80. Remove Duplicates from Sorted Array II

```
#include <vector>
using namespace std;

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int insert_position = 0;

        for (int num : nums) {
            // If insert_position is less than 2, we add the element unconditionally.
            // Otherwise, check if the current element `num` is different from
            // the element at `insert_position - 2` to ensure at most two duplicates.
            if (insert_position < 2 || num != nums[insert_position - 2]) {
                nums[insert_position] = num;
                insert_position++;
            }
        }

        // `insert_position` is the new length of the modified array.
        return insert_position;
    }
};
```

how this brute force works ?

---

189. Rotate Array

```
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        int n = nums.size();
        k = k % n;  // Normalize k if it's greater than n
```

.

```
        // Reverse the entire array
        reverse(nums.begin(), nums.end());

        // Reverse the first k elements
        reverse(nums.begin(), nums.begin() + k);

        // Reverse the rest of the elements
        reverse(nums.begin() + k, nums.end());
    }
};
```

---

122. Best Time to Buy and Sell Stock II

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int max_profit = 0;
        // if any kind of thing like maximun or minimum then we are just focusing on the one side of the solution (ie focusing on only max here)
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] > prices[i - 1]) {
                max_profit += prices[i] - prices[i - 1];
            }
        }
        return max_profit;
    }
};
```

```
for(num:nums0{
if(nums[i]>nums){

Check whether the
tracking indices values would be workout for this

}
```

---

55. Jump Game

```
class Solution {
public:
```

```
    bool canJump(vector<int>& nums) {
        int max_reachable = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (i > max_reachable) return false;
            max_reachable = max(max_reachable, i + nums[i]);
        }
        return true;
    }
};
```

==========

45. Jump Game II

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int jumps = 0, current_end = 0, farthest = 0;
        for (int i = 0; i < nums.size() - 1; i++) {
            farthest = max(farthest, i + nums[i]);
            if (i == current_end) {
                jumps++;
                current_end = farthest;
            }
        }
        return jumps;
    }
};
```

// Maximum finding for the jumps we move

// Moving to the lesser minimum of that

// Counting the number of moves or distance to the end form my pos need for this will be for avoinding the considering the cost more than need

//

---

274. H-Index

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        sort(citations.begin(), citations.end(), greater<int>());
        int h = 0;
        for (int i = 0; i < citations.size(); i++) {
            if (citations[i] >= i + 1) { //trick cannot Idebtify
                h = i + 1;
            } else {
                break;
            }
        }
        return h;
    }
};
```

---

380. Insert Delete GetRandom O(1)

```
class RandomizedSet {
    unordered_map<int, int> val_to_index;
    vector<int> values;

public:
    RandomizedSet() {}

    bool insert(int val) {
```

```
        if (val_to_index.count(val)) {
            return false;
        }
        val_to_index[val] = values.size();
        values.push_back(val);
        return true;
    }

    bool remove(int val) {
        if (!val_to_index.count(val)) {
            return false;
        }
        int index = val_to_index[val];
        int last_val = values.back();
        values[index] = last_val;
        val_to_index[last_val] = index;
        values.pop_back();
        val_to_index.erase(val);
        return true;
    }

    int getRandom() {
        int random_index = rand() % values.size();
        return values[random_index];
    }
};
```

==========

238. Product of Array Except Self

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n = nums.size();
        vector<int> answer(n, 1);  // Initialize answer array with 1s

        // First pass: Calculate left products for each element
        int left_product = 1;
        for (int i = 0; i < n; ++i) {
            answer[i] = left_product;
            left_product *= nums[i];
        }

        // Second pass: Calculate right products for each element and multiply
        int right_product = 1;
        for (int i = n - 1; i >= 0; --i) {
            answer[i] *= right_product;
            right_product *= nums[i];
        }

        return answer;
    }
};
```

---

134. Gas Station

```
class Solution {
public:
// one of the answer is in dynamic way
//
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
```

```
        int total_gas = 0, total_cost = 0;
        int current_gas = 0;
        int start_station = 0;

        for (int i = 0; i < gas.size(); ++i) {
            total_gas += gas[i];
            total_cost += cost[i];
            current_gas += gas[i] - cost[i];

            if (current_gas < 0) {
                start_station = i + 1;
                current_gas = 0;
            }
        }

        return (total_gas >= total_cost) ? start_station : -1;
    }
};
```

---

12. Integer to Roman

```
class Solution {
public:
    std::string intToRoman(int num) {
        std::vector<std::pair<int, std::string>> roman_numerals = {
            {1000, "M"}, {900, "CM"}, {500, "D"}, {400, "CD"},
            {100, "C"}, {90, "XC"}, {50, "L"}, {40, "XL"},
            {10, "X"}, {9, "IX"}, {5, "V"}, {4, "IV"},
            {1, "I"}
        };

        std::string result;
```

```
        for (const auto& [value, symbol] : roman_numerals) {
            while (num >= value) {
                result += symbol;
                num -= value;
            }
        }

        return result;
    }
};
```

---

151. Reverse Words in a String

```
class Solution {
public:
    std::string reverseWords(std::string s) {

  //edge
        // Step 1: Trim leading and trailing spaces
        //what is size_t

        size_t start = s.find_first_not_of(" ");
        size_t end = s.find_last_not_of(" ");

        if (start == std::string::npos) return ""; // If the string is empty or contains only spaces

        s = s.substr(start, end - start + 1); // Trimmed string
//main
 //splitting
        // Step 2: Split the string into words
        std::istringstream iss(s); // iss?
```

```
        std::vector<std::string> words;
        std::string word;

//pushing_back
    //what does that >> symbol do in general and here

        while (iss >> word) {
            words.push_back(word);
        }

//reversing

        // Step 3: Reverse the list of words
        std::reverse(words.begin(), words.end());

        // Step 4: Join the words with a single space
        // here only they were adding spaces
        std::string result;
        for (size_t i = 0; i < words.size(); ++i) {
            result += words[i];
            if (i < words.size() - 1) {
                result += " "; // Add space between words
            }
        }

        for(int i=0;i>words.size();i++){
            result +=words[i];
            if(i<words.size()-1){
                result += " ";
            }
        }
```

```
        return result;
    }
};
```

---

6. Zigzag Conversion

```
class Solution {
public:
    std::string convert(std::string s, int numRows) {
        // If numRows is 1 or greater than or equal to the length of the string
        if (numRows == 1 || numRows >= s.length()) {
            return s;
        }

        // 1)Create a vector to hold the rows
        std::vector<std::string> rows(numRows);
       //2
        int currentRow = 0;      // Track the current row
       //3
        bool goingDown = false;  // Direction flag
        // 4 Concatenate all rows into the result string
        std::string result;

        // Iterate through the characters of the string
        for (char c : s) {
            rows[currentRow] += c; // Add the character to the current row

            // Change direction if we hit the top or bottom row
            if (currentRow == 0) {
                goingDown = true;
            } else if (currentRow == numRows - 1) {
```

```
            goingDown = false;
        }

        // Update current row based on direction
        currentRow += goingDown ? 1 : -1;
    }

    for (const std::string& row : rows) {
        result += row;
    }
        return result;
  }
};
```

---

167. Two Sum II - Input Array Is Sorted

```
class Solution {
public:
  vector<int> twoSum(vector<int>& numbers, int target) {
    int i = 0;              // Start pointer at the beginning
    int j = numbers.size() - 1; // End pointer at the end
    while (i < j) {
      int sum = numbers[i] + numbers[j];
      if (sum == target) {
        return {i + 1, j + 1}; // Return 1-based indices
      } else if (sum < target) {
        i++; // Move the left pointer right to increase the sum
      } else {
        j--; // Move the right pointer left to decrease the sum
      }
```

```
        }
        return {}; // Should never reach here because there's always exactly one solution
    }
};
```

---

11. Container With Most Water

```
class Solution {
public:
    int maxArea(std::vector<int>& height) {
        int left = 0;               // Initialize the left pointer
        int right = height.size() - 1; // Initialize the right pointer
        int maxArea = 0;            // Variable to store the maximum area

        // Loop until the two pointers meet
        while (left < right) {

            // Calculate the width and height
            int width = right - left;
            int h = std::min(height[left], height[right]);

            // Calculate the area
            int area = width * h;
            maxArea = std::max(maxArea, area); // Update maxArea if current area is larger

            // Move the pointer pointing to the shorter line
            if (height[left] < height[right]) {
                left++; // Move the left pointer to the right
            } else {
                right--; // Move the right pointer to the left
```

```
        }
      }

      return maxArea; // Return the maximum area found
    }
};
```

---

15. 3Sum

```
class Solution {
public:
    std::vector<std::vector<int>> threeSum(std::vector<int>& nums) {
        std::vector<std::vector<int>> result;
        std::sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size(); ++i) {
            // the element do not need to  be adjacent right ?

            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }

            int left = i + 1;
            int right = nums.size() - 1;

            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {
```

```
                result.push_back({nums[i], nums[left], nums[right]});

                // Move the left pointer to the right, skipping duplicates
                // why we twice removing the duplicates,
                while (left < right && nums[left] == nums[left + 1]) {
                    left++;
                }
                while (left < right && nums[right] == nums[right - 1]) {
                    right--;
                }

                left++;
                right--;
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }
    return result; // Return the list of triplets
  }
};
```

---

36. Valid Sudoku

```
class Solution {
public:
  bool isValidSudoku(vector<vector<char>>& board) {
    vector<unordered_set<char>> rows(9), cols(9), boxes(9);
```

```
        for (int row = 0; row < 9; ++row) {
            for (int col = 0; col < 9; ++col) {
                char num = board[row][col];

                if (num == '.') continue;  // Skip empty cells

                int boxIndex = (row / 3) * 3 + (col / 3);

                // Check if the current number is already in the corresponding row, column, or box
                if (rows[row].count(num) || cols[col].count(num) || boxes[boxIndex].count(num)) {
                    return false;
                }

                // Add the number to the respective row, column, and box sets
                rows[row].insert(num);
                cols[col].insert(num);
                boxes[boxIndex].insert(num);
                // I cpuld not imagine this working of above three lines
            }
        }

        return true;  // All checks passed, the board is valid
    }
};
```

---

54. Spiral Matrix

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> result;
```

```
    if (matrix.empty()) return result;
    int top = 0, bottom = matrix.size() - 1;
    int left = 0, right = matrix[0].size() - 1;

    while (top <= bottom && left <= right) {
        // Traverse from left to right across the top boundary
        for (int i = left; i <= right; ++i) {
            result.push_back(matrix[top][i]);
        }
        ++top;
`
    // Traverse from top to bottom along the right boundary
    for (int i = top; i <= bottom; ++i) {
        result.push_back(matrix[i][right]);
        }
        --right;

        // Traverse from right to left across the bottom boundary, if within bounds
        if (top <= bottom) {
            for (int i = right; i >= left; --i) {
                result.push_back(matrix[bottom][i]);
            }
            --bottom;
        }

        // Traverse from bottom to top along the left boundary, if within bounds
        if (left <= right) {
            for (int i = bottom; i >= top; --i) {
                result.push_back(matrix[i][left]);
            }
```

```
            ++left;
        }
    }

    return result;
  }
};
```

---

48. Rotate Image

```
class Solution {
public:
  void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();

    for (int i = 0; i < n; ++i) {
      for (int j = i + 1; j < n; ++j) {
        swap(matrix[i][j], matrix[j][i]);
      }
      reverse(matrix[i].begin(), matrix[i].end());
    }


  }
};
```

---

73. Set Matrix Zeroes

```
class Solution {
public:
  void setZeroes(vector<vector<int>>& matrix) {
```

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

    // Flags to check if the first row and first column should be zeroed
    bool isFirstRowZero = false;
    bool isFirstColZero = false;

    // Check if the first row contains any zeros
    for (int j = 0; j < n; j++) {
        if (matrix[0][j] == 0) {
            isFirstRowZero = true;
            break;
        }
    }

    // Check if the first column contains any zeros
    for (int i = 0; i < m; i++) {
        if (matrix[i][0] == 0) {
            isFirstColZero = true;
            break;
        }
    }

    // Use the first row and column as markers
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0; // Mark the row
                matrix[0][j] = 0;  // Mark the column
            }
        }
```

```
        }

        // Set matrix elements to zero based on markers
        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;
                }
            }
        }

        // Zero out the first row if needed
        if (isFirstRowZero) {
            for (int j = 0; j < n; j++) {
                matrix[0][j] = 0;
            }
        }

        // Zero out the first column if needed
        if (isFirstColZero) {
            for (int i = 0; i < m; i++) {
                matrix[i][0] = 0;
            }
        }
    }
};
```

---

289. Game of Life

```
class Solution {
```

```
public:
    void gameOfLife(vector<vector<int>>& board) {
        int m = board.size();
        int n = board[0].size();

        // Helper function to count live neighbors
        auto countLiveNeighbors = [&](int i, int j) {
            int liveNeighbors = 0;
            for (int x = max(i - 1, 0); x <= min(i + 1, m - 1); ++x) {
                for (int y = max(j - 1, 0); y <= min(j + 1, n - 1); ++y) {
                    if (x == i && y == j) continue; // skip the cell itself
                    if (board[x][y] == 1 || board[x][y] == 2) {
                        liveNeighbors++;
                    }
                }
            }
            return liveNeighbors;
        };

        // First pass: Apply the rules using temporary states
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                int liveNeighbors = countLiveNeighbors(i, j);

                if (board[i][j] == 1) {
                    // Apply Rule 1, 2, or 3
                    if (liveNeighbors < 2 || liveNeighbors > 3) {
                        board[i][j] = 2; // Mark live cell as dead
                    }
                } else if (board[i][j] == 0) {
                    // Apply Rule 4
```

```
                if (liveNeighbors == 3) {
                    board[i][j] = -1; // Mark dead cell as live
                }
            }
        }
    }

    // Second pass: Finalize the board by updating temporary states
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (board[i][j] == 2) {
                board[i][j] = 0; // Was live, now dead
            } else if (board[i][j] == -1) {
                board[i][j] = 1; // Was dead, now live
            }
        }
    }
  }
};
```

---

49. Group Anagrams

```
class Solution {
public:
   vector<vector<string>> groupAnagrams(vector<string>& strs) {
      unordered_map<string, vector<string>> anagramMap;

      for (string s : strs) {

         string sortedStr = s;
```

```
            sort(sortedStr.begin(), sortedStr.end());


            anagramMap[sortedStr].push_back(s);
        }

        vector<vector<string>> result;
        for (auto& entry : anagramMap) {
            result.push_back(entry.second);
        }

        return result;
    }
};
```

---

56. Merge Intervals

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        // Edge case: if the intervals list is empty, return an empty list
        if (intervals.empty()) return {};

        // Step 1: Sort intervals by their start times
        sort(intervals.begin(), intervals.end());

        // Step 2: Create a result vector to store merged intervals
        vector<vector<int>> merged;

        // Step 3: Initialize the first interval as the starting point
        merged.push_back(intervals[0]);
```

```
        // Step 4: Iterate through each interval and merge if overlapping
        for (int i = 1; i < intervals.size(); i++) {
            // Get the last interval in the merged list
            vector<int>& lastMerged = merged.back();

            // If the current interval overlaps with the last merged interval, merge them
            if (intervals[i][0] <= lastMerged[1]) {
                // Update the end time of the last merged interval
                lastMerged[1] = max(lastMerged[1], intervals[i][1]);
            } else {
                // Otherwise, there is no overlap, so add the current interval as a new interval
                merged.push_back(intervals[i]);
            }
        }

        // Step 5: Return the merged intervals
        return merged;
    }
};
```

---

57. Insert Interval

```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
        vector<vector<int>> result;
        int i = 0;

        // Step 1: Add all intervals before the new interval that do not overlap
```

```
        while (i < intervals.size() && intervals[i][1] < newInterval[0]) {
            result.push_back(intervals[i]);
            i++;
        }

        // Step 2: Merge all overlapping intervals with the new interval
        while (i < intervals.size() && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = min(newInterval[0], intervals[i][0]);
            newInterval[1] = max(newInterval[1], intervals[i][1]);
            i++;
        }
        result.push_back(newInterval); // Add the merged newInterval

        // Step 3: Add all intervals after the new interval that do not overlap
        while (i < intervals.size()) {
            result.push_back(intervals[i]);
            i++;
        }

        return result;
    }
};
```

---

452. Minimum Number of Arrows to Burst Balloons

```
class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.empty()) return 0;
```

```
        // Sort balloons by their end points (xend)
        sort(points.begin(), points.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[1] < b[1];
        });

        int arrows = 1;  // We need at least one arrow for the first balloon
        int currentEnd = points[0][1];

        for (const auto& balloon : points) {
            if (balloon[0] > currentEnd) {
                // This balloon starts after the current end, so we need a new arrow
                arrows++;
                currentEnd = balloon[1];
            }
        }

        return arrows;
    }
};
```

---

71. Simplify Path

```
class Solution {
public:
    string simplifyPath(string path) {
        vector<string> stack;
        stringstream ss(path);
        string component;

        // Split path by '/' using stringstream
        while (getline(ss, component, '/')) {
```

```
            if (component == "" || component == ".") {
                // Skip empty or current directory components
                continue;
            } else if (component == "..") {
                // Go up one level if possible (if the stack is not empty)
                if (!stack.empty()) {
                    stack.pop_back();
                }
            } else {
                // Add valid directory name to the stack
                stack.push_back(component);
            }
        }

        // Construct the simplified path
        string simplifiedPath = "/";
        for (size_t i = 0; i < stack.size(); ++i) {
            simplifiedPath += stack[i];
            if (i < stack.size() - 1) {
                simplifiedPath += "/";
            }
        }

        return simplifiedPath;
    }
};
```

---

155. Min Stack

```
class MinStack {
public:
    // Declare two stacks
```

```
    stack<int> stack, minStack;

    // Constructor initializes the stacks

    //why here they inticalized a minstack constructot here
    MinStack() {
    }

    // Push the value onto the main stack
    void push(int val) {
        stack.push(val);
        // Push the value onto the minStack if minStack is empty or the value is smaller or equal to the current minimum

        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }

    // Pop the top value from the stack

    void pop() {
        if (!stack.empty()) {
            int topValue = stack.top();
            stack.pop();

            // If the popped value is the current minimum, pop it from minStack as well
            if (topValue == minStack.top()) {
                minStack.pop();
            }
        }
```

```
    }

    // Get the top value of the stack
    int top() {
        if (!stack.empty()) {
            return stack.top();
        }
        return -1; // Return -1 or some error value if stack is empty
    }

    // Retrieve the minimum value from the stack
    int getMin() {
        if (!minStack.empty()) {
            return minStack.top();
        }
        return -1; // Return -1 or some error value if stack is empty
    }
};
```

---

150. Evaluate Reverse Polish Notation

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> stk;  // Stack to hold operands

        for (string& token : tokens) {
            if (token == "+" || token == "-" || token == "*" || token == "/") {
                // Pop two operands from the stack
                int b = stk.top(); stk.pop();
                int a = stk.top(); stk.pop();
```

```
            // Perform the operation and push the result back onto the stack
            if (token == "+") {
                stk.push(a + b);
            } else if (token == "-") {
                stk.push(a - b);
            } else if (token == "*") {
                stk.push(a * b);
            } else if (token == "/") {
                // Handle division, truncating toward zero
                stk.push(a / b);
            }
        } else {
            // Convert the token to an integer and push it onto the stack
            stk.push(stoi(token));
        }
    }

    // The result is the only element left in the stack
    return stk.top();
  }
};
```
