sum;
}
```

//Plus One

-> Given large int as an array, add 1 (consider carry)

Ex. digits = [1,2,3] -> [1,2,4]

Hint:- From right to left, keep carrying until digit < 9, add 1

```
/*  Time: O(n)
    Space: O(1)
*/
vector<int>plusOne(vector<int>&digits){
        for(int i= digits.size()-1;i>=0;i--){
                if(digits[i]<9){
                        digits[i]++;
                        return digits;
                }
                digits[i] = 0;
        }
        digits[0] = 1;
        digits.push_back(0);
        return digits;
}
```

—---------- 🔥END 🔥—----------