Data Structures

2. Add two numbers

-
  ```
  // Definition for singly-linked list.
   struct ListNode {
       int val;
       ListNode *next;
       ListNode() : val(0), next(nullptr) {}
       ListNode(int x) : val(x), next(nullptr) {}
       ListNode(int x, ListNode *next) : val(x), next(next) {}
   };

  // colon operator initializes the values
  ```
- You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit.
- Add the two numbers and return the sum as a linked list.

-
  ```
  Input: l1 = [2,4,3], l2 = [5,6,4]
  Output: [7,0,8]
  Explanation: 342 + 465 = 807.
  ```

```
#include <iostream>

using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};



class Solution {
public:
    static ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
       int carry = 0;
       ListNode * curr = new ListNode(0);
       ListNode * returnVal = curr;
       while(l1 != nullptr || l2!=nullptr){
           int numa = 0;
           int numb = 0;
```

```
            int sum = 0;

            if(l1!=nullptr){
                numa = l1->val;
                l1 = l1->next;
            }

            if(l2!=nullptr){
                numb = l2->val;
                l2 = l2->next;
            }

            ListNode * node = new ListNode;

            sum = numa + numb + carry;

            if(sum >= 10){
                node->val = sum % 10;
                carry = 1;
            } else {
                node->val = sum;
                carry = 0;
            }

             curr->next = node;
             curr = curr->next;

        }

         if(carry > 0){
             curr->next = new ListNode(carry);
         }

        return returnVal->next;
    }
};



int main(){



    ListNode n3(3);
    ListNode n2(4, &n3);
    ListNode n1(2, &n2);

    ListNode n6(4);
    ListNode n5(6, &n6);
    ListNode n4(5, &n5);
    Solution::addTwoNumbers(&n1, &n4);

    return 0;
}
```

- create two nodes one to keep track of the other
- the other starts at zero outside the loop
- its next is the new node,

-
  ```
      curr->next = node;
      curr = curr->next;
  ```
-

1. Two sum

- To get index from iterator : `distance(nums.begin(), itr)`
- convert set to vector

```
    std::set<char> s = { 'a', 'b', 'c', 'd', 'e' };

    std::vector<char> v(s.begin(), s.end());
```

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

```
// O(n^2)
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {

        set<int> s;
        for(auto itr1 = nums.begin(); itr1 != nums.end(); ++itr1 ){
            for(auto itr2 = nums.begin(); itr2 != nums.end(); ++itr2 ){
                    if(*itr1 + *itr2 == target){
                      if(itr1 != itr2){
                        s.emplace(distance(nums.begin(), itr1));
                        s.emplace(distance(nums.begin(), itr2));
                      }
                    }
            }
        }
```

```
        vector<int> v(s.begin(), s.end());
        return v;
    }
};
```

- Unordered map implements hash table
- Both at() and operator[] is used to refer the element present at the given ***position***, the only difference is, at() throws out-of-range exception whereas operator[] shows **undefined behavior** i.e. if operator[] is used to find the value corresponding to key and if key is not present in unordered map, it will first insert the key into the map and then assign the default value '0' corresponding to that key. .
-
```
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    // Declaring umap to be of <string, double> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, double> umap;

    // inserting values by using [] operator
    umap["PI"] = 3.14;
    umap["root2"] = 1.414;
    umap["root3"] = 1.732;
    umap["log10"] = 2.302;
    umap["loge"] = 1.0;

    // inserting value by insert function
    umap.insert(make_pair("e", 2.718));

    string key = "PI";

    // If key not found in map iterator to end is returned
    if (umap.find(key) == umap.end())
        cout << key << " not found\n\n";

    // If key found then iterator to that key is returned
    else
        cout << "Found " << key << "\n\n";

    key = "lambda";
    if (umap.find(key) == umap.end())
        cout << key << " not found\n";
    else
        cout << "Found " << key << endl;

    //    iterating over all value of umap
    unordered_map<string, double>:: iterator itr;
    cout << "\nAll Elements : \n";
    for (itr = umap.begin(); itr != umap.end(); itr++)
    {
        // itr works as a pointer to pair<string, double>
        // type itr->first stores the key part  and
        // itr->second stores the value part
        cout << itr->first << "  " << itr->second << endl;
     }
}
```

- Binary search tree is implemented by `map`

- Solution

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        set<int> s;
        unordered_map<int, int> table;

        for(auto it = nums.begin(); it != nums.end(); ++it){
          table[*it] = distance(nums.begin(), it);
        }

        for(auto it = nums.begin(); it != nums.end(); ++it){
            int diff =target - *it ;
            if(table.find(diff) != table.end() && table[diff] != distance(nums.begin(),
it)){
              s.emplace(table[diff]);
              s.emplace(distance(nums.begin(), it));
            }
        }

        vector<int> v(s.begin(), s.end());
        return v;
    }
};
```

53. Maximum subarray

Whenever you see a question that asks for the maximum or minimum of something, consider Dynamic Programming as a possibility.

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

```
// our solution
class Solution {
public:
    int generateAllSubArray(vector<int> nums, int start, int end){
        int sum = INT32_MIN;
        if(end == nums.size()){
            return sum ;
        }
        else if(start > end){
            return generateAllSubArray(nums, 0, end+1);
        } else {
            int val = 0;
            //cout<< "sub: ";
            for(int i = start; i<= end; i++){
                //cout << nums[i] << " ";
                val += nums[i];
            }
            //cout << endl;

            if(sum < val) sum = val;

            int r = generateAllSubArray(nums, start+1, end);
            if (r > sum){
                return r;
            } else return sum;
        }
    }


    int maxSubArray(vector<int>& nums) {
        return  generateAllSubArray(nums, 0, 0);
    }
};

int main(){

    vector<int> nums = {1,2,3,4,5};

    int val =   generateAllSubArray(nums, 0, 0);
    cout << " val : " << val;
    return 0;
}
```

printing all subarrays

```
class Solution {
public:
    // iterative approach
    static void maxSubArray(vector<int>& nums) {
        int startMarker = 0;
        int endMarker = 0;
        while(endMarker != nums.size()){

          if(startMarker > endMarker){
            endMarker++;
            startMarker = 0;
          }

          else {
            //cout << " start: " << startMarker << " end " << endMarker << endl;
            for(int i = startMarker; i <= endMarker; i++){
              cout << nums[i] << " ";
            }
            cout << endl;

            startMarker++;
          }

        }
    }
};
```

max subarray solution

- Kadane's Algorithm.
- Let's focus on one important part: where the optimal subarray **begins**.
- We need a general way to figure out when a part of the array is **worth** keeping.
- any subarray whose sum is *positive* is worth keeping.
- Let's start with an empty array, and iterate through the input, adding numbers to our array as we go along. Whenever the sum of the array is negative, we know the entire array is not worth keeping, so we'll reset it back to an empty array.
- However, we don't actually need to build the subarray, we can just keep an integer variable `current_subarray` and add the values of each element there. When it becomes negative, we reset it to 0 (an empty array).
- Algorithm
    - Initialize 2 integer variables. Set both of them equal to the first value in the array.

        - `currentSubarray` will keep the running count of the current subarray we are focusing on.
        - `maxSubarray` will be our final return value. Continuously update it whenever we find a bigger subarray.
    - Iterate through the array, starting with the 2nd element (as we used the first element to initialize our variables). For each number, add it to the `currentSubarray` we are building. If `currentSubarray` becomes negative, we know it isn't worth keeping, so throw it away. Remember to update `maxSubarray` every time we find a new maximum.
    - Return `maxSubarray`.

```
class Solution {
public:
    // iterative approach
    int  maxSubArray(vector<int>& nums) {

      int maxSum = nums[0];
      int currentSum = nums[0];

      for(int i = 1; i< nums.size(); i++){
          // compare current num with running current sum
        currentSum = max(nums[i], currentSum + nums[i]);

          // compare currentrunningsum with max
        maxSum = max(currentSum, maxSum);
      }

      return maxSum;


    }
};
```

88. Merge Sorted Array

- You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

  Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

  The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

      for(int i = 0; i < n; i++){
        nums1[m+i] = nums2[i];
      }

      sort(nums1.begin(), nums1.end());

    }
};
```

350. Intersection of Two Arrays II

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

My approach

-
  ```
  class Solution {
  public:
      vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        map<int, int> map;
        vector<int> v;


        for(int i = 0; i < nums1.size(); i++){
          for(int j = 0; j < nums2.size(); j++){
            if(nums1[i] == nums2[j]){
                map[nums1[i]]++;
            }
          }
        }

        for(auto it : map){
            int i =  it.second == 2 ? 1 : it.second/2;

            while(i != 0){
              v.push_back(it.first);
              i--;
            }




        }

        return v;
  ```

```
    }
};
```

Correct Approach

- If an interviewer gives you this problem, your first question should be - *how should I handle duplicates?*
- Your second question, perhaps, can be about the order of inputs and outputs. Such questions manifest your problem-solving skills, and help you steer to the right solution.
- We collect numbers and their counts from one of the arrays into a hash map. Then, we iterate along the second array, and check if the number exists in the hash map and its count is positive. If so - add the number to the result and decrease its count in the hash map.

  ```
  class Solution {
  public:
      vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        map<int, int> map;
        vector<int> v;

        for(auto i: nums1){
          map[i]++;
        }

        for(auto i: nums2){
          auto it = map.find(i);
          if(it != map.end()){
            if(it->second > 0){
              v.push_back(it->first);
              map[i]--;
            }
          }
        }

        return v;
      }
  };
  ```

121. Best time to buy and sell stock

my approach

- Wrong

  -
    ```
    class Solution {
    public:
        int maxProfit(vector<int>& prices) {
    ```

```
        auto min_it = min_element(begin(prices), end(prices));
        int minNdx = distance(prices.begin(), min_it);
        auto max_it = max_element(prices.begin()+minNdx, prices.end());
        if(*max_it - *min_it > 0){
          return *max_it - *min_it;
        } else {
          return 0;
        }
    }
};
```

- Slow

  -
    ```
    class Solution {
    public:
        int maxProfit(vector<int>& prices) {
          int maxp = 0;
          int min = 0;
          int max = 0;
          for(int i = 0; i < prices.size(); i++){
            for(int j = i+1; j < prices.size(); j++){
              int i_profit = prices[j] - prices[i];
              if(i_profit > maxp){
                //min = prices[i];
                //max = prices[j];
                maxp = i_profit;
              }
            }
          }

          return maxp;
        }
    };
    ```

- their solution

  -
    ```
    class Solution {
    public:

        int maxProfit(vector<int>& prices) {
          int maxprofit = 0;
          int minprice = INT_MAX;
          for(int i = 0; i < prices.size(); i++){
              if(prices[i] < minprice){
                minprice = prices[i];
              } else if(prices[i] - minprice > maxprofit){
                maxprofit = prices[i] - minprice;
              }
          }
    ```

```
        return maxprofit;
    }
};
```

◦

566. Reshape the Matrix

In MATLAB, there is a handy function called `reshape` which can reshape an `m x n` matrix into a new one with a different size `r x c` keeping its original data.

You are given an `m x n` matrix `mat` and two integers `r` and `c` representing the number of rows and the number of columns of the wanted reshaped matrix.

The reshaped matrix should be filled with all the elements of the original matrix in the same row-traversing order as they were.

If the `reshape` operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

- Wrong answer

-
  ```
  class Solution {
  public:
      vector<vector<int>> matrixReshape(vector<vector<int>>& mat, int r, int c) {
          int curr_row_num = mat.size();
          int curr_col_num = mat.at(0).size();

          vector<vector<int>> v;

          int max_row_or_col_num = curr_row_num * curr_col_num;

          if((r > max_row_or_col_num) || (c > max_row_or_col_num)){
            return mat;
          }

          if((r < 1) || (c < 1)){
            return mat;
          }

          if( r*c > max_row_or_col_num  ){
            return mat;
          }


          vector<int> tempv;
          for(int i = 0; i < mat.size(); i++){
            for(int j = 0; j < mat.at(i).size(); j++){
  ```

```
          if(v.size() < r){
            if(tempv.size() < c ){
              cout << " if " << endl;

              tempv.push_back(mat[i][j]);
              print(tempv);

            } else {
              cout << "else " << endl;
              print(tempv);
              v.push_back(tempv);
              tempv.clear();

              tempv.push_back(mat[i][j]);
              print(tempv);
            }
          }


        }

      }

      if(!tempv.empty() && v.size() < r) {
        v.push_back(tempv);
        tempv.empty();
      }

      if(tempv.empty()){
        return mat;
      }

      if(v.size() * v.at(0).size() == r*c){
        return v;
      } else return mat;



  }
};
```

-
  ```
  class Solution {
  public:
      vector<vector<int>> matrixReshape(vector<vector<int>>& mat, int r, int c) {

        queue<int> q;
        vector<vector<int>> finalv;
        vector<int> intv;

        for(int i = 0; i < mat.size(); i++){
          for(int j = 0; j < mat.at(i).size(); j++){
  ```

```
                q.push(mat[i][j]);
            }
        }

        if(q.size() < r*c) {
            return mat;
        }

        while(!q.empty()){
            int num = q.front();
            q.pop();
            if(finalv.size() < r){
                if(intv.size() < c){
                    intv.push_back(num);
                } else {
                    finalv.push_back(intv);
                    intv.clear();
                    intv.push_back(num);
                }
            }
        }

        if(finalv.size() * finalv.at(0).size() < mat.size() * mat.at(0).size()){
            return mat;
        } else return finalv;


    }
};
```

-

Solution 1

- The simplest method is to extract all the elements of the given matrix by reading the elements in a row-wise fashion. In this implementation, we use a queue to put the extracted elements. Then, we can take out the elements of the queue formed in a serial order and arrange the elements in the resultant required matrix in a row-by-row order again.

  The formation of the resultant matrix won't be possible if the number of elements in the original matrix isn't equal to the number of elements in the resultant matrix.

Approach 3: Using division and modulus

- The element `nums[i][j]` of nums array is represented in the form of a one dimensional array by using the index in the form: `nums[n*i + j]` where `n` is the number of columns in the given matrix.
-
  ```
  nums[i][j] = nums[n*i + j]
  ```
- if `count` is the total number of elements in the two D array

-
  ```
  res[count/c][count%c]

  count/c //is the row number
  count%c //is the coloumn number.
  ```

```
class Solution {
public:
    static vector<vector<int>> matrixReshape(vector<vector<int>>& mat, int r, int c) {

      vector<vector<int>> finalv;
      int count = 0;

      if(mat.size() == 0 || r*c != mat.size() * mat.at(0).size())
        {
          return mat;
        }


      for(int i = 0; i < mat.size(); i++){
        for(int j = 0; j < mat.at(i).size(); j++){
            finalv[count / c][count % c] = mat[i][j];
            count++;
        }
      }

      return finalv;

    }
};
```

118. Pascal's Triangle

- Given an integer `numRows`, return the first numRows of **Pascal's triangle**.

  In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown

  -
    ```
    Input: numRows = 5
    Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
    ```

Dynamic Programming framework

1. Base Case
   - The value at the beginning or end of a row is always 1
2. Recurrence Relation
   - Each number is the sum of the two numbers directly above it

- `triangle[row][col] = triangle[row-1][col-1] + triangle[row-1][col]`

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> v2;
        vector<int> v;
        v.push_back(1);
        v2.push_back(v);

        for(int i = 1; i < numRows; i++){
          vector<int> currentRow;
          vector<int> prevRow = v2.at(i-1);
          currentRow.push_back(1);

          for(int col = 1; col < prevRow.size(); col++){
            currentRow.push_back(prevRow[col-1] + prevRow[col]);
          }

          currentRow.push_back(1);
          v2.push_back(currentRow);

        }

        return v2;
    }
};
```

36. Valid soduko

- flipping `i` and `j` gives row wise traversal or column wise traversal

```
vector<vector<int>> v = {{0,1,2}, {3,4,5}, {6,7,8}};

for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        cout << v[i][j] << " ";
    }
}

// 0 1 2 3 4 5 6 7 8

for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        cout << v[j][i] << " ";
    }
}
```

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

Dont be afraid to use as many hashmaps as possible

74. Search a 2D Matrix

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

```
class Solution {
public:
    bool binarySearch(vector<int>v, int target){
        int low = 0;
        int high = v.size()-1;

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

          if(v[mid] == target){
            return true;
          }

          if(target > v[mid]){
            low = mid+1;
          }

          if(target < v[mid]){
            high = mid -1;
          }
        }

        return false;
    }

    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        for(int i = 0; i < matrix.size(); i++){
          bool state = binarySearch(matrix[i], target);
          if(state){
            return true;
          }
        }
        return false;
    }
};
```

387. First Unique Character in a String

- Given a string `s`, *find the first non-repeating character in it and return its index*. If it does not exist, return `-1`.

```
class Solution {
public:
    int firstUniqChar(string s) {
        unordered_map<char,int> charCounter;
        for(int i = 0; i < s.size(); i++){
          charCounter[s[i]]++;
        }

        vector<char> onItemedArray;
        for(auto a: charCounter){
          if(a.second == 1){
            onItemedArray.push_back(a.first);
          }
        }

        for(int i = 0; i < s.size(); i++){
          auto it = find(onItemedArray.begin(), onItemedArray.end(), s[i]);
          if(it != onItemedArray.end()){
            return i;
          }
        }

        return -1;

    }
};
```

383. Ransom Note

Given two strings `ransomNote` and `magazine`, return `true` *if* `ransomNote` *can be constructed from* `magazine` *and* `false` *otherwise*.

Each letter in `magazine` can only be used once in `ransomNote`.

```
class Solution {
public:
    static bool canConstruct(string ransomNote, string magazine) {
        if(magazine.size() < ransomNote.size()){
          return false;
        }
        unordered_map<char, int> charCount;
        for(auto a: ransomNote){
          charCount[a]++;
        }

        for(auto b: magazine){
          if(charCount.find(b) != charCount.end()){
            charCount[b]--;
          }
        }

        for(auto c: charCount){
          if(c.second >=1){
            return false;
          }
        }

        return true;
    }
};
```

242. Valid Anagram

Given two strings `s` and `t`, return `true` *if* `t` *is an anagram of* `s`*, and* `false` *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        unordered_map<char, int> sCharCount;
        for(auto a: s){
          sCharCount[a]++;
        }

        for(auto a : t){
          sCharCount[a]--;
        }

        for(auto m: sCharCount){
          if(m.second != 0){
            return false;
          }
        }

        return true;
    }
};
```

141. Linked List

- Given `head`, the head of a linked list, determine if the linked list has a cycle in it.
- Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

```
// first approach

class Solution {
public:
    bool hasCycle(ListNode *head) {

      if(head == nullptr || head->next == nullptr){
        return false;
      }


      ListNode* first = head ->next;

      if(first->next == nullptr){
        return false;
      }


      ListNode * itr = first->next;

      while(itr != nullptr){
        if(itr == first){
          return true;
        }
        itr = itr->next;
      }
      return false;
    }
};
```

Floyd's cycle finding algorithm

- Imagine two runners running on a track at different speed. What happens when the track is actually a circle?
- The space complexity can be reduced to O(1)$O(1)$ by considering two pointers at **different speed** - a slow pointer and a fast pointer. The slow pointer moves one step at a time while the fast pointer moves two steps at a time.
- If there is no cycle in the list, the fast pointer will eventually reach the end and we can return false in this case.

```
class Solution {
public:
    bool hasCycle(ListNode *head) {

      if(head == nullptr){
        return false;
      }

      // they must start at differnt points
      ListNode * slow = head;
      ListNode * fast = head->next;

      while(slow != fast){
        if(fast == nullptr || fast->next == nullptr){
          return false;
        }

        slow = slow->next;
        fast = fast->next->next;
      }
      return true;
    }
};
```

21. Merge two sorted list

Sorting a list

- Merge sort is often preferred for sorting a linked list
-

\

144. Binary Tree Preorder traversal

- Given the `root` of a binary tree, return *the preorder traversal of its nodes' values*.

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};
```

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
      std::vector<int> v;
      stack<TreeNode *> s;

      if(root == nullptr){
        return v;
      }

      s.push(root);

      while(!s.empty()){
          TreeNode * node = s.top();
          s.pop();
          v.push_back(node->val);
          if(node->right != nullptr){
            s.push(node->right);
          }

          if(node->left != nullptr){
            s.push(node->left);
          }

      }
        return v;
    }
};
```

How to traverse the tree

There are two general strategies to traverse a tree:

- *Breadth First Search* (`BFS`)

  We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

- *Depth First Search* (`DFS`)

  In this strategy, we adopt the `depth` as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

  The DFS strategy can further be distinguished as `preorder`, `inorder`, and `postorder` depending on the relative order among the root node, left node and right node.

94. Binary Tree Inorder Traversal

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
      std::vector<int> v;
      stack<TreeNode *> s;

      if(root == nullptr){
        return v;
      }

      TreeNode * current = root;

      while(current != nullptr || !s.empty()){
        while(current != nullptr){
          s.push(current);
          current = current->left;
        }

        current = s.top();
        s.pop();
        v.push_back(current->val);
        current = current->right;
      }

      return v;
    }
};
```

Binary tree post Order traversal

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        std::vector<int> v;
        std::stack<TreeNode *> s;

        TreeNode * current = root;
        TreeNode * previous = nullptr;

        while(current != nullptr || !s.empty()){
          if(current != nullptr){
            s.push(current);
            current = current->left;
          }

          else {
            current = s.top();
            if(current->right == nullptr || current->right == previous){
              v.push_back(current->val);
              s.pop();
              previous = current;
              current = nullptr;
            }
            else{
              current = current->right;
            }
          }




        }

        return v;
    }
};
```

102. Binary Tree Level Order Traversal

- Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).
-
  ```

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

          TreeNode * current = root;
          if(root == nullptr){
            return v;
          }
          q.push(current);

          while( !q.empty()){
            vector<int> vt;
            current = q.front(); q.pop();
            vt.push_back(current->val);
            if(current->left != nullptr){
              q.push(current->left);
            }
            if(current->right != nullptr){
              q.push(current->right);
            }
            v.push_back(vt);
          }
          return v;
      }
  };

  ```

-
  ```
  // with level information
  class Solution {
  ```

```
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        queue<TreeNode *> q;
        vector<vector<int>> v;
        int level = 0;

        TreeNode * current = root;
        if(root == nullptr){
          return v;
        }
        q.push(current);

        while( !q.empty()){

          vector<int> vt;
          v.push_back(vt);
          int itrSize = q.size();

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

            current = q.front(); q.pop();

            v[level].push_back(current->val);

            if(current->left != nullptr){
              q.push(current->left);
            }
            if(current->right != nullptr){
              q.push(current->right);
            }

          }
          level++;

        }
        return v;
    }
};
```

-

104. Maximum depth of binary tree

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        queue<TreeNode*> q;
        int level = 0;
        TreeNode * current = root;

        if(current == nullptr){
          return 0;
        }

        q.push(current);

        while(!q.empty()){
          int qSize = q.size();
          for(int i = 0; i< qSize; i++){
            current = q.front(); q.pop();
            if(current->left != nullptr){
              q.push(current->left);
            }
            if(current->right != nullptr){
              q.push(current->right);
            }
          }

          level++;

        }
        return level;
    }
};
```

101. Symmetric Tree

Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        queue<TreeNode *> q;
        if(root == nullptr){
          return false;
        }

        TreeNode * current = root;
        q.push(current);
        q.push(current);

        while(!q.empty()){
          TreeNode * t1 = q.front(); q.pop();
          TreeNode * t2 = q.front(); q.pop();
```

```
            if(t1 == nullptr && t2 == nullptr){
              continue;
            }

            if(t2 == nullptr || t1 == nullptr){
              return false;
            }

            if(t2->val != t1->val){
              return false;
            }

            // mirror image right and left flipped
            q.push(t1->right);
            q.push(t2->left);
            q.push(t1->left);
            q.push(t2->right);

          }

          return true;
      }
  };
```

226. Invert Binary Trees

- Given the `root` of a binary tree, invert the tree, and return *its root*

recursive

```
class Solution {
public:

    TreeNode* invertTree(TreeNode* root) {

        if(root == nullptr){
          return root;
        }

        TreeNode * leftNode = invertTree(root->left);
        TreeNode * rightNode =invertTree( root->right);

        root->left = rightNode;
        root->right = leftNode;

        return root;
    }
};
```

- Since each node in the tree is visited only once, the time complexity is $O(n)$, where *n* is the number of nodes in the tree
- Because of recursion, O(h) function calls will be placed on the stack in the worst case, where *h* is the height of the tree. O(n)

iterative

```
class Solution {
public:

    TreeNode* invertTree(TreeNode* root) {
      queue<TreeNode *> q;
      if(root == nullptr){
        return nullptr;
      }

      q.push(root);

      if(!q.empty()){
        TreeNode * current = q.front(); q.pop();
        cout << current->val << " ";
        swap(current->left, current->right);

        if(current->left != nullptr){
          q.push(current->left);
        }

        if(current->right != nullptr){
          q.push(current->right);
        }

      }
      return root;
    }
};
```

122.Path Sum

- Given the `root` of a binary tree and an integer `targetSum`, return `true` if the tree has a **root-to-leaf** path such that adding up all the values along the path equals `targetSum`.

A **leaf** is a node with no children.

- Our solution

-
  ```
  class Solution {
  ```

```
public:
    bool track = false;
    void check(TreeNode * root, int targetSum, int runSum){
      if(root == nullptr){
        cout << " target: " << targetSum << " rumSum : " << runSum <<endl;
        if(targetSum == runSum){
          track = true;
        }
        return;
      }

      check(root->left, targetSum, runSum+root->val);
      check(root->right, targetSum, runSum+root->val);

    }

    bool hasPathSum(TreeNode* root, int targetSum) {
      if(root == nullptr){
        return false;
      }
        check(root, targetSum, 0);
        return track;

    }
};
```

right solution

```
class Solution {
public:

    bool hasPathSum(TreeNode* root, int targetSum) {
      if(root == nullptr){
        return false;
      }

      targetSum -= root->val;

      if(root->left == nullptr && root->right == nullptr){
        return targetSum == 0;
      }

      return hasPathSum(root->left, targetSum) || hasPathSum(root->right, targetSum);

    }
};
```

700. Search in a binary Search tree

- You are given the `root` of a binary search tree (BST) and an integer `val`.

  Find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.

```
// iterative
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == nullptr){
          return nullptr;
        }

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

        while(!q.empty()){
          TreeNode * current = q.front(); q.pop();
          if(current->val == val){
            return current;
          }
          if(current->left != nullptr){
            q.push(current->left);
          }

          if(current->right != nullptr){
            q.push(current->right);
          }
        }
        return nullptr;
    }
};
```

```
// recursive and smart
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == nullptr){
          return nullptr;
        }

        if(root->val == val){
          return root;
        }

        return val < root->val ? searchBST(root->left, val) : searchBST(root->right, val);
    }
};
```

```
// iterative and smart
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == nullptr){
          return nullptr;
        }

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

        while(!q.empty()){
          TreeNode * current = q.front(); q.pop();
          if(current->val == val){
            return current;
          }

          if(val < current->val){
            if(current->left != nullptr){
              q.push(current->left);
            }
          } else {
            if(current->right != nullptr){
              q.push(current->right);
            }
          }
        }
        return nullptr;
    }
};
```

701. Insert into binary Search tree

```
// correct solution
class Solution {
public:
    void insert(TreeNode * root, int val){
      if(root->val < val){
        if(root->right == nullptr){
          root->right = new TreeNode(val);
        } else {
          insert(root->right, val);
        }
      } else {
        if(root->left == nullptr){
          root->left = new TreeNode(val);
        } else {
          insert(root->left, val);
        }
      }

    }

    TreeNode* insertIntoBST(TreeNode* root, int val) {
      if(root == nullptr){
        root = new TreeNode(val);

      } else {
        insert(root, val);
      }

        return root;
    }
};
```

```
// smart solution
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
      if(root == nullptr){
        return new TreeNode(val);
      }

      if(root->val > val){
        root->left = insertIntoBST(root->left, val);
      } else {
        root->right = insertIntoBST(root->right, val);
      }
      return root;

  }
};
```

98. Validate Binary Search Tree

- Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

  A **valid BST** is defined as follows:

  - The left subtree of a node contains only nodes with keys **less than** the node's key.
  - The right subtree of a node contains only nodes with keys **greater than** the node's key.
  - Both the left and right subtrees must also be binary search trees.
-

```
// not a solution
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if(root == nullptr){
          return true;
        }

        bool leftValid = (root->left != nullptr) && (root->val <= root->left->val);
        bool rightValid = (root->right != nullptr) && (root->val >= root->right->val);

        if(leftValid || rightValid){
          return true;
        } else return false;

        return  isValidBST(root->right) && isValidBST(root->left);
    }
};
```

653. Two Sum IV - Input is a BST

- Given the `root` of a Binary Search Tree and a target number `k`, return `true` *if there exist two elements in the BST such that their sum is equal to the given target*.

```
class Solution {
public:
    vector<int> count;
    bool found = false;
    void itr(TreeNode * root, int k){
      if(root == nullptr){
        return;
      }
      int seeking = k - root->val;
      cout << "seeking: " << seeking << " val: " << root->val << endl;

      auto it = find(count.begin(), count.end(),  root->val);

      if(it != count.end()){
        found = true;
        return;
      }

      count.push_back(seeking);

      if(root->left != nullptr){
        itr(root->left, k);
      }
```

```
        if(root->right != nullptr){
          itr(root->right, k);
        }

      }
      bool findTarget(TreeNode* root, int k) {
          itr(root, k);
          for(auto a: count){
            cout << a << " ";
          }
          return found;
      }
};
```

- When you are pushing to the count array you are calculating the seeking value. but when you are searching you are searching for the current value not seeking.
    - Seeking is added to the lookup array. Seeking is only pushed
    - Current value is searched from the array not seeking itself.

235. Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**)."

algorithm

1. Start traversing the tree from the root node.
2. If both the nodes `p` and `q` are in the right subtree, then continue the search with right subtree starting step 1.
3. If both the nodes `p` and `q` are in the left subtree, then continue the search with left subtree starting step 1.
4. If both step 2 and step 3 are not true, this means we have found the node which is common to node `p`'s and `q`'s subtrees. and hence we return this common node as the LCA.

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(p->val > root->val && q->val > root->val){
          return lowestCommonAncestor(root->right, p, q);
        }
        else if(p->val < root->val && q->val < root->val){
          return lowestCommonAncestor(root->left, p, q);
        }
        else return root;
    }
};
```

- time complexity is `O(n)`
  - N is number of nodes in bst
- space complexity is `O(n)`
  - because maximum space utilized by recursion stack would be N since the height max is N
-

```
// iterative
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

        // Value of p
        int pVal = p.val;

        // Value of q;
        int qVal = q.val;

        // Start from the root node of the tree
        TreeNode node = root;

        // Traverse the tree
        while (node != null) {

            // Value of ancestor/parent node.
            int parentVal = node.val;

            if (pVal > parentVal && qVal > parentVal) {
                // If both p and q are greater than parent
                node = node.right;
            } else if (pVal < parentVal && qVal < parentVal) {
                // If both p and q are lesser than parent
                node = node.left;
            } else {
                // We have found the split point, i.e. the LCA node.
                return node;
            }
        }
        return null;
    }
}
```

232. Implement Queue using Stacks

- Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

  Implement the `MyQueue` class:

  - `void push(int x)` Pushes element x to the back of the queue.
  - `int pop()` Removes the element from the front of the queue and returns it.
  - `int peek()` Returns the element at the front of the queue.
  - `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.
- first attempt

-
  ```
  class MyQueue {
    private:
      stack<int> s1;
      stack<int> s2;
    public:
        MyQueue() {

        }

        void push(int x) {
            s1.push(x);
        }

        int pop() {
          while(!s1.empty()){
            s2.push(s1.top());
            s1.pop();
          }
          int x = s2.top();
          s2.pop();
          while(!s2.empty()){
            s1.push(s2.top());
            s2.pop();
          }
          return x;
        }

        int peek() {
          while(!s1.empty()){
            s2.push(s1.top());
            s1.pop();
          }
  ```

```
        int x = s2.top();
        //s2.pop();
        while(!s2.empty()){
          s1.push(s2.top());
          s2.pop();
        }
        return x;
      }

      bool empty() {
        return s1.size() == 0;
      }
};
```

-
  ```
  // smart solution

  class MyQueue {
    stack<int> s1;
    stack<int> s2;
  public:
      MyQueue() {

      }

      void push(int x) {
          if(s1.size() == 0){
            s1.push(x);
            return;
          }

          while(!s1.empty()){
            s2.push(s1.top());
            s1.pop();
          }
          s1.push(x);

          while(!s2.empty()){
            s1.push(s2.top());
            s2.pop();
          }

      }

      int pop() {
          int x = s1.top();
          s1.pop();
          return x;
      }

      int peek() {
          return s1.top();
  ```

```
    }

    bool empty() {
        return s1.size() == 0;
    }
};
```

225. Implement stack using queues

- Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

  Implement the `MyStack` class:

  - `void push(int x)` Pushes element x to the top of the stack.
  - `int pop()` Removes the element on the top of the stack and returns it.
  - `int top()` Returns the element on the top of the stack.
  - `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

-
  ```
  class MyStack {
      queue<int> q1;
  public:
      MyStack() {

      }

      void push(int x) {
        if(q1.empty()){
          q1.push(x);
          return;
        }

        q1.push(x);
        int Qsize = q1.size() -1;
        for(int i = 0; i <Qsize;i++){
          int x = q1.front(); q1.pop();
          q1.push(x);
        }

      }

      int pop() {
          int x = q1.front();
          q1.pop();
          return x;
      }

      int top() {
         return q1.front();
      }

      bool empty() {
  ```

```
        return q1.size() == 0;
    }
};
```

-