Algorithms

Big O Notation

- General rules
  - Different steps get added
  - Ignore Constants
    1. `5n->o(n)`
  - Different inputs => different variables
    - ```
      int intersectionSize(int ArrayA, int ArrayB){
          int count = 0;
          for a in array{
              for b in arrayB{
                  if a == b {
                      count = count + 1;
                  }
              }
          }
      }
      ```
    - `O(a*b)`
  - Drop non dominant terms
    - `O(n + n^2) = O(n^2)`
    -

704. Binary search

```
// recursive
class Solution {
public:

    int binarySearch(vector<int> v, int start, int end, int target){
        if(start <= end){
          int mid = start + (end -start)/2;

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

          if(target > v[mid]){
            return binarySearch(v, mid+1, end, target);
          }
```
```

```
            if(target < v[mid]){
              return binarySearch(v, start, mid-1, target);
            }
        }

        return -1;
    }

    int search(vector<int>& nums, int target) {
        return binarySearch(nums, 0, nums.size()-1, target);
    }
};
```

```
// iterative
class Solution {
public:

    int binarySearch(vector<int> v, int start, int end, int target){
        while(start <= end){
          int mid = start + (end -start)/2;

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

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

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

        return -1;
    }

    int search(vector<int>& nums, int target) {
        return binarySearch(nums, 0, nums.size()-1, target);
    }
};
```

278. First Bad Version

```
class Solution {
public:

    int firstBadVersion(int n) {
          int start = 1;
```

```
        int end = n ;

        while(start < end){
            int mid  = start + (end - start)/2;

            if(isBadVersion(mid)){
                end = mid;
            }else {
              start = mid+1;
            }
        }
        return start;
    }
};
```

35. Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

- for binary search, when low is equal to high that's where the target should have been if the list were sorted

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int low = 0;
        int high = nums.size()- 1;
        int midval = 0;
        while(low <= high){
            int mid = low + (high - low)/2;
            midval = mid;
            if(nums[mid] == target){
              return mid;
            } else if(nums[mid] > target){
              high = mid-1;
            } else if(nums[mid] < target){
              low = mid + 1;

              // when low == high thats where it should be
            } else if(low == high){
              return low;
            }
        }
        return low;
    }
};
```

977. Squares of a Sorted Array

Given an integer array `nums` sorted in **non-decreasing** order, return *an array of **the squares of each number** sorted in non-decreasing order*.

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
      map<int, int> map;
      vector<int> v;
        for(int i = 0; i < nums.size(); i++){
            int val = nums[i]*nums[i];
            cout << i << " "<< val << endl;
            map[val]++;

        }
        for(auto a : map){
          vector<int> v2(a.second, a.first);
          v.insert(v.end(), v2.begin(), v2.end());
        }
        return v;
    }
};
```

- Their solution
  - iterate over the negative part in reverse, and the positive part in the forward direction.
  - ```
    class Solution {
    public:
        vector<int> sortedSquares(vector<int>& nums) {
            int n = nums.size();

            vector<int> result(n);

            int left = 0;
            int right = n - 1;

            for (int i = n - 1; i >= 0; i--) {
                int square;
                if (abs(nums[left]) < abs(nums[right])) {
                    square = nums[right];
                    right--;
                } else {
                    square = nums[left];
                    left++;
                }
                result[i] = square * square;
            }
            return result;
    ```
    ```

```
    }
};
```

-

189. Rotate Array

Given an array, rotate the array to the right by `k` steps, where `k` is non-negative.

- cheated solution
- ```
  class Solution {
  public:
      void rotate(vector<int>& nums, int k) {
          std::rotate(nums.begin(), nums.end()-k, nums.end());
      }
  };
  ```
- ```
  class Solution {
  public:
      static void rotate(vector<int>& nums, int k) {
          vector<int> v(nums); // copy of nums

          for(int i = 0; i < nums.size(); i++){
            int findex = (i+k)%(nums.size());
            cout << findex << endl;
            nums[findex] = v[i];

          }

      }
  };
  ```

modulo operator (a= b%c) keeps the value between of a between zero and c-1, when b == c, value is zero

283. Move Zeroes

- Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

  Note that you must do this in-place without making a copy of the array.

  ```
  class Solution {
  public:
      static void moveZeroes(vector<int>& nums) {
        queue<int> q;
        int zeroCounter = 0;
        for(auto a: nums){
  ```

```
        if(a != 0){
          q.push(a);
        } else zeroCounter++;
      }

      for(int i = 0; i < nums.size(); i++){
        if(!q.empty()){
          nums[i] = q.front();
          q.pop();
        } else {
          if(zeroCounter != 0){
            nums[i] = 0;
          }
        }
      }

    }
};
```

167. Two Sum II - Input Array Is Sorted

- Given a **1-indexed** array of integers `numbers` that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where `1 <= index1 < index2 <= numbers.length`.

  Return *the indices of the two numbers,* `index1` *and* `index2`*, **added by one** as an integer array* `[index1, index2]` *of length 2.*

  The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

  Your solution must use only constant extra space.

```
class Solution {
public:
    static vector<int> twoSum(vector<int>& numbers, int target) {
        unordered_map<int,int> maps;
        vector<int> v;
        for(int i = 0; i < numbers.size(); i++){
            int seeking = target - numbers[i];
            cout << "seeking: " << seeking << endl;

            auto it = maps.find(numbers[i]);

            if(it == maps.end()){
               maps[seeking] = i+1;
            }

            else {
              v.push_back(it->second);
```

```
                v.push_back(i+1);
                cout << " found" ;
            }

        }

        return v;
    }
};
```

```
// their solution
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int low = 0;
        int high = numbers.size() - 1;
        while (low < high) {
            int sum = numbers[low] + numbers[high];

            if (sum == target) {
                return {low + 1, high + 1};
            } else if (sum < target) {
                ++low;
            } else {
                --high;
            }
        }
        // In case there is no solution, return {-1, -1}.
        return {-1, -1};
    }
};
```

344. Reverse String

- Write a function that reverses a string. The input string is given as an array of characters `s`.

  You must do this by modifying the input array [in-place](#) with `O(1)` extra memory.

Example 1:

```
Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
```

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int low = 0;
        int high = s.size()-1;
        while(low <= high){
          swap(s[low], s[high]);
          low++;
          high--;
        }
    }
};
```

557. Reverse Words in a String III

Given a string `s`, reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

```
Input: s = "Let's take LeetCode contest"
Output: "s'teL ekat edoCteeL tsetnoc"
```

```
class Solution {
public:

    void reverse(string & s){
      int low = 0;
      int high = s.size()-1;
      while(low <= high){
        swap(s[low++], s[high--]);
      }
    }

    string reverseWords(string s) {
        stringstream ss(s);
        stringstream s2;
        string word;
        while(ss >> word){
          //cout << word << endl;
          reverse(word);
          s2 << word;
          s2 << " ";
        }

        string rs = s2.str();
        rs.erase(rs.begin()+rs.size()-1);
        return rs;

    }
```

```
};
```

- `stringstream` can be used to separate things based on space
- `string.erase()` takes iterator as parameter and removes that entry or character

876. Middle of the Linked List

Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        int count = 0;
        ListNode * it = head;
        while(it != nullptr){
          count++;
          it = it->next;
        }

        int mid = 0;
        ListNode * it2 = head;
        while(mid <= (count/2 -1) && it2 != nullptr){
          it2 = it2->next;
          mid++;
        }
        return it2;
    }
};
```

19. Remove Nth Node from the end of the list

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

```
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        if(head == nullptr){
          return head;
        }

        int count = 0;
        ListNode * it = head;
        while(it != nullptr){
          count++;
          it = it->next;
        }

        if(count == 1){
          ListNode * i = head;
          delete i;
          return nullptr;
        }

        int removeIndex = count - n;
        it = head;
        for(int i = 0; i < removeIndex -1; i++){
          it = it->next;
        }

        ListNode * temp = it->next;
        it->next = temp->next;
        delete temp;

        return head;
    }
};
```

3. Longest Substring Without Repeating Characters (unsolved)

- Given a string `s`, find the length of the **longest substring** without repeating characters.
- ```
  class Solution {
  public:
      int lengthOfLongestSubstring(string s) {

        unordered_map<char, int> table;
        int high = 0;
        int low = 0;
        int max_count = INT_MIN;

        while(high < s.size()){
          if(table.count(s.at(high))){
            while(s.at(high) != s.at(low)){
  ```
  ```

```
            low++;
          }
          table.erase(s.at(high));
        }
        max_count = max(max_count, (high-low));
        table[s.at(high)]++;
        high++;
      }

      return max_count;
    }
};
```

- ```
  // their solution
  class Solution {
  public:
      int lengthOfLongestSubstring(string s) {
          // we will store a senitel value of -1 to simulate 'null'/'None' in C++
          vector<int> chars(128, -1);

          int left = 0;
          int right = 0;

          int res = 0;
          while (right < s.length()) {
              char r = s[right];

              int index = chars[r];
              if (index != -1 and index >= left and index < right) {
                  left = index + 1;
              }
              res = max(res, right - left + 1);

              chars[r] = right;
              right++;
          }
          return res;
      }
  };
  ```

-

567. Permutation in string

- Given two strings `s1` and `s2`, return `true` *if* `s2` *contains a permutation of* `s1`*, or* `false` *otherwise.*

  In other words, return `true` if one of `s1`'s permutations is the substring of `s2`
  ```

permutation

- A permutation is an arrangement of objects in a definite order. The members or elements of sets are arranged here in a sequence or linear order. For example, **the permutation of set A={1,6} is 2, such as {1,6}, {6,1}**.
- there are six permutations of the set {1, 2, 3}, namely
- (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1).
- These are all the possible orderings of this three-element set.
- The number of permutations of *n* distinct objects is *n* factorial, usually written as *n*!, which means the product of all positive integers less than or equal to *n*.

combination

- a **combination** is a selection of items from a set that has distinct members, such that the order of selection does not matter unlike permutation
- given three fruits, say an apple, an orange and a pear, there are three combinations of two that can be drawn from this set: an apple and a pear; an apple and an orange; or a pear and an orange.

```
// my wrong solution
class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        unordered_map<char, int> table;

        for(auto s: s1){
          table[s]++;
        }

        bool start = false;
        int count = s1.size();
        for(auto s: s2){
          auto it = table.find(s);
          if(it != table.end() && it->second > 0){
              count--;

          }else {
            count = s1.size();
          }

          if(count == 0){
            return true;
          }

        }

        return false;

    }
};
```

733. Flood Fill

- An image is represented by an `m x n` integer grid `image` where `image[i][j]` represents the pixel value of the image.
-

4 directionally

- 4-connected pixels are neighbors to every pixel that touches one of their edges. These pixels are connected horizontally and vertically. In terms of pixel coordinates, every pixel that has the coordinates is connected
  - (x+- 1, y)
  - (x, y+-1)
- to the pixel at (x, y)
-

```
class Solution {
public:

    void fill(vector<vector<int>>& image, int sr, int sc, int newColor, int oldColor){
      if(sr < 0 || sr >= image.size() || sc < 0 || sc >= image[sr].size()){
        return;
      }

      if(image[sr][sc] == newColor || image[sr][sc] != oldColor ) return;

      image[sr][sc] = newColor;

      fill(image, sr-1, sc, newColor, oldColor);
      fill(image, sr+1, sc, newColor, oldColor);
      fill(image, sr, sc+1, newColor, oldColor);
      fill(image, sr, sc-1, newColor, oldColor);


    }

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)
{
        if(newColor == image[sr][sc]) return image;

        fill(image, sr, sc, newColor, image[sr][sc]);
        return image;
    }
};
```

- uses depth first traversal to navigate to top right boom left. if the value already exist dont do anything else change the color
- flood fill mean iteratively/recursively fill horizontally or vertically connected pixels

695. Max Area of Island

- You are given an `m x n` binary matrix `grid`. An island is a group of `1`'s (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

  The **area** of an island is the number of cells with a value `1` in the island.

  Return *the maximum **area** of an island in* `grid`. If there is no island, return `0`

```
class Solution {
public:
    int count = 0;
    vector<vector<int>> graph;
    vector<vector<bool>> visited;

    int maxAread(int x, int y){

        if(x < 0 || x >= graph.size() || y < 0 || y >= graph[x].size()){
          return 0;
        }

        if(visited[x][y] || graph[x][y] == 0){
          return 0;
        }

        visited[x][y] = true;

        return 1 + maxAread(x+1,y) + maxAread(x-1, y) + maxAread(x,y-1) + maxAread(x, y+1);

    }

    int maxAreaOfIsland(vector<vector<int>>& grid) {
        graph = grid;
        vector<vector<bool>> v(grid.size(), vector<bool>(grid[0].size()));
        visited = v;
        int ans = 0;

        for(int r = 0; r < graph.size(); r++){
          for(int c = 0; c < graph[r].size(); c++){
            ans = max(maxAread(r, c), ans);
          }
        }

        return ans;
    }
};
```

```
  vector<vector<int>> v = {
     {0,0,1,0,0,0,0,1,0,0,0,0,0},
```

```
    {0,0,0,0,0,0,0,1,1,1,0,0,0},
    {0,1,1,0,1,0,0,0,0,0,0,0,0},
    {0,1,0,0,1,1,0,0,1,0,1,0,0},
    {0,1,0,0,1,1,0,0,1,1,1,0,0},
    {0,0,0,0,0,0,0,0,0,0,1,0,0},
    {0,0,0,0,0,0,0,1,1,1,0,0,0},
    {0,0,0,0,0,0,0,1,1,0,0,0,0}
  };

  Solution s;
  cout <<s.maxAreaOfIsland(v);
// 6
```

617. Merge Two Binary Trees

- You are given two binary trees `root1` and `root2`.

  Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

  Return *the merged tree*.

  Note: The merging process must start from the root nodes of both trees.

```
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
      if(root1 == nullptr){
        return root2;
      }

      if(root2 == nullptr){
        return root1;
      }

      root1->val += root2->val;

      root1->left = mergeTrees(root1->left, root2->left);
      root1->right = mergeTrees(root1->right, root2->right);

      return root1;

    }
};
```

116. Populating Next Right Pointers in Each Node

- A perfect binary tree is **a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level**.
- Perfect Binary Tree. All the internal nodes have a degree of 2.

```
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

class Solution {
public:
    Node* connect(Node* root) {
        queue<Node *> q;
        q.push(root);

        if(root == NULL)
        return root;

        while(!q.empty()){
          int Qsize = q.size();
          Node * prev = NULL;
          for(int i = 0; i < Qsize; i++){

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

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

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

            if(prev != NULL){
              prev->next = current;
            }
            prev = current;

          }
        }
```

```
        return root;
    }
};
```

542. 01 Matrix

Given an `m x n` binary matrix `mat`, return *the distance of the nearest* `0` *for each cell*.

The distance between two adjacent cells is `1`.