**Cpp Practice on Leet Code:**

Question-1

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. Example 1: Input: nums = [2,7,11,15], target = 9 Output: [0,1] Explanation: Because nums[0] + nums[1] == 9, we return [0, 1]. Example 2: Input: nums = [3,2,4], target = 6 Output: [1,2] Example 3: Input: nums = [3,3], target = 6 Output: [0,1] Constraints: 2 <= nums.length <= 104 -109 <= nums[i] <= 109 -109 <= target <= 109 Only one valid answer exists.

Solution-

#include <vector>

#include <unordered\_map>

using namespace std;

class Solution {

public:

vector<int> twoSum(vector<int>& nums, int target) {

unordered\_map<int, int> mp; // value -> index

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

int complement = target - nums[i];

if (mp.find(complement) != mp.end()) {

return {mp[complement], i};

}

mp[nums[i]] = i;

}

return {}; // will never reach here because problem guarantees a solution

}

};

**Question-2**

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. You may assume the two numbers do not contain any leading zero, except the number 0 itself. Example 1: Input: l1 = [2,4,3], l2 = [5,6,4] Output: [7,0,8] Explanation: 342 + 465 = 807. Example 2: Input: l1 = [0], l2 = [0] Output: [0] Example 3: Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9] Output: [8,9,9,9,0,0,0,1] Constraints: The number of nodes in each linked list is in the range [1, 100]. 0 <= Node.val <= 9 It is guaranteed that the list represents a number that does not have leading zeros.

**Solution-**

class Solution {

public:

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

ListNode\* dummy = new ListNode(0);

ListNode\* current = dummy;

int carry = 0;

while (l1 != nullptr || l2 != nullptr || carry != 0) {

int x = (l1 != nullptr) ? l1->val : 0;

int y = (l2 != nullptr) ? l2->val : 0;

int sum = x + y + carry;

carry = sum / 10;

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

current = current->next;

if (l1 != nullptr) l1 = l1->next;

if (l2 != nullptr) l2 = l2->next;

}

return dummy->next;

}

};

**Question-3**

Given a string s, find the length of the longest substring without duplicate characters. Example 1: Input: s = "abcabcbb" Output: 3 Explanation: The answer is "abc", with the length of 3. Example 2: Input: s = "bbbbb" Output: 1 Explanation: The answer is "b", with the length of 1. Example 3: Input: s = "pwwkew" Output: 3 Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring. Constraints: 0 <= s.length <= 5 \* 104 s consists of English letters, digits, symbols and spaces.

**Solution-**

#include <unordered\_map>

#include <string>

using namespace std;

class Solution {

public:

int lengthOfLongestSubstring(string s) {

unordered\_map<char, int> mp; // character -> last index

int left = 0, maxLen = 0;

for (int right = 0; right < s.size(); right++) {

if (mp.find(s[right]) != mp.end() && mp[s[right]] >= left) {

left = mp[s[right]] + 1; // move left pointer

}

mp[s[right]] = right; // update last index

maxLen = max(maxLen, right - left + 1);

}

return maxLen;

}

};

**Question-4**

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)). Example 1: Input: nums1 = [1,3], nums2 = [2] Output: 2.00000 Explanation: merged array = [1,2,3] and median is 2. Example 2: Input: nums1 = [1,2], nums2 = [3,4] Output: 2.50000 Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5. Constraints: nums1.length == m nums2.length == n 0 <= m <= 1000 0 <= n <= 1000 1 <= m + n <= 2000 -106 <= nums1[i], nums2[i] <= 106

**Solution-**

#include <vector>

#include <climits>

using namespace std;

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

if (nums1.size() > nums2.size()) {

return findMedianSortedArrays(nums2, nums1); // ensure nums1 is smaller

}

int m = nums1.size(), n = nums2.size();

int total = m + n;

int half = (total + 1) / 2;

int left = 0, right = m;

while (left <= right) {

int i = (left + right) / 2;

int j = half - i;

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;

if (left1 <= right2 && left2 <= right1) {

if (total % 2 == 0) {

return (max(left1, left2) + min(right1, right2)) / 2.0;

} else {

return max(left1, left2);

}

} else if (left1 > right2) {

right = i - 1;

} else {

left = i + 1;

}

}

return 0.0; // should never reach here

}

};

**Question-5**

Given a string s, return the longest palindromic substring in s. Example 1: Input: s = "babad" Output: "bab" Explanation: "aba" is also a valid answer. Example 2: Input: s = "cbbd" Output: "bb" Constraints: 1 <= s.length <= 1000 s consist of only digits and English letters.

**Solution-**

#include <string>

using namespace std;

class Solution {

public:

string longestPalindrome(string s) {

if (s.empty()) return "";

int start = 0, maxLen = 1;

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

// Odd length palindrome

expandAroundCenter(s, i, i, start, maxLen);

// Even length palindrome

expandAroundCenter(s, i, i + 1, start, maxLen);

}

return s.substr(start, maxLen);

}

private:

void expandAroundCenter(const string& s, int left, int right, int& start, int& maxLen) {

while (left >= 0 && right < s.size() && s[left] == s[right]) {

if (right - left + 1 > maxLen) {

start = left;

maxLen = right - left + 1;

}

left--;

right++;

}

}

};

**Question-6**

You are given a 2D array points of size n x 2 representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi]. We define the right direction as positive x-axis (increasing x-coordinate) and the left direction as negative x-axis (decreasing x-coordinate). Similarly, we define the up direction as positive y-axis (increasing y-coordinate) and the down direction as negative y-axis (decreasing y-coordinate) You have to place n people, including Alice and Bob, at these points such that there is exactly one person at every point. Alice wants to be alone with Bob, so Alice will build a rectangular fence with Alice's position as the upper left corner and Bob's position as the lower right corner of the fence (Note that the fence might not enclose any area, i.e. it can be a line). If any person other than Alice and Bob is either inside the fence or on the fence, Alice will be sad. Return the number of pairs of points where you can place Alice and Bob, such that Alice does not become sad on building the fence. Note that Alice can only build a fence with Alice's position as the upper left corner, and Bob's position as the lower right corner. For example, Alice cannot build either of the fences in the picture below with four corners (1, 1), (1, 3), (3, 1), and (3, 3), because: With Alice at (3, 3) and Bob at (1, 1), Alice's position is not the upper left corner and Bob's position is not the lower right corner of the fence. With Alice at (1, 3) and Bob at (1, 1), Bob's position is not the lower right corner of the fence. Example 1: Input: points = [[1,1],[2,2],[3,3]] Output: 0 Explanation: There is no way to place Alice and Bob such that Alice can build a fence with Alice's position as the upper left corner and Bob's position as the lower right corner. Hence we return 0. Example 2: Input: points = [[6,2],[4,4],[2,6]] Output: 2 Explanation: There are two ways to place Alice and Bob such that Alice will not be sad: - Place Alice at (4, 4) and Bob at (6, 2). - Place Alice at (2, 6) and Bob at (4, 4). You cannot place Alice at (2, 6) and Bob at (6, 2) because the person at (4, 4) will be inside the fence. Example 3: Input: points = [[3,1],[1,3],[1,1]] Output: 2 Explanation: There are two ways to place Alice and Bob such that Alice will not be sad: - Place Alice at (1, 1) and Bob at (3, 1). - Place Alice at (1, 3) and Bob at (1, 1). You cannot place Alice at (1, 3) and Bob at (3, 1) because the person at (1, 1) will be on the fence. Note that it does not matter if the fence encloses any area, the first and second fences in the image are valid. Constraints: 2 <= n <= 1000 points[i].length == 2 -109 <= points[i][0], points[i][1] <= 109 All points[i] are distinct.

**Solution-**

#include <vector>

using namespace std;

class Solution {

public:

int numberOfPairs(vector<vector<int>>& points) {

int n = points.size();

int ans = 0;

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

int xa = points[i][0];

int ya = points[i][1];

for (int j = 0; j < n; j++) { // Bob

if (i == j) continue;

int xb = points[j][0];

int yb = points[j][1];

// Alice must be upper-left, Bob lower-right

if (xa > xb || ya < yb) continue;

bool ok = true;

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

if (k == i || k == j) continue;

int x = points[k][0];

int y = points[k][1];

// If point lies inside/on rectangle → invalid

if (xa <= x && x <= xb && yb <= y && y <= ya) {

ok = false;

break;

}

}

if (ok) ans++;

}

}

return ans;

}

};

**OR**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int numberOfPairs(vector<vector<int>>& points) {

int n = points.size();

// 1) Coordinate compression for x and y

vector<long long> xs, ys;

xs.reserve(n); ys.reserve(n);

for (auto &p : points) {

xs.push\_back(p[0]);

ys.push\_back(p[1]);

}

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

xs.erase(unique(xs.begin(), xs.end()), xs.end());

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

ys.erase(unique(ys.begin(), ys.end()), ys.end());

int X = (int)xs.size();

int Y = (int)ys.size();

// compressed coordinates for each point (1-based for prefix sums)

vector<int> cx(n), cy(n);

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

cx[i] = (int)(lower\_bound(xs.begin(), xs.end(), (long long)points[i][0]) - xs.begin()) + 1;

cy[i] = (int)(lower\_bound(ys.begin(), ys.end(), (long long)points[i][1]) - ys.begin()) + 1;

}

// 2) Build grid and 2D prefix sums

vector<vector<int>> grid(X + 1, vector<int>(Y + 1, 0));

for (int i = 0; i < n; i++) grid[cx[i]][cy[i]] = 1;

vector<vector<int>> ps(X + 1, vector<int>(Y + 1, 0));

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

for (int j = 1; j <= Y; j++) {

ps[i][j] = grid[i][j]

+ ps[i - 1][j]

+ ps[i][j - 1]

- ps[i - 1][j - 1];

}

}

auto rectSum = [&](int x1, int y1, int x2, int y2) -> int {

// assumes 1 <= x1 <= x2 <= X and 1 <= y1 <= y2 <= Y

return ps[x2][y2] - ps[x1 - 1][y2] - ps[x2][y1 - 1] + ps[x1 - 1][y1 - 1];

};

// 3) Count valid (Alice, Bob) ordered pairs

long long ans = 0;

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

for (int j = 0; j < n; j++) { // Bob

if (i == j) continue;

// Alice must be upper-left of Bob: xA <= xB and yA >= yB

if (points[i][0] <= points[j][0] && points[i][1] >= points[j][1]) {

int xA = cx[i], yA = cy[i];

int xB = cx[j], yB = cy[j];

// query inclusive rectangle [xA..xB] × [yB..yA]

int cnt = rectSum(min(xA, xB), min(yA, yB), max(xA, xB), max(yA, yB));

if (cnt == 2) ans++; // only Alice and Bob are inside/on the fence

}

}

}

return (int)ans;

}

};

**Question-7**

int dist1 = abs(x - z);

int dist2 = abs(y - z);

if (dist1 < dist2) return 1;

else if (dist2 < dist1) return 2;

else return 0;

**Solution-**

#include <cmath>

using namespace std;

class Solution {

public:

int findClosest(int x, int y, int z) {

int dist1 = abs(x - z);

int dist2 = abs(y - z);

if (dist1 < dist2) return 1;

else if (dist2 < dist1) return 2;

else return 0;

}

};

**Question-7**

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility) P A H N A P L S I I G Y I R And then read line by line: "PAHNAPLSIIGYIR" Write the code that will take a string and make this conversion given a number of rows: string convert(string s, int numRows); Example 1: Input: s = "PAYPALISHIRING", numRows = 3 Output: "PAHNAPLSIIGYIR" Example 2: Input: s = "PAYPALISHIRING", numRows = 4 Output: "PINALSIGYAHRPI" Explanation: P I N A L S I G Y A H R P I Example 3: Input: s = "A", numRows = 1 Output: "A" Constraints: 1 <= s.length <= 1000 s consists of English letters (lower-case and upper-case), ',' and '.'. 1 <= numRows <= 1000

**Solution-**

#include <string>

#include <vector>

using namespace std;

class Solution {

public:

string convert(string s, int numRows) {

if (numRows == 1 || numRows >= s.size()) return s;

vector<string> rows(min(numRows, (int)s.size()));

int curRow = 0;

bool goingDown = false;

for (char c : s) {

rows[curRow] += c;

if (curRow == 0 || curRow == numRows - 1) {

goingDown = !goingDown;

}

curRow += goingDown ? 1 : -1;

}

string result;

for (string &row : rows) result += row;

return result;

}

};

**Question-8**

Given a signed 32-bit integer x, return x*with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

**Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

**Example 1:**

**Input:** x = 123

**Output:** 321

**Example 2:**

**Input:** x = -123

**Output:** -321

**Example 3:**

**Input:** x = 120

**Output:** 21

**Constraints:**

* -231 <= x <= 231 - 1

**Solution-**

class Solution {

public:

int reverse(int x) {

int rev = 0;

while (x != 0) {

int digit = x % 10;

x /= 10;

// Check for overflow before multiplying

if (rev > INT\_MAX / 10 || (rev == INT\_MAX / 10 && digit > 7)) return 0;

if (rev < INT\_MIN / 10 || (rev == INT\_MIN / 10 && digit < -8)) return 0;

rev = rev \* 10 + digit;

}

return rev;

}

};

**Question-9**

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer.

The algorithm for myAtoi(string s) is as follows:

1. **Whitespace**: Ignore any leading whitespace (" ").
2. **Signedness**: Determine the sign by checking if the next character is '-' or '+', assuming positivity if neither present.
3. **Conversion**: Read the integer by skipping leading zeros until a non-digit character is encountered or the end of the string is reached. If no digits were read, then the result is 0.
4. **Rounding**: If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then round the integer to remain in the range. Specifically, integers less than -231 should be rounded to -231, and integers greater than 231 - 1 should be rounded to 231 - 1.

Return the integer as the final result.

**Example 1:**

**Input:** s = "42"

**Output:** 42

**Explanation:**

The underlined characters are what is read in and the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

**Example 2:**

**Input:** s = " -042"

**Output:** -42

**Explanation:**

Step 1: " -042" (leading whitespace is read and ignored)

^

Step 2: " -042" ('-' is read, so the result should be negative)

^

Step 3: " -042" ("042" is read in, leading zeros ignored in the result)

^

**Example 3:**

**Input:** s = "1337c0d3"

**Output:** 1337

**Explanation:**

Step 1: "1337c0d3" (no characters read because there is no leading whitespace)

^

Step 2: "1337c0d3" (no characters read because there is neither a '-' nor '+')

^

Step 3: "1337c0d3" ("1337" is read in; reading stops because the next character is a non-digit)

^

**Example 4:**

**Input:** s = "0-1"

**Output:** 0

**Explanation:**

Step 1: "0-1" (no characters read because there is no leading whitespace)

^

Step 2: "0-1" (no characters read because there is neither a '-' nor '+')

^

Step 3: "0-1" ("0" is read in; reading stops because the next character is a non-digit)

^

**Example 5:**

**Input:** s = "words and 987"

**Output:** 0

**Explanation:**

Reading stops at the first non-digit character 'w'.

**Constraints:**

* 0 <= s.length <= 200
* s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

**Solution-**

class Solution {

public:

int myAtoi(string s) {

int i = 0, n = s.size();

// Step 1: Skip leading whitespace

while (i < n && s[i] == ' ') {

i++;

}

// Step 2: Check sign

int sign = 1;

if (i < n && (s[i] == '+' || s[i] == '-')) {

sign = (s[i] == '-') ? -1 : 1;

i++;

}

// Step 3: Convert digits

long result = 0; // use long to detect overflow before returning

while (i < n && isdigit(s[i])) {

int digit = s[i] - '0';

result = result \* 10 + digit;

// Step 4: Handle overflow

if (sign == 1 && result > INT\_MAX) return INT\_MAX;

if (sign == -1 && -result < INT\_MIN) return INT\_MIN;

i++;

}

return (int)(sign \* result);

}

};

**Question-10**

Given an integer x, return true*if*x*is a****palindrome****, and*false*otherwise*.

**Example 1:**

**Input:** x = 121

**Output:** true

**Explanation:** 121 reads as 121 from left to right and from right to left.

**Example 2:**

**Input:** x = -121

**Output:** false

**Explanation:** From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

**Example 3:**

**Input:** x = 10

**Output:** false

**Explanation:** Reads 01 from right to left. Therefore it is not a palindrome.

**Constraints:**

* -231 <= x <= 231 - 1

**Follow up:** Could you solve it without converting the integer to a string?

**Solution-**

class Solution {

public:

bool isPalindrome(int x) {

// Step 1: Handle edge cases

if (x < 0 || (x % 10 == 0 && x != 0)) return false;

int reversedHalf = 0;

while (x > reversedHalf) {

int digit = x % 10;

reversedHalf = reversedHalf \* 10 + digit;

x /= 10;

}

// Step 2: Check palindrome

// For even length: x == reversedHalf

// For odd length: x == reversedHalf/10 (remove middle digit)

return (x == reversedHalf) || (x == reversedHalf / 10);

}

};

**Question-11**

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '\*' where:

* '.' Matches any single character.​​​​
* '\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:** s = "ab", p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Constraints:**

* 1 <= s.length <= 20
* 1 <= p.length <= 20
* s contains only lowercase English letters.
* p contains only lowercase English letters, '.', and '\*'.
* It is guaranteed for each appearance of the character '\*', there will be a previous valid character to match.

**Solution-**

class Solution {

public:

bool isMatch(string s, string p) {

int m = s.size(), n = p.size();

vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

dp[0][0] = true;

// Handle patterns like a\*, a\*b\*, a\*b\*c\* at the beginning

for (int j = 2; j <= n; j++) {

if (p[j - 1] == '\*') {

dp[0][j] = dp[0][j - 2];

}

}

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

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

if (p[j - 1] == '.' || p[j - 1] == s[i - 1]) {

dp[i][j] = dp[i - 1][j - 1];

}

else if (p[j - 1] == '\*') {

// Case 1: '\*' means zero occurrence

dp[i][j] = dp[i][j - 2];

// Case 2: '\*' means one or more occurrences

if (p[j - 2] == '.' || p[j - 2] == s[i - 1]) {

dp[i][j] = dp[i][j] || dp[i - 1][j];

}

}

}

}

return dp[m][n];

}

};

**Question-12**

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

**Notice** that you may not slant the container.

**Example 1:**



**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Constraints:**

* n == height.length
* 2 <= n <= 105
* 0 <= height[i] <= 104

**Solution-**

class Solution {

public:

int maxArea(vector<int>& height) {

int l = 0, r = height.size() - 1;

int maxWater = 0;

while (l < r) {

int h = min(height[l], height[r]);

int width = r - l;

maxWater = max(maxWater, h \* width);

// Move the pointer with smaller height

if (height[l] < height[r]) {

l++;

} else {

r--;

}

}

return maxWater;

}

};

**Question-13**

Seven different symbols represent Roman numerals with the following values:

| Symbol | Value |
| --- | --- |
| I | 1 |
| V | 5 |
| X | 10 |
| L | 50 |
| C | 100 |
| D | 500 |
| M | 1000 |

Roman numerals are formed by appending the conversions of decimal place values from highest to lowest. Converting a decimal place value into a Roman numeral has the following rules:

* If the value does not start with 4 or 9, select the symbol of the maximal value that can be subtracted from the input, append that symbol to the result, subtract its value, and convert the remainder to a Roman numeral.
* If the value starts with 4 or 9 use the **subtractive form** representing one symbol subtracted from the following symbol, for example, 4 is 1 (I) less than 5 (V): IV and 9 is 1 (I) less than 10 (X): IX. Only the following subtractive forms are used: 4 (IV), 9 (IX), 40 (XL), 90 (XC), 400 (CD) and 900 (CM).
* Only powers of 10 (I, X, C, M) can be appended consecutively at most 3 times to represent multiples of 10. You cannot append 5 (V), 50 (L), or 500 (D) multiple times. If you need to append a symbol 4 times use the **subtractive form**.

Given an integer, convert it to a Roman numeral.

**Example 1:**

**Input:** num = 3749

**Output:** "MMMDCCXLIX"

**Explanation:**

3000 = MMM as 1000 (M) + 1000 (M) + 1000 (M)

700 = DCC as 500 (D) + 100 (C) + 100 (C)

40 = XL as 10 (X) less of 50 (L)

9 = IX as 1 (I) less of 10 (X)

Note: 49 is not 1 (I) less of 50 (L) because the conversion is based on decimal places

**Example 2:**

**Input:** num = 58

**Output:** "LVIII"

**Explanation:**

50 = L

8 = VIII

**Example 3:**

**Input:** num = 1994

**Output:** "MCMXCIV"

**Explanation:**

1000 = M

900 = CM

90 = XC

4 = IV

**Constraints:**

* 1 <= num <= 3999

**Solution-**

class Solution {

public:

string intToRoman(int num) {

vector<pair<int, string>> valueSymbols = {

{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"}

};

string result = "";

for (auto &pair : valueSymbols) {

int value = pair.first;

string symbol = pair.second;

while (num >= value) {

result += symbol;

num -= value;

}

}

return result;

}

};

**Question-14**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

**Explanation:** III = 3.

**Example 2:**

**Input:** s = "LVIII"

**Output:** 58

**Explanation:** L = 50, V= 5, III = 3.

**Example 3:**

**Input:** s = "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= s.length <= 15
* s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
* It is **guaranteed** that s is a valid roman numeral in the range [1, 3999].

**Solution-**

class Solution {

public:

int romanToInt(string s) {

unordered\_map<char, int> values = {

{'I', 1}, {'V', 5}, {'X', 10},

{'L', 50}, {'C', 100},

{'D', 500}, {'M', 1000}

};

int result = 0;

int n = s.size();

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

if (i + 1 < n && values[s[i]] < values[s[i + 1]]) {

result -= values[s[i]]; // subtract if smaller before larger

} else {

result += values[s[i]]; // normal add

}

}

return result;

}

};

**Question-15**

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

**Example 1:**

**Input:** strs = ["flower","flow","flight"]

**Output:** "fl"

**Example 2:**

**Input:** strs = ["dog","racecar","car"]

**Output:** ""

**Explanation:** There is no common prefix among the input strings.

**Constraints:**

* 1 <= strs.length <= 200
* 0 <= strs[i].length <= 200
* strs[i] consists of only lowercase English letters if it is non-empty.

**Solution-**

#include <string>

#include <vector>

using namespace std;

class Solution {

public:

string longestCommonPrefix(vector<string>& strs) {

if (strs.empty()) return "";

string prefix = strs[0];

for (int i = 1; i < strs.size(); i++) {

while (strs[i].find(prefix) != 0) { // prefix not at start

prefix = prefix.substr(0, prefix.size() - 1);

if (prefix.empty()) return "";

}

}

return prefix;

}

};

**Question-16**

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

**Input:** nums = [-1,0,1,2,-1,-4]

**Output:** [[-1,-1,2],[-1,0,1]]

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

**Input:** nums = [0,1,1]

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

**Example 3:**

**Input:** nums = [0,0,0]

**Output:** [[0,0,0]]

**Explanation:** The only possible triplet sums up to 0.

**Constraints:**

* 3 <= nums.length <= 3000
* -105 <= nums[i] <= 105

**Solution-**

#include <vector>

#include <algorithm>

using namespace std;

class Solution {

public:

vector<vector<int>> threeSum(vector<int>& nums) {

vector<vector<int>> res;

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

int n = nums.size();

for (int i = 0; i < n - 2; i++) {

if (i > 0 && nums[i] == nums[i-1]) continue; // skip duplicates

int left = i + 1, right = n - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (sum == 0) {

res.push\_back({nums[i], nums[left], nums[right]});

left++;

right--;

while (left < right && nums[left] == nums[left-1]) left++; // skip duplicates

while (left < right && nums[right] == nums[right+1]) right--; // skip duplicates

} else if (sum < 0) {

left++;

} else {

right--;

}

}

}

return res;

}

};

**Question-17**

You are given two integers num1 and num2.

In one operation, you can choose integer i in the range [0, 60] and subtract 2i + num2 from num1.

Return *the integer denoting the****minimum****number of operations needed to make* num1 *equal to* 0.

If it is impossible to make num1 equal to 0, return -1.

**Example 1:**

**Input:** num1 = 3, num2 = -2

**Output:** 3

**Explanation:** We can make 3 equal to 0 with the following operations:

- We choose i = 2 and subtract 22 + (-2) from 3, 3 - (4 + (-2)) = 1.

- We choose i = 2 and subtract 22 + (-2) from 1, 1 - (4 + (-2)) = -1.

- We choose i = 0 and subtract 20 + (-2) from -1, (-1) - (1 + (-2)) = 0.

It can be proven, that 3 is the minimum number of operations that we need to perform.

**Example 2:**

**Input:** num1 = 5, num2 = 7

**Output:** -1

**Explanation:** It can be proven, that it is impossible to make 5 equal to 0 with the given operation.

**Constraints:**

* 1 <= num1 <= 109
* -109 <= num2 <= 109

**Solution-**

class Solution {

public:

int makeTheIntegerZero(int num1, int num2) {

for (int k = 1; k <= 60; ++k) {

long long remaining = (long long)num1 - (long long)k \* num2;

if (remaining < k) continue; // cannot form sum with k positive powers of 2

int ones = \_\_builtin\_popcountll(remaining);

if (ones <= k) return k;

}

return -1;

}

};

**Question-18**

You are given a 2D array queries, where queries[i] is of the form [l, r]. Each queries[i] defines an array of integers nums consisting of elements ranging from l to r, both **inclusive**.

In one operation, you can:

* Select two integers a and b from the array.
* Replace them with floor(a / 4) and floor(b / 4).

Your task is to determine the **minimum** number of operations required to reduce all elements of the array to zero for each query. Return the sum of the results for all queries.

**Example 1:**

**Input:** queries = [[1,2],[2,4]]

**Output:** 3

**Explanation:**

For queries[0]:

* The initial array is nums = [1, 2].
* In the first operation, select nums[0] and nums[1]. The array becomes [0, 0].
* The minimum number of operations required is 1.

For queries[1]:

* The initial array is nums = [2, 3, 4].
* In the first operation, select nums[0] and nums[2]. The array becomes [0, 3, 1].
* In the second operation, select nums[1] and nums[2]. The array becomes [0, 0, 0].
* The minimum number of operations required is 2.

The output is 1 + 2 = 3.

**Example 2:**

**Input:** queries = [[2,6]]

**Output:** 4

**Explanation:**

For queries[0]:

* The initial array is nums = [2, 3, 4, 5, 6].
* In the first operation, select nums[0] and nums[3]. The array becomes [0, 3, 4, 1, 6].
* In the second operation, select nums[2] and nums[4]. The array becomes [0, 3, 1, 1, 1].
* In the third operation, select nums[1] and nums[2]. The array becomes [0, 0, 0, 1, 1].
* In the fourth operation, select nums[3] and nums[4]. The array becomes [0, 0, 0, 0, 0].
* The minimum number of operations required is 4.

The output is 4.

**Constraints:**

* 1 <= queries.length <= 105
* queries[i].length == 2
* queries[i] == [l, r]
* 1 <= l < r <= 109

**Solution-**

class Solution {

public:

long long minOperations(vector<vector<int>>& queries) {

auto rangeSum = [&](long long l, long long r) {

long long ans = 0;

while (l <= r) {

long long p = 1;

int cnt = 0;

while (p \* 4 <= l) {

p \*= 4;

cnt++;

}

long long upper = min(r, 4 \* p - 1);

ans += (upper - l + 1) \* (cnt + 1LL);

l = upper + 1;

}

return ans;

};

long long total = 0;

for (auto &q : queries) {

long long l = q[0], r = q[1];

long long s = rangeSum(l, r);

total += (s + 1) / 2; // ceil(s/2)

}

return total;

}

};

**Question-19**

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Example 2:**

**Input:** nums = [0,0,0], target = 1

**Output:** 0

**Explanation:** The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

**Constraints:**

* 3 <= nums.length <= 500
* -1000 <= nums[i] <= 1000
* -104 <= target <= 104

**Solution-**

class Solution {

public:

int threeSumClosest(vector<int>& nums, int target) {

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

int n = nums.size();

int closest = nums[0] + nums[1] + nums[2];

for (int i = 0; i < n - 2; i++) {

int left = i + 1, right = n - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (abs(sum - target) < abs(closest - target)) {

closest = sum;

}

if (sum < target) {

left++;

} else if (sum > target) {

right--;

} else {

return target; // exact match

}

}

}

return closest;

}

};

**Question-20**

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**Constraints:**

* 0 <= digits.length <= 4
* digits[i] is a digit in the range ['2', '9'].

**Solution-**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

vector<string> letterCombinations(string digits) {

if (digits.empty()) return {};

vector<string> mapping = {

"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"

};

vector<string> result;

string current;

backtrack(digits, 0, current, result, mapping);

return result;

}

private:

void backtrack(string& digits, int index, string& current, vector<string>& result, vector<string>& mapping) {

if (index == digits.size()) {

result.push\_back(current);

return;

}

string letters = mapping[digits[index] - '0'];

for (char c : letters) {

current.push\_back(c);

backtrack(digits, index + 1, current, result, mapping);

current.pop\_back();

}

}

};

**Question-21**

Given an integer n, return **any** array containing n **unique** integers such that they add up to 0.

**Example 1:**

**Input:** n = 5

**Output:** [-7,-1,1,3,4]

**Explanation:** These arrays also are accepted [-5,-1,1,2,3] , [-3,-1,2,-2,4].

**Example 2:**

**Input:** n = 3

**Output:** [-1,0,1]

**Example 3:**

**Input:** n = 1

**Output:** [0]

**Constraints:**

* 1 <= n <= 1000

**Solution-**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

vector<vector<int>> fourSum(vector<int>& nums, long long target) {

vector<vector<int>> result;

int n = nums.size();

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

for (int i = 0; i < n - 3; i++) {

if (i > 0 && nums[i] == nums[i - 1]) continue; // skip duplicate

for (int j = i + 1; j < n - 2; j++) {

if (j > i + 1 && nums[j] == nums[j - 1]) continue; // skip duplicate

long long newTarget = target - (long long)nums[i] - (long long)nums[j];

int left = j + 1, right = n - 1;

while (left < right) {

long long sum = (long long)nums[left] + nums[right];

if (sum == newTarget) {

result.push\_back({nums[i], nums[j], nums[left], nums[right]});

while (left < right && nums[left] == nums[left + 1]) left++;

while (left < right && nums[right] == nums[right - 1]) right--;

left++;

right--;

} else if (sum < newTarget) {

left++;

} else {

right--;

}

}

}

}

return result;

}

};

**Question-22**

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

**Example 1:**



**Input:** head = [1,2,3,4,5], n = 2

**Output:** [1,2,3,5]

**Example 2:**

**Input:** head = [1], n = 1

**Output:** []

**Example 3:**

**Input:** head = [1,2], n = 1

**Output:** [1]

**Constraints:**

* The number of nodes in the list is sz.
* 1 <= sz <= 30
* 0 <= Node.val <= 100
* 1 <= n <= sz

**Follow up:** Could you do this in one pass?

**Solution-**

/\*\*

\* 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) {}

\* };

\*/

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

ListNode dummy(0, head);

ListNode\* fast = &dummy;

ListNode\* slow = &dummy;

// Move fast ahead by n+1 steps

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

fast = fast->next;

}

// Move fast to the end, maintaining the gap

while (fast != nullptr) {

fast = fast->next;

slow = slow->next;

}

// Remove nth node

ListNode\* toDelete = slow->next;

slow->next = slow->next->next;

delete toDelete;

return dummy.next;

}

};

**Question-23**

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

**Example 1:**

**Input:** s = "()"

**Output:** true

**Example 2:**

**Input:** s = "()[]{}"

**Output:** true

**Example 3:**

**Input:** s = "(]"

**Output:** false

**Example 4:**

**Input:** s = "([])"

**Output:** true

**Example 5:**

**Input:** s = "([)]"

**Output:** false

**Constraints:**

* 1 <= s.length <= 104
* s consists of parentheses only '()[]{}'.

**Solution-**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

bool isValid(string s) {

stack<char> st;

for (char c : s) {

if (c == '(' || c == '[' || c == '{') {

st.push(c);

} else {

if (st.empty()) return false;

char top = st.top();

st.pop();

if ((c == ')' && top != '(') ||

(c == ']' && top != '[') ||

(c == '}' && top != '{')) {

return false;

}

}

}

return st.empty();

}

};

**Question-24**

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

**Example 1:**



**Input:** list1 = [1,2,4], list2 = [1,3,4]

**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** list1 = [], list2 = []

**Output:** []

**Example 3:**

**Input:** list1 = [], list2 = [0]

**Output:** [0]

**Constraints:**

* The number of nodes in both lists is in the range [0, 50].
* -100 <= Node.val <= 100
* Both list1 and list2 are sorted in **non-decreasing** order.

**Solution-**

/\*\*

\* 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) {}

\* };

\*/

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

ListNode dummy; // dummy head

ListNode\* tail = &dummy;

while (list1 != nullptr && list2 != nullptr) {

if (list1->val <= list2->val) {

tail->next = list1;

list1 = list1->next;

} else {

tail->next = list2;

list2 = list2->next;

}

tail = tail->next;

}

// Append the remaining nodes

if (list1 != nullptr) tail->next = list1;

else tail->next = list2;

return dummy.next;

}

};

**Question-25**

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

**Example 2:**

**Input:** n = 1

**Output:** ["()"]

**Constraints:**

* 1 <= n <= 8

**Solution-**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

vector<string> generateParenthesis(int n) {

vector<string> result;

string current;

backtrack(result, current, 0, 0, n);

return result;

}

private:

void backtrack(vector<string>& result, string& current, int open, int close, int n) {

if (current.size() == 2 \* n) {

result.push\_back(current);

return;

}

if (open < n) {

current.push\_back('(');

backtrack(result, current, open + 1, close, n);

current.pop\_back();

}

if (close < open) {

current.push\_back(')');

backtrack(result, current, open, close + 1, n);

current.pop\_back();

}

}

};

**Question-26**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

**Input:** lists = [[1,4,5],[1,3,4],[2,6]]

**Output:** [1,1,2,3,4,4,5,6]

**Explanation:** The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted linked list:

1->1->2->3->4->4->5->6

**Example 2:**

**Input:** lists = []

**Output:** []

**Example 3:**

**Input:** lists = [[]]

**Output:** []

**Constraints:**

* k == lists.length
* 0 <= k <= 104
* 0 <= lists[i].length <= 500
* -104 <= lists[i][j] <= 104
* lists[i] is sorted in **ascending order**.
* The sum of lists[i].length will not exceed 104.

**Solution-**

/\*\*

\* 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) {}

\* };

\*/

class Solution {

public:

ListNode\* mergeKLists(vector<ListNode\*>& lists) {

struct Compare {

bool operator()(ListNode\* a, ListNode\* b) {

return a->val > b->val; // min-heap

}

};

priority\_queue<ListNode\*, vector<ListNode\*>, Compare> pq;

// Push the head of each list

for (auto node : lists) {

if (node) pq.push(node);

}

ListNode dummy(0);

ListNode\* tail = &dummy;

while (!pq.empty()) {

ListNode\* node = pq.top();

pq.pop();

tail->next = node;

tail = tail->next;

if (node->next) {

pq.push(node->next);

}

}

return dummy.next;

}

};

**Question-27**

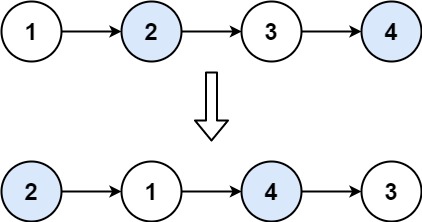
Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

**Example 1:**

**Input:** head = [1,2,3,4]

**Output:** [2,1,4,3]

**Explanation:**



**Example 2:**

**Input:** head = []

**Output:** []

**Example 3:**

**Input:** head = [1]

**Output:** [1]

**Example 4:**

**Input:** head = [1,2,3]

**Output:** [2,1,3]

**Constraints:**

* The number of nodes in the list is in the range [0, 100].
* 0 <= Node.val <= 100

**Solution-**

/\*\*

\* 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) {}

\* };

\*/

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

ListNode dummy(0, head);

ListNode\* prev = &dummy;

while (prev->next && prev->next->next) {

ListNode\* first = prev->next;

ListNode\* second = prev->next->next;

// Swapping

first->next = second->next;

second->next = first;

prev->next = second;

// Move prev two steps ahead

prev = first;

}

return dummy.next;

}

};

**Question-28**

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:

Input: head = [1,2,3,4,5], k = 2

Output: [2,1,4,3,5]

Example 2:

Input: head = [1,2,3,4,5], k = 3

Output: [3,2,1,4,5]

Constraints:

The number of nodes in the list is n.

1 <= k <= n <= 5000

0 <= Node.val <= 1000

Follow-up: Can you solve the problem in O(1) extra memory space?

**Solution-**

class Solution {

public:

// Helper function to reverse [first, last)

ListNode\* reverse(ListNode\* first, ListNode\* last) {

ListNode\* prev = last;

while (first != last) {

ListNode\* temp = first->next;

first->next = prev;

prev = first;

first = temp;

}

return prev;

}

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode dummy(0, head);

ListNode\* prevGroupEnd = &dummy;

while (true) {

ListNode\* kth = prevGroupEnd;

for (int i = 0; i < k && kth; i++) {

kth = kth->next;

}

if (!kth) break; // less than k nodes remain

ListNode\* groupStart = prevGroupEnd->next;

ListNode\* nextGroupStart = kth->next;

prevGroupEnd->next = reverse(groupStart, nextGroupStart);

groupStart->next = nextGroupStart;

prevGroupEnd = groupStart;

}

return dummy.next;

}

};

**Question-29**

**No-Zero integer** is a positive integer that **does not contain any 0** in its decimal representation.

Given an integer n, return *a list of two integers* [a, b] *where*:

* a and b are **No-Zero integers**.
* a + b = n

The test cases are generated so that there is at least one valid solution. If there are many valid solutions, you can return any of them.

**Example 1:**

**Input:** n = 2

**Output:** [1,1]

**Explanation:** Let a = 1 and b = 1.

Both a and b are no-zero integers, and a + b = 2 = n.

**Example 2:**

**Input:** n = 11

**Output:** [2,9]

**Explanation:** Let a = 2 and b = 9.

Both a and b are no-zero integers, and a + b = 11 = n.

Note that there are other valid answers as [8, 3] that can be accepted.

**Constraints:**

* 2 <= n <= 104

**Solution-**

**Question-30**

On a social network consisting of m users and some friendships between users, two users can communicate with each other if they know a common language.

You are given an integer n, an array languages, and an array friendships where:

* There are n languages numbered 1 through n,
* languages[i] is the set of languages the i​​​​​​th​​​​ user knows, and
* friendships[i] = [u​​​​​​i​​​, v​​​​​​i] denotes a friendship between the users u​​​​​​​​​​​i​​​​​ and vi.

You can choose **one** language and teach it to some users so that all friends can communicate with each other. Return *the* ***minimum****number of users you need to teach.*

Note that friendships are not transitive, meaning if x is a friend of y and y is a friend of z, this doesn't guarantee that x is a friend of z.

**Example 1:**

**Input:** n = 2, languages = [[1],[2],[1,2]], friendships = [[1,2],[1,3],[2,3]]

**Output:** 1

**Explanation:** You can either teach user 1 the second language or user 2 the first language.

**Example 2:**

**Input:** n = 3, languages = [[2],[1,3],[1,2],[3]], friendships = [[1,4],[1,2],[3,4],[2,3]]

**Output:** 2

**Explanation:** Teach the third language to users 1 and 3, yielding two users to teach.

**Constraints:**

* 2 <= n <= 500
* languages.length == m
* 1 <= m <= 500
* 1 <= languages[i].length <= n
* 1 <= languages[i][j] <= n
* 1 <= u​​​​​​i < v​​​​​​i <= languages.length
* 1 <= friendships.length <= 500
* All tuples (u​​​​​i, v​​​​​​i) are unique
* languages[i] contains only unique values

**Solution-**

class Solution {

public:

int minimumTeachings(int n, vector<vector<int>>& languages, vector<vector<int>>& friendships) {

int m = languages.size();

// Convert each user's languages to a set for fast lookup

vector<unordered\_set<int>> userLang(m + 1);

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

for (int lang : languages[i]) {

userLang[i + 1].insert(lang);

}

}

// Find users who cannot communicate with their friends

unordered\_set<int> needTeach;

for (auto& f : friendships) {

int u = f[0], v = f[1];

bool canCommunicate = false;

for (int lang : userLang[u]) {

if (userLang[v].count(lang)) {

canCommunicate = true;

break;

}

}

if (!canCommunicate) {

needTeach.insert(u);

needTeach.insert(v);

}

}

// If all friendships are already communicable

if (needTeach.empty()) return 0;

int ans = INT\_MAX;

// Try each language as the "teaching" language

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

int teachCount = 0;

for (int user : needTeach) {

if (!userLang[user].count(lang)) {

teachCount++;

}

}

ans = min(ans, teachCount);

}

return ans;

}

};

**Question-31**

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in*nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

* Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
* Return k.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

**Input:** nums = [1,1,2]

**Output:** 2, nums = [1,2,\_]

**Explanation:** Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**Example 2:**

**Input:** nums = [0,0,1,1,1,2,2,3,3,4]

**Output:** 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

**Explanation:** Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -100 <= nums[i] <= 100
* nums is sorted in **non-decreasing** order.

**Solution-**

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

if (nums.empty()) return 0;

int k = 0; // pointer for the place of last unique element

for (int i = 1; i < nums.size(); i++) {

if (nums[i] != nums[k]) {

k++;

nums[k] = nums[i]; // move unique element forward

}

}

return k + 1; // length of unique elements

}

};

**Question-32**

You are given a string s consisting of lowercase English letters ('a' to 'z').

Your task is to:

* Find the vowel (one of 'a', 'e', 'i', 'o', or 'u') with the **maximum** frequency.
* Find the consonant (all other letters excluding vowels) with the **maximum** frequency.

Return the sum of the two frequencies.

**Note**: If multiple vowels or consonants have the same maximum frequency, you may choose any one of them. If there are no vowels or no consonants in the string, consider their frequency as 0.

The **frequency** of a letter x is the number of times it occurs in the string.

**Example 1:**

**Input:** s = "successes"

**Output:** 6

**Explanation:**

* The vowels are: 'u' (frequency 1), 'e' (frequency 2). The maximum frequency is 2.
* The consonants are: 's' (frequency 4), 'c' (frequency 2). The maximum frequency is 4.
* The output is 2 + 4 = 6.

**Example 2:**

**Input:** s = "aeiaeia"

**Output:** 3

**Explanation:**

* The vowels are: 'a' (frequency 3), 'e' ( frequency 2), 'i' (frequency 2). The maximum frequency is 3.
* There are no consonants in s. Hence, maximum consonant frequency = 0.
* The output is 3 + 0 = 3.

**Constraints:**

* 1 <= s.length <= 100
* s consists of lowercase English letters only.

**Solution:**

class Solution {

public:

int maxVowelConsonantSum(string s) {

vector<int> freq(26, 0);

for (char c : s) {

freq[c - 'a']++;

}

auto isVowel = [](char c) {

return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';

};

int maxVowel = 0, maxConsonant = 0;

for (int i = 0; i < 26; i++) {

if (freq[i] > 0) {

char c = 'a' + i;

if (isVowel(c))

maxVowel = max(maxVowel, freq[i]);

else

maxConsonant = max(maxConsonant, freq[i]);

}

}

return maxVowel + maxConsonant;

}

};

**Question-33**

You are given a string s consisting of lowercase English letters ('a' to 'z').

Your task is to:

Find the vowel (one of 'a', 'e', 'i', 'o', or 'u') with the maximum frequency.

Find the consonant (all other letters excluding vowels) with the maximum frequency.

Return the sum of the two frequencies.

Note: If multiple vowels or consonants have the same maximum frequency, you may choose any one of them. If there are no vowels or no consonants in the string, consider their frequency as 0.

The frequency of a letter x is the number of times it occurs in the string.

Example 1:

Input: s = "successes"

Output: 6

Explanation:

The vowels are: 'u' (frequency 1), 'e' (frequency 2). The maximum frequency is 2.

The consonants are: 's' (frequency 4), 'c' (frequency 2). The maximum frequency is 4.

The output is 2 + 4 = 6.

Example 2:

Input: s = "aeiaeia"

Output: 3

Explanation:

The vowels are: 'a' (frequency 3), 'e' ( frequency 2), 'i' (frequency 2). The maximum frequency is 3.

There are no consonants in s. Hence, maximum consonant frequency = 0.

The output is 3 + 0 = 3.

Constraints:

1 <= s.length <= 100

s consists of lowercase English letters only.

**Solution:**

class Solution {

public:

int maxVowelConsonantSum(string s) {

vector<int> freq(26, 0);

for (char c : s) {

freq[c - 'a']++;

}

auto isVowel = [](char c) {

return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';

};

int maxVowel = 0, maxConsonant = 0;

for (int i = 0; i < 26; i++) {

if (freq[i] > 0) {

char c = 'a' + i;

if (isVowel(c))

maxVowel = max(maxVowel, freq[i]);

else

maxConsonant = max(maxConsonant, freq[i]);

}

}

return maxVowel + maxConsonant;

}

};

**Question-34**

There is a malfunctioning keyboard where some letter keys do not work. All other keys on the keyboard work properly.

Given a string text of words separated by a single space (no leading or trailing spaces) and a string brokenLetters of all **distinct** letter keys that are broken, return *the****number of words****in* text *you can fully type using this keyboard*.

**Example 1:**

**Input:** text = "hello world", brokenLetters = "ad"

**Output:** 1

**Explanation:** We cannot type "world" because the 'd' key is broken.

**Example 2:**

**Input:** text = "leet code", brokenLetters = "lt"

**Output:** 1

**Explanation:** We cannot type "leet" because the 'l' and 't' keys are broken.

**Example 3:**

**Input:** text = "leet code", brokenLetters = "e"

**Output:** 0

**Explanation:** We cannot type either word because the 'e' key is broken.

**Constraints:**

* 1 <= text.length <= 104
* 0 <= brokenLetters.length <= 26
* text consists of words separated by a single space without any leading or trailing spaces.
* Each word only consists of lowercase English letters.
* brokenLetters consists of **distinct** lowercase English letters.

**Solution:**

#include <iostream>

#include <unordered\_set>

#include <sstream>

using namespace std;

class Solution {

public:

int canBeTypedWords(string text, string brokenLetters) {

unordered\_set<char> brokenSet;

for (char c : brokenLetters) {

brokenSet.insert(c);

}

int count = 0;

istringstream iss(text);

string word;

while (iss >> word) {

bool canType = true;

for (char c : word) {

if (brokenSet.count(c)) {

canType = false;

break;

}

}

if (canType) count++;

}

return count;

}

};

**Question-35:**

Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.

Consider the number of elements in nums which are not equal to val be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums.

Return k.

Custom Judge:

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int val = ...; // Value to remove

int[] expectedNums = [...]; // The expected answer with correct length.

// It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;

sort(nums, 0, k); // Sort the first k elements of nums

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be accepted.

Example 1:

Input: nums = [3,2,2,3], val = 3

Output: 2, nums = [2,2,\_,\_]

Explanation: Your function should return k = 2, with the first two elements of nums being 2.

It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,1,2,2,3,0,4,2], val = 2

Output: 5, nums = [0,1,4,0,3,\_,\_,\_]

Explanation: Your function should return k = 5, with the first five elements of nums containing 0, 0, 1, 3, and 4.

Note that the five elements can be returned in any order.

It does not matter what you leave beyond the returned k (hence they are underscores).

Constraints:

0 <= nums.length <= 100

0 <= nums[i] <= 50

0 <= val <= 100

class Solution {

public:

int removeElement(vector<int>& nums, int val) {

}

};

**Solution:**

#include <vector>

using namespace std;

class Solution {

public:

int removeElement(vector<int>& nums, int val) {

int k = 0; // Index to place the next valid element

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

if (nums[i] != val) {

nums[k] = nums[i]; // Keep the element

k++;

}

}

return k;

}

};

**Question-36:**

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

Example 2:

Input: haystack = "leetcode", needle = "leeto"

Output: -1

Explanation: "leeto" did not occur in "leetcode", so we return -1.

Constraints:

1 <= haystack.length, needle.length <= 104

haystack and needle consist of only lowercase English characters.

**Solution:**

#include <string>

using namespace std;

class Solution {

public:

int strStr(string haystack, string needle) {

int n = haystack.size();

int m = needle.size();

if (m == 0) return 0; // Edge case: empty needle

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

if (haystack.substr(i, m) == needle) {

return i;

}

}

return -1;

}

};

**Question-37:**

Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.

The integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.

Return the quotient after dividing dividend by divisor.

Note: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: [−231, 231 − 1]. For this problem, if the quotient is strictly greater than 231 - 1, then return 231 - 1, and if the quotient is strictly less than -231, then return -231.

Example 1:

Input: dividend = 10, divisor = 3

Output: 3

Explanation: 10/3 = 3.33333.. which is truncated to 3.

Example 2:

Input: dividend = 7, divisor = -3

Output: -2

Explanation: 7/-3 = -2.33333.. which is truncated to -2.

Constraints:

-231 <= dividend, divisor <= 231 - 1

divisor != 0

**Solution:**

#include <climits>

using namespace std;

class Solution {

public:

int divide(int dividend, int divisor) {

// Handle overflow case

if (dividend == INT\_MIN && divisor == -1)

return INT\_MAX;

// Determine sign of result

bool negative = (dividend < 0) ^ (divisor < 0);

// Convert to long to handle edge cases (e.g., abs(INT\_MIN))

long long ldividend = abs((long long)dividend);

long long ldivisor = abs((long long)divisor);

long long quotient = 0;

// Bit manipulation approach: subtract multiples of divisor

for (int i = 31; i >= 0; i--) {

if ((ldividend >> i) >= ldivisor) {

ldividend -= (ldivisor << i);

quotient += (1LL << i);

}

}

// Apply sign

quotient = negative ? -quotient : quotient;

// Clamp result to 32-bit signed integer range

return quotient > INT\_MAX ? INT\_MAX : quotient;

}

};

**Question-38:**

You are given a string s and an array of strings words. All the strings of words are of **the same length**.

A **concatenated string** is a string that exactly contains all the strings of any permutation of words concatenated.

* For example, if words = ["ab","cd","ef"], then "abcdef", "abefcd", "cdabef", "cdefab", "efabcd", and "efcdab" are all concatenated strings. "acdbef" is not a concatenated string because it is not the concatenation of any permutation of words.

Return an array of *the starting indices* of all the concatenated substrings in s. You can return the answer in **any order**.

**Example 1:**

**Input:** s = "barfoothefoobarman", words = ["foo","bar"]

**Output:** [0,9]

**Explanation:**

The substring starting at 0 is "barfoo". It is the concatenation of ["bar","foo"] which is a permutation of words.  
The substring starting at 9 is "foobar". It is the concatenation of ["foo","bar"] which is a permutation of words.

**Example 2:**

**Input:** s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]

**Output:** []

**Explanation:**

There is no concatenated substring.

**Example 3:**

**Input:** s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]

**Output:** [6,9,12]

**Explanation:**

The substring starting at 6 is "foobarthe". It is the concatenation of ["foo","bar","the"].  
The substring starting at 9 is "barthefoo". It is the concatenation of ["bar","the","foo"].  
The substring starting at 12 is "thefoobar". It is the concatenation of ["the","foo","bar"].

**Constraints:**

* 1 <= s.length <= 104
* 1 <= words.length <= 5000
* 1 <= words[i].length <= 30
* s and words[i] consist of lowercase English letters.

**Solution:**

#include <vector>

#include <string>

#include <unordered\_map>

using namespace std;

class Solution {

public:

vector<int> findSubstring(string s, vector<string>& words) {

vector<int> result;

if (words.empty() || s.empty()) return result;

int wordLen = words[0].length();

int wordCount = words.size();

int substringLen = wordLen \* wordCount;

if (s.length() < substringLen) return result;

unordered\_map<string, int> wordFreq;

for (const string& word : words) {

wordFreq[word]++;

}

// Loop over wordLen different offsets

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

int left = i, count = 0;

unordered\_map<string, int> windowFreq;

for (int j = i; j <= s.length() - wordLen; j += wordLen) {

string word = s.substr(j, wordLen);

if (wordFreq.find(word) != wordFreq.end()) {

windowFreq[word]++;

count++;

// If word frequency exceeds expected, move left pointer

while (windowFreq[word] > wordFreq[word]) {

string leftWord = s.substr(left, wordLen);

windowFreq[leftWord]--;

left += wordLen;

count--;

}

// If window contains exactly all words

if (count == wordCount) {

result.push\_back(left);

// Move left to look for new window

string leftWord = s.substr(left, wordLen);

windowFreq[leftWord]--;

left += wordLen;

count--;

}

} else {

// Reset window if word not found

windowFreq.clear();

count = 0;

left = j + wordLen;

}

}

}

return result;

}

};

**Question-39:**

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

* For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

* For example, the next permutation of arr = [1,2,3] is [1,3,2].
* Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
* While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** nums = [3,2,1]

**Output:** [1,2,3]

**Example 3:**

**Input:** nums = [1,1,5]

**Output:** [1,5,1]

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

**Solution:**

#include <vector>

#include <algorithm>

using namespace std;

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int n = nums.size();

int i = n - 2;

// Step 1: Find the first decreasing element

while (i >= 0 && nums[i] >= nums[i + 1]) {

i--;

}

if (i >= 0) {

// Step 2: Find the next larger element to swap with

int j = n - 1;

while (nums[j] <= nums[i]) {

j--;

}

swap(nums[i], nums[j]);

}

// Step 3: Reverse the suffix

reverse(nums.begin() + i + 1, nums.end());

}

};

**Question-40:**

Given a string containing just the characters '(' and ')', return *the length of the longest valid (well-formed) parentheses substring*.

**Example 1:**

**Input:** s = "(()"

**Output:** 2

**Explanation:** The longest valid parentheses substring is "()".

**Example 2:**

**Input:** s = ")()())"

**Output:** 4

**Explanation:** The longest valid parentheses substring is "()()".

**Example 3:**

**Input:** s = ""

**Output:** 0

**Constraints:**

* 0 <= s.length <= 3 \* 104
* s[i] is '(', or ')'.

**Solution:**

**#include <string>**

**#include <stack>**

**#include <algorithm>**

**using namespace std;**

**class Solution {**

**public:**

**int longestValidParentheses(string s) {**

**stack<int> stk;**

**int maxLen = 0;**

**int base = -1; // Initialize base index**

**for (int i = 0; i < s.length(); ++i) {**

**if (s[i] == '(') {**

**stk.push(i);**

**} else {**

**if (stk.empty()) {**

**base = i; // Reset base when unmatched ')'**

**} else {**

**stk.pop();**

**if (stk.empty()) {**

**maxLen = max(maxLen, i - base);**

**} else {**

**maxLen = max(maxLen, i - stk.top());**

**}**

**}**

**}**

**}**

**return maxLen;**

**}**

**};**

**Question-41:**

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly left rotated** at an unknown index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be left rotated by 3 indices and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of*target*if it is in*nums*, or*-1*if it is not in*nums.

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

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2], target = 3

**Output:** -1

**Example 3:**

**Input:** nums = [1], target = 0

**Output:** -1

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is an ascending array that is possibly rotated.
* -104 <= target <= 104

**Solution:**

#include <vector>

using namespace std;

class Solution {

public:

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

int left = 0;

int right = nums.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) {

return mid;

}

// Left half is sorted

if (nums[left] <= nums[mid]) {

if (nums[left] <= target && target < nums[mid]) {

right = mid - 1; // Target in left half

} else {

left = mid + 1; // Target in right half

}

}

// Right half is sorted

else {

if (nums[mid] < target && target <= nums[right]) {

left = mid + 1; // Target in right half

} else {

right = mid - 1; // Target in left half

}

}

}

return -1; // Target not found

}

};

**Question-42:**

vGiven an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

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

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

**Example 2:**

**Input:** nums = [5,7,7,8,8,10], target = 6

**Output:** [-1,-1]

**Example 3:**

**Input:** nums = [], target = 0

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

**Solution:**

#include <vector>

using namespace std;

class Solution {

public:

vector<int> searchRange(vector<int>& nums, int target) {

int first = findBound(nums, target, true);

int last = findBound(nums, target, false);

return {first, last};

}

private:

int findBound(const vector<int>& nums, int target, bool isFirst) {

int left = 0;

int right = nums.size() - 1;

int bound = -1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) {

bound = mid;

if (isFirst) {

right = mid - 1; // Search left half

} else {

left = mid + 1; // Search right half

}

} else if (nums[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return bound;

}

};

**Question-43:**

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.

Example 1:

Input: nums = [1,3,5,6], target = 5

Output: 2

Example 2:

Input: nums = [1,3,5,6], target = 2

Output: 1

Example 3:

Input: nums = [1,3,5,6], target = 7

Output: 4

Constraints:

1 <= nums.length <= 104

-104 <= nums[i] <= 104

nums contains distinct values sorted in ascending order.

-104 <= target <= 104

**Solution:**

#include <vector>

using namespace std;

class Solution {

public:

int searchInsert(vector<int>& nums, int target) {

int left = 0;

int right = nums.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) {

return mid; // Target found

} else if (nums[mid] < target) {

left = mid + 1; // Search in the right half

} else {

right = mid - 1; // Search in the left half

}

}

// If not found, left is the correct insertion position

return left;

}

};

**Question-44:**

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

**Note:**

* A Sudoku board (partially filled) could be valid but is not necessarily solvable.
* Only the filled cells need to be validated according to the mentioned rules.

**Example 1:**

A square puzzle with numbers

AI-generated content may be incorrect.

**Input:** board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** true

**Example 2:**

**Input:** board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** false

**Explanation:** Same as Example 1, except with the **5** in the top left corner being modified to **8**. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

**Constraints:**

* board.length == 9
* board[i].length == 9
* board[i][j] is a digit 1-9 or '.'.

**Solution:**

#include <vector>

#include <unordered\_set>

#include <string>

using namespace std;

class Solution {

public:

bool isValidSudoku(vector<vector<char>>& board) {

unordered\_set<string> seen;

for (int i = 0; i < 9; ++i) {

for (int j = 0; j < 9; ++j) {

char current = board[i][j];

if (current != '.') {

string rowKey = "row" + to\_string(i) + "-" + current;

string colKey = "col" + to\_string(j) + "-" + current;

string boxKey = "box" + to\_string(i / 3) + to\_string(j / 3) + "-" + current;

if (seen.count(rowKey) || seen.count(colKey) || seen.count(boxKey)) {

return false; // Duplicate found

}

seen.insert(rowKey);

seen.insert(colKey);

seen.insert(boxKey);

}

}

}

return true;

}

};

**Question-45:**

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

**Example 1:**

A square puzzle with numbers

AI-generated content may be incorrect.

**Input:** board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

**Output:** [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

**Explanation:** The input board is shown above and the only valid solution is shown below:

A square of numbers with red and black text

AI-generated content may be incorrect.

**Constraints:**

* board.length == 9
* board[i].length == 9
* board[i][j] is a digit or '.'.
* It is **guaranteed** that the input board has only one solution.

**Solution:**

#include <vector>

using namespace std;

class Solution {

public:

bool isValid(vector<vector<char>>& board, int row, int col, char c) {

for (int i = 0; i < 9; ++i) {

// Check row

if (board[row][i] == c) return false;

// Check column

if (board[i][col] == c) return false;

// Check 3x3 sub-box

if (board[3 \* (row / 3) + i / 3][3 \* (col / 3) + i % 3] == c) return false;

}

return true;

}

bool solveSudokuHelper(vector<vector<char>>& board) {

for (int row = 0; row < 9; ++row) {

for (int col = 0; col < 9; ++col) {

if (board[row][col] == '.') {

for (char c = '1'; c <= '9'; ++c) {

if (isValid(board, row, col, c)) {

board[row][col] = c;

if (solveSudokuHelper(board)) return true;

board[row][col] = '.'; // Backtrack

}

}

return false; // No valid number found

}

}

}

return true; // Solved

}

void solveSudoku(vector<vector<char>>& board) {

solveSudokuHelper(board);

}

};

**Question-45**

There is a task management system that allows users to manage their tasks, each associated with a priority. The system should efficiently handle adding, modifying, executing, and removing tasks.

Implement the TaskManager class:

* TaskManager(vector<vector<int>>& tasks) initializes the task manager with a list of user-task-priority triples. Each element in the input list is of the form [userId, taskId, priority], which adds a task to the specified user with the given priority.
* void add(int userId, int taskId, int priority) adds a task with the specified taskId and priority to the user with userId. It is **guaranteed** that taskId does not *exist* in the system.
* void edit(int taskId, int newPriority) updates the priority of the existing taskId to newPriority. It is **guaranteed** that taskId *exists* in the system.
* void rmv(int taskId) removes the task identified by taskId from the system. It is **guaranteed** that taskId *exists* in the system.
* int execTop() executes the task with the **highest** priority across all users. If there are multiple tasks with the same **highest** priority, execute the one with the highest taskId. After executing, thetaskIdis **removed** from the system. Return the userId associated with the executed task. If no tasks are available, return -1.

**Note** that a user may be assigned multiple tasks.

**Example 1:**

**Input:**  
["TaskManager", "add", "edit", "execTop", "rmv", "add", "execTop"]  
[[[[1, 101, 10], [2, 102, 20], [3, 103, 15]]], [4, 104, 5], [102, 8], [], [101], [5, 105, 15], []]

**Output:**  
[null, null, null, 3, null, null, 5]

**Explanation**

TaskManager taskManager = new TaskManager([[1, 101, 10], [2, 102, 20], [3, 103, 15]]); // Initializes with three tasks for Users 1, 2, and 3.  
taskManager.add(4, 104, 5); // Adds task 104 with priority 5 for User 4.  
taskManager.edit(102, 8); // Updates priority of task 102 to 8.  
taskManager.execTop(); // return 3. Executes task 103 for User 3.  
taskManager.rmv(101); // Removes task 101 from the system.  
taskManager.add(5, 105, 15); // Adds task 105 with priority 15 for User 5.  
taskManager.execTop(); // return 5. Executes task 105 for User 5.

**Constraints:**

* 1 <= tasks.length <= 105
* 0 <= userId <= 105
* 0 <= taskId <= 105
* 0 <= priority <= 109
* 0 <= newPriority <= 109
* At most 2 \* 105 calls will be made in **total** to add, edit, rmv, and execTop methods.
* The input is generated such that taskId will be valid.

**Solution:-**

#include <bits/stdc++.h>

using namespace std;

class TaskManager {

unordered\_map<int, pair<int,int>> taskMap;

// taskId -> {userId, priority}

// Max-heap by priority, then by taskId

priority\_queue<pair<pair<int,int>, int>> pq;

// {{priority, taskId}, userId}

public:

// Constructor

TaskManager(vector<vector<int>>& tasks) {

for (auto &t : tasks) {

int userId = t[0], taskId = t[1], priority = t[2];

taskMap[taskId] = {userId, priority};

pq.push({{priority, taskId}, userId});

}

}

void add(int userId, int taskId, int priority) {

taskMap[taskId] = {userId, priority};

pq.push({{priority, taskId}, userId});

}

void edit(int taskId, int newPriority) {

if (taskMap.find(taskId) != taskMap.end()) {

int userId = taskMap[taskId].first;

taskMap[taskId] = {userId, newPriority};

pq.push({{newPriority, taskId}, userId});

}

}

void rmv(int taskId) {

taskMap.erase(taskId); // lazy delete

}

int execTop() {

while (!pq.empty()) {

auto [priTask, userId] = pq.top();

int priority = priTask.first;

int taskId = priTask.second;

pq.pop();

if (taskMap.find(taskId) != taskMap.end() && taskMap[taskId].second == priority) {

taskMap.erase(taskId);

return userId;

}

}

return -1;

}

};

**Question-46**

Alice and Bob are playing a game on a string.

You are given a string s, Alice and Bob will take turns playing the following game where Alice starts first:

On Alice's turn, she has to remove any non-empty substring from s that contains an odd number of vowels.

On Bob's turn, he has to remove any non-empty substring from s that contains an even number of vowels.

The first player who cannot make a move on their turn loses the game. We assume that both Alice and Bob play optimally.

Return true if Alice wins the game, and false otherwise.

The English vowels are: a, e, i, o, and u.

Example 1:

Input: s = "leetcoder"

Output: true

Explanation:

Alice can win the game as follows:

Alice plays first, she can delete the underlined substring in s = "leetcoder" which contains 3 vowels. The resulting string is s = "der".

Bob plays second, he can delete the underlined substring in s = "der" which contains 0 vowels. The resulting string is s = "er".

Alice plays third, she can delete the whole string s = "er" which contains 1 vowel.

Bob plays fourth, since the string is empty, there is no valid play for Bob. So Alice wins the game.

Example 2:

Input: s = "bbcd"

Output: false

Explanation:

There is no valid play for Alice in her first turn, so Alice loses the game.

Constraints:

1 <= s.length <= 105

s consists only of lowercase English letters.

**Solution:**

class Solution {

public:

bool doesAliceWin(string s) {

string vowels = "aeiou";

for (char c : s) {

if (vowels.find(c) != string::npos) {

return true; // Alice wins if at least one vowel exists

}

}

return false; // No vowels → Alice cannot move → Bob wins

}

};

**Question-47**

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:

Input: head = [1,2,3,4,5], k = 2

Output: [2,1,4,3,5]

Example 2:

Input: head = [1,2,3,4,5], k = 3

Output: [3,2,1,4,5]

Constraints:

The number of nodes in the list is n.

1 <= k <= n <= 5000

0 <= Node.val <= 1000

Follow-up: Can you solve the problem in O(1) extra memory space?

**Solution:**

class Solution {

public:

// Helper function to reverse [first, last)

ListNode\* reverse(ListNode\* first, ListNode\* last) {

ListNode\* prev = last;

while (first != last) {

ListNode\* temp = first->next;

first->next = prev;

prev = first;

first = temp;

}

return prev;

}

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode dummy(0, head);

ListNode\* prevGroupEnd = &dummy;

while (true) {

ListNode\* kth = prevGroupEnd;

for (int i = 0; i < k && kth; i++) {

kth = kth->next;

}

if (!kth) break; // less than k nodes remain

ListNode\* groupStart = prevGroupEnd->next;

ListNode\* nextGroupStart = kth->next;

prevGroupEnd->next = reverse(groupStart, nextGroupStart);

groupStart->next = nextGroupStart;

prevGroupEnd = groupStart;

}

return dummy.next;

}

};

**Question-48**

There is a task management system that allows users to manage their tasks, each associated with a priority. The system should efficiently handle adding, modifying, executing, and removing tasks.

Implement the TaskManager class:

TaskManager(vector<vector<int>>& tasks) initializes the task manager with a list of user-task-priority triples. Each element in the input list is of the form [userId, taskId, priority], which adds a task to the specified user with the given priority.

void add(int userId, int taskId, int priority) adds a task with the specified taskId and priority to the user with userId. It is guaranteed that taskId does not exist in the system.

void edit(int taskId, int newPriority) updates the priority of the existing taskId to newPriority. It is guaranteed that taskId exists in the system.

void rmv(int taskId) removes the task identified by taskId from the system. It is guaranteed that taskId exists in the system.

int execTop() executes the task with the highest priority across all users. If there are multiple tasks with the same highest priority, execute the one with the highest taskId. After executing, the taskId is removed from the system. Return the userId associated with the executed task. If no tasks are available, return -1.

Note that a user may be assigned multiple tasks.

Example 1:

Input:

["TaskManager", "add", "edit", "execTop", "rmv", "add", "execTop"]

[[[[1, 101, 10], [2, 102, 20], [3, 103, 15]]], [4, 104, 5], [102, 8], [], [101], [5, 105, 15], []]

Output:

[null, null, null, 3, null, null, 5]

Explanation

TaskManager taskManager = new TaskManager([[1, 101, 10], [2, 102, 20], [3, 103, 15]]); // Initializes with three tasks for Users 1, 2, and 3.

taskManager.add(4, 104, 5); // Adds task 104 with priority 5 for User 4.

taskManager.edit(102, 8); // Updates priority of task 102 to 8.

taskManager.execTop(); // return 3. Executes task 103 for User 3.

taskManager.rmv(101); // Removes task 101 from the system.

taskManager.add(5, 105, 15); // Adds task 105 with priority 15 for User 5.

taskManager.execTop(); // return 5. Executes task 105 for User 5.

Constraints:

1 <= tasks.length <= 105

0 <= userId <= 105

0 <= taskId <= 105

0 <= priority <= 109

0 <= newPriority <= 109

At most 2 \* 105 calls will be made in total to add, edit, rmv, and execTop methods.

The input is generated such that taskId will be valid.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class TaskManager {

unordered\_map<int, pair<int,int>> taskMap;

// taskId -> {userId, priority}

// Max-heap by priority, then by taskId

priority\_queue<pair<pair<int,int>, int>> pq;

// {{priority, taskId}, userId}

public:

// Constructor

TaskManager(vector<vector<int>>& tasks) {

for (auto &t : tasks) {

int userId = t[0], taskId = t[1], priority = t[2];

taskMap[taskId] = {userId, priority};

pq.push({{priority, taskId}, userId});

}

}

void add(int userId, int taskId, int priority) {

taskMap[taskId] = {userId, priority};

pq.push({{priority, taskId}, userId});

}

void edit(int taskId, int newPriority) {

if (taskMap.find(taskId) != taskMap.end()) {

int userId = taskMap[taskId].first;

taskMap[taskId] = {userId, newPriority};

pq.push({{newPriority, taskId}, userId});

}

}

void rmv(int taskId) {

taskMap.erase(taskId); // lazy delete

}

int execTop() {

while (!pq.empty()) {

auto [priTask, userId] = pq.top();

int priority = priTask.first;

int taskId = priTask.second;

pq.pop();

if (taskMap.find(taskId) != taskMap.end() && taskMap[taskId].second == priority) {

taskMap.erase(taskId);

return userId;

}

}

return -1;

}

};

**Question-49:**

A spreadsheet is a grid with 26 columns (labeled from 'A' to 'Z') and a given number of rows. Each cell in the spreadsheet can hold an integer value between 0 and 105.

Implement the Spreadsheet class:

Spreadsheet(int rows) Initializes a spreadsheet with 26 columns (labeled 'A' to 'Z') and the specified number of rows. All cells are initially set to 0.

void setCell(String cell, int value) Sets the value of the specified cell. The cell reference is provided in the format "AX" (e.g., "A1", "B10"), where the letter represents the column (from 'A' to 'Z') and the number represents a 1-indexed row.

void resetCell(String cell) Resets the specified cell to 0.

int getValue(String formula) Evaluates a formula of the form "=X+Y", where X and Y are either cell references or non-negative integers, and returns the computed sum.

Note: If getValue references a cell that has not been explicitly set using setCell, its value is considered 0.

Example 1:

Input:

["Spreadsheet", "getValue", "setCell", "getValue", "setCell", "getValue", "resetCell", "getValue"]

[[3], ["=5+7"], ["A1", 10], ["=A1+6"], ["B2", 15], ["=A1+B2"], ["A1"], ["=A1+B2"]]

Output:

[null, 12, null, 16, null, 25, null, 15]

Explanation

Spreadsheet spreadsheet = new Spreadsheet(3); // Initializes a spreadsheet with 3 rows and 26 columns

spreadsheet.getValue("=5+7"); // returns 12 (5+7)

spreadsheet.setCell("A1", 10); // sets A1 to 10

spreadsheet.getValue("=A1+6"); // returns 16 (10+6)

spreadsheet.setCell("B2", 15); // sets B2 to 15

spreadsheet.getValue("=A1+B2"); // returns 25 (10+15)

spreadsheet.resetCell("A1"); // resets A1 to 0

spreadsheet.getValue("=A1+B2"); // returns 15 (0+15)

Constraints:

1 <= rows <= 103

0 <= value <= 105

The formula is always in the format "=X+Y", where X and Y are either valid cell references or non-negative integers with values less than or equal to 105.

Each cell reference consists of a capital letter from 'A' to 'Z' followed by a row number between 1 and rows.

At most 104 calls will be made in total to setCell, resetCell, and getValue.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Spreadsheet {

private:

int rows, cols;

vector<vector<int>> grid;

pair<int, int> parseCell(const string &cell) {

// "A1" -> (rowIndex, colIndex)

int col = cell[0] - 'A';

int row = stoi(cell.substr(1)) - 1; // 1-indexed -> 0-indexed

return {row, col};

}

int getCellValue(const string &token) {

if (isalpha(token[0])) { // token is a cell reference

auto [r, c] = parseCell(token);

return grid[r][c];

} else { // token is an integer

return stoi(token);

}

}

public:

Spreadsheet(int r) {

rows = r;

cols = 26;

grid.assign(rows, vector<int>(cols, 0)); // initialize all cells to 0

}

void setCell(string cell, int value) {

auto [r, c] = parseCell(cell);

grid[r][c] = value;

}

void resetCell(string cell) {

auto [r, c] = parseCell(cell);

grid[r][c] = 0;

}

int getValue(string formula) {

// formula is always of type "=X+Y"

string expr = formula.substr(1); // remove '='

size\_t pos = expr.find('+');

string left = expr.substr(0, pos);

string right = expr.substr(pos + 1);

int val1 = getCellValue(left);

int val2 = getCellValue(right);

return val1 + val2;

}

};

/\*\*

\* Your Spreadsheet object will be instantiated and called as such:

\* Spreadsheet\* obj = new Spreadsheet(rows);

\* obj->setCell(cell,value);

\* obj->resetCell(cell);

\* int param\_3 = obj->getValue(formula);

\*/

**Question-50:**

You are given an integer n.

We need to group the numbers from 1 to n according to the sum of its digits. For example, the numbers 14 and 5 belong to the **same** group, whereas 13 and 3 belong to **different** groups.

Return the number of groups that have the largest size, i.e. the **maximum** number of elements.

**Example 1:**

**Input:** n = 13

**Output:** 4

**Explanation:** There are 9 groups in total, they are grouped according sum of its digits of numbers from 1 to 13:

[1,10], [2,11], [3,12], [4,13], [5], [6], [7], [8], [9].

There are 4 groups with largest size.

**Example 2:**

**Input:** n = 2

**Output:** 2

**Explanation:** There are 2 groups [1], [2] of size 1.

**Constraints:**

* 1 <= n <= 104

**Solution:**

class Solution {

public:

int countLargestGroup(int n) {

unordered\_map<int,int> freq;

// Step 1: Count numbers in each digit-sum group

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

int sum = digitSum(i);

freq[sum]++;

}

// Step 2: Find maximum group size

int maxSize = 0;

for (auto &p : freq) {

maxSize = max(maxSize, p.second);

}

// Step 3: Count groups with max size

int count = 0;

for (auto &p : freq) {

if (p.second == maxSize) count++;

}

return count;

}

private:

int digitSum(int x) {

int s = 0;

while (x > 0) {

s += x % 10;

x /= 10;

}

return s;

}

};

Question-51

Design a data structure that can efficiently manage data packets in a network router. Each data packet consists of the following attributes:

source: A unique identifier for the machine that generated the packet.

destination: A unique identifier for the target machine.

timestamp: The time at which the packet arrived at the router.

Implement the Router class:

Router(int memoryLimit): Initializes the Router object with a fixed memory limit.

memoryLimit is the maximum number of packets the router can store at any given time.

If adding a new packet would exceed this limit, the oldest packet must be removed to free up space.

bool addPacket(int source, int destination, int timestamp): Adds a packet with the given attributes to the router.

A packet is considered a duplicate if another packet with the same source, destination, and timestamp already exists in the router.

Return true if the packet is successfully added (i.e., it is not a duplicate); otherwise return false.

int[] forwardPacket(): Forwards the next packet in FIFO (First In First Out) order.

Remove the packet from storage.

Return the packet as an array [source, destination, timestamp].

If there are no packets to forward, return an empty array.

int getCount(int destination, int startTime, int endTime):

Returns the number of packets currently stored in the router (i.e., not yet forwarded) that have the specified destination and have timestamps in the inclusive range [startTime, endTime].

Note that queries for addPacket will be made in increasing order of timestamp.

Example 1:

Input:

["Router", "addPacket", "addPacket", "addPacket", "addPacket", "addPacket", "forwardPacket", "addPacket", "getCount"]

[[3], [1, 4, 90], [2, 5, 90], [1, 4, 90], [3, 5, 95], [4, 5, 105], [], [5, 2, 110], [5, 100, 110]]

Output:

[null, true, true, false, true, true, [2, 5, 90], true, 1]

Explanation

Router router = new Router(3); // Initialize Router with memoryLimit of 3.

router.addPacket(1, 4, 90); // Packet is added. Return True.

router.addPacket(2, 5, 90); // Packet is added. Return True.

router.addPacket(1, 4, 90); // This is a duplicate packet. Return False.

router.addPacket(3, 5, 95); // Packet is added. Return True

router.addPacket(4, 5, 105); // Packet is added, [1, 4, 90] is removed as number of packets exceeds memoryLimit. Return True.

router.forwardPacket(); // Return [2, 5, 90] and remove it from router.

router.addPacket(5, 2, 110); // Packet is added. Return True.

router.getCount(5, 100, 110); // The only packet with destination 5 and timestamp in the inclusive range [100, 110] is [4, 5, 105]. Return 1.

Example 2:

Input:

["Router", "addPacket", "forwardPacket", "forwardPacket"]

[[2], [7, 4, 90], [], []]

Output:

[null, true, [7, 4, 90], []]

Explanation

Router router = new Router(2); // Initialize Router with memoryLimit of 2.

router.addPacket(7, 4, 90); // Return True.

router.forwardPacket(); // Return [7, 4, 90].

router.forwardPacket(); // There are no packets left, return [].

Constraints:

2 <= memoryLimit <= 105

1 <= source, destination <= 2 \* 105

1 <= timestamp <= 109

1 <= startTime <= endTime <= 109

At most 105 calls will be made to addPacket, forwardPacket, and getCount methods altogether.

queries for addPacket will be made in increasing order of timestamp.

class Router {

public:

Router(int memoryLimit) {

}

bool addPacket(int source, int destination, int timestamp) {

}

vector<int> forwardPacket() {

}

int getCount(int destination, int startTime, int endTime) {

}

};

/\*\*

\* Your Router object will be instantiated and called as such:

\* Router\* obj = new Router(memoryLimit);

\* bool param\_1 = obj->addPacket(source,destination,timestamp);

\* vector<int> param\_2 = obj->forwardPacket();

\* int param\_3 = obj->getCount(destination,startTime,endTime);

\*/

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Router {

private:

int memoryLimit;

deque<array<int,3>> buffer; // {source, dest, timestamp}

unordered\_set<long long> seen; // for duplicates

unordered\_map<int, vector<int>> destMap; // destination -> timestamps

long long makeKey(int s, int d, int t) {

// encode triple into long long

return ((long long)s << 40) | ((long long)d << 20) | t;

}

void removeOldest() {

if (buffer.size() <= memoryLimit) return;

auto pkt = buffer.front();

buffer.pop\_front();

long long key = makeKey(pkt[0], pkt[1], pkt[2]);

seen.erase(key);

// remove from destMap

auto &vec = destMap[pkt[1]];

// since timestamps are unique per packet insertion order,

// we can pop from front if matches

if (!vec.empty() && vec[0] == pkt[2]) {

vec.erase(vec.begin());

} else {

// fallback (should be rare, but for safety)

auto it = lower\_bound(vec.begin(), vec.end(), pkt[2]);

if (it != vec.end() && \*it == pkt[2]) vec.erase(it);

}

}

public:

Router(int memoryLimit) {

this->memoryLimit = memoryLimit;

}

bool addPacket(int source, int destination, int timestamp) {

long long key = makeKey(source, destination, timestamp);

if (seen.count(key)) return false; // duplicate

buffer.push\_back({source, destination, timestamp});

seen.insert(key);

destMap[destination].push\_back(timestamp); // timestamps always increasing

// enforce memory limit

if ((int)buffer.size() > memoryLimit) {

removeOldest();

}

return true;

}

vector<int> forwardPacket() {

if (buffer.empty()) return {};

auto pkt = buffer.front();

buffer.pop\_front();

long long key = makeKey(pkt[0], pkt[1], pkt[2]);

seen.erase(key);

// remove from destMap

auto &vec = destMap[pkt[1]];

auto it = lower\_bound(vec.begin(), vec.end(), pkt[2]);

if (it != vec.end() && \*it == pkt[2]) vec.erase(it);

return {pkt[0], pkt[1], pkt[2]};

}

int getCount(int destination, int startTime, int endTime) {

if (!destMap.count(destination)) return 0;

auto &vec = destMap[destination];

auto lo = lower\_bound(vec.begin(), vec.end(), startTime);

auto hi = upper\_bound(vec.begin(), vec.end(), endTime);

return (int)(hi - lo);

}

};

**Question-52:**

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

* countAndSay(1) = "1"
* countAndSay(n) is the run-length encoding of countAndSay(n - 1).

[Run-length encoding](http://en.wikipedia.org/wiki/Run-length_encoding) (RLE) is a string compression method that works by replacing consecutive identical characters (repeated 2 or more times) with the concatenation of the character and the number marking the count of the characters (length of the run). For example, to compress the string "3322251" we replace "33" with "23", replace "222" with "32", replace "5" with "15" and replace "1" with "11". Thus the compressed string becomes "23321511".

Given a positive integer n, return *the*nth*element of the****count-and-say****sequence*.

**Example 1:**

**Input:** n = 4

**Output:** "1211"

**Explanation:**

countAndSay(1) = "1"

countAndSay(2) = RLE of "1" = "11"

countAndSay(3) = RLE of "11" = "21"

countAndSay(4) = RLE of "21" = "1211"

**Example 2:**

**Input:** n = 1

**Output:** "1"

**Explanation:**

This is the base case.

**Constraints:**

* 1 <= n <= 30

**Solution:**

class Solution {

public:

string countAndSay(int n) {

string result = "1";

for (int i = 2; i <= n; i++) {

string current = "";

int count = 1;

for (int j = 1; j <= (int)result.size(); j++) {

if (j < result.size() && result[j] == result[j-1]) {

count++;

} else {

current += to\_string(count);

current.push\_back(result[j-1]);

count = 1;

}

}

result = current;

}

return result;

}

};

**Question-53**

Given an array of **distinct** integers candidates and a target integer target, return *a list of all****unique combinations****of*candidates*where the chosen numbers sum to*target*.* You may return the combinations in **any order**.

The **same** number may be chosen from candidates an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

**Example 1:**

**Input:** candidates = [2,3,6,7], target = 7

**Output:** [[2,2,3],[7]]

**Explanation:**

2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

**Example 2:**

**Input:** candidates = [2,3,5], target = 8

**Output:** [[2,2,2,2],[2,3,3],[3,5]]

**Example 3:**

**Input:** candidates = [2], target = 1

**Output:** []

**Constraints:**

* 1 <= candidates.length <= 30
* 2 <= candidates[i] <= 40
* All elements of candidates are **distinct**.
* 1 <= target <= 40

**Solution:**

class Solution {

public:

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

vector<vector<int>> result;

vector<int> current;

backtrack(candidates, target, 0, current, result);

return result;

}

private:

void backtrack(vector<int>& candidates, int target, int start,

vector<int>& current, vector<vector<int>>& result) {

if (target == 0) {

result.push\_back(current);

return;

}

if (target < 0) return; // overshoot, stop

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

current.push\_back(candidates[i]);

// allow reuse of same element, so pass `i` (not i+1)

backtrack(candidates, target - candidates[i], i, current, result);

current.pop\_back(); // backtrack

}

}

};

**Question: 54**

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used **once** in the combination.

**Note:** The solution set must not contain duplicate combinations.

**Example 1:**

**Input:** candidates = [10,1,2,7,6,1,5], target = 8

**Output:**

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

**Example 2:**

**Input:** candidates = [2,5,2,1,2], target = 5

**Output:**

[

[1,2,2],

[5]

]

**Constraints:**

* 1 <= candidates.length <= 100
* 1 <= candidates[i] <= 50
* 1 <= target <= 30

**Solution:**

class Solution {

public:

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

sort(candidates.begin(), candidates.end()); // sort to handle duplicates

vector<vector<int>> result;

vector<int> current;

backtrack(candidates, target, 0, current, result);

return result;

}

private:

void backtrack(vector<int>& candidates, int target, int start,

vector<int>& current, vector<vector<int>>& result) {

if (target == 0) {

result.push\_back(current);

return;

}

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

// skip duplicates at the same recursion level

if (i > start && candidates[i] == candidates[i-1]) continue;

if (candidates[i] > target) break; // pruning

current.push\_back(candidates[i]);

backtrack(candidates, target - candidates[i], i + 1, current, result);

current.pop\_back();

}

}

};

**Question-55**

You have a movie renting company consisting of n shops. You want to implement a renting system that supports searching for, booking, and returning movies. The system should also support generating a report of the currently rented movies.

Each movie is given as a 2D integer array entries where entries[i] = [shopi, moviei, pricei] indicates that there is a copy of movie moviei at shop shopi with a rental price of pricei. Each shop carries at most one copy of a movie moviei.

The system should support the following functions:

Search: Finds the cheapest 5 shops that have an unrented copy of a given movie. The shops should be sorted by price in ascending order, and in case of a tie, the one with the smaller shopi should appear first. If there are less than 5 matching shops, then all of them should be returned. If no shop has an unrented copy, then an empty list should be returned.

Rent: Rents an unrented copy of a given movie from a given shop.

Drop: Drops off a previously rented copy of a given movie at a given shop.

Report: Returns the cheapest 5 rented movies (possibly of the same movie ID) as a 2D list res where res[j] = [shopj, moviej] describes that the jth cheapest rented movie moviej was rented from the shop shopj. The movies in res should be sorted by price in ascending order, and in case of a tie, the one with the smaller shopj should appear first, and if there is still tie, the one with the smaller moviej should appear first. If there are fewer than 5 rented movies, then all of them should be returned. If no movies are currently being rented, then an empty list should be returned.

Implement the MovieRentingSystem class:

MovieRentingSystem(int n, int[][] entries) Initializes the MovieRentingSystem object with n shops and the movies in entries.

List<Integer> search(int movie) Returns a list of shops that have an unrented copy of the given movie as described above.

void rent(int shop, int movie) Rents the given movie from the given shop.

void drop(int shop, int movie) Drops off a previously rented movie at the given shop.

List<List<Integer>> report() Returns a list of cheapest rented movies as described above.

Note: The test cases will be generated such that rent will only be called if the shop has an unrented copy of the movie, and drop will only be called if the shop had previously rented out the movie.

Example 1:

Input

["MovieRentingSystem", "search", "rent", "rent", "report", "drop", "search"]

[[3, [[0, 1, 5], [0, 2, 6], [0, 3, 7], [1, 1, 4], [1, 2, 7], [2, 1, 5]]], [1], [0, 1], [1, 2], [], [1, 2], [2]]

Output

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

Explanation

MovieRentingSystem movieRentingSystem = new MovieRentingSystem(3, [[0, 1, 5], [0, 2, 6], [0, 3, 7], [1, 1, 4], [1, 2, 7], [2, 1, 5]]);

movieRentingSystem.search(1); // return [1, 0, 2], Movies of ID 1 are unrented at shops 1, 0, and 2. Shop 1 is cheapest; shop 0 and 2 are the same price, so order by shop number.

movieRentingSystem.rent(0, 1); // Rent movie 1 from shop 0. Unrented movies at shop 0 are now [2,3].

movieRentingSystem.rent(1, 2); // Rent movie 2 from shop 1. Unrented movies at shop 1 are now [1].

movieRentingSystem.report(); // return [[0, 1], [1, 2]]. Movie 1 from shop 0 is cheapest, followed by movie 2 from shop 1.

movieRentingSystem.drop(1, 2); // Drop off movie 2 at shop 1. Unrented movies at shop 1 are now [1,2].

movieRentingSystem.search(2); // return [0, 1]. Movies of ID 2 are unrented at shops 0 and 1. Shop 0 is cheapest, followed by shop 1.

Constraints:

1 <= n <= 3 \* 105

1 <= entries.length <= 105

0 <= shopi < n

1 <= moviei, pricei <= 104

Each shop carries at most one copy of a movie moviei.

At most 105 calls in total will be made to search, rent, drop and report.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class MovieRentingSystem {

unordered\_map<int, unordered\_map<int,int>> priceMap;

unordered\_map<int, set<pair<int,int>>> unrented; // movie -> {price, shop}

set<tuple<int,int,int>> rented; // {price, shop, movie}

public:

MovieRentingSystem(int n, vector<vector<int>>& entries) {

for (auto &e : entries) {

int shop = e[0], movie = e[1], price = e[2];

priceMap[shop][movie] = price;

unrented[movie].insert({price, shop});

}

}

vector<int> search(int movie) {

vector<int> res;

if (unrented.count(movie)) {

for (auto &p : unrented[movie]) {

res.push\_back(p.second);

if (res.size() == 5) break;

}

}

return res;

}

void rent(int shop, int movie) {

int price = priceMap[shop][movie];

unrented[movie].erase({price, shop});

rented.insert({price, shop, movie});

}

void drop(int shop, int movie) {

int price = priceMap[shop][movie];

rented.erase({price, shop, movie});

unrented[movie].insert({price, shop});

}

vector<vector<int>> report() {

vector<vector<int>> res;

int cnt = 0;

for (auto &t : rented) {

int price, shop, movie;

tie(price, shop, movie) = t;

res.push\_back({shop, movie});

cnt++;

if (cnt == 5) break;

}

return res;

}

};

**Question-56**

Given two version strings, version1 and version2, compare them. A version string consists of revisions separated by dots '.'. The value of the revision is its integer conversion ignoring leading zeros.

To compare version strings, compare their revision values in left-to-right order. If one of the version strings has fewer revisions, treat the missing revision values as 0.

Return the following:

If version1 < version2, return -1.

If version1 > version2, return 1.

Otherwise, return 0.

Example 1:

Input: version1 = "1.2", version2 = "1.10"

Output: -1

Explanation:

version1's second revision is "2" and version2's second revision is "10": 2 < 10, so version1 < version2.

Example 2:

Input: version1 = "1.01", version2 = "1.001"

Output: 0

Explanation:

Ignoring leading zeroes, both "01" and "001" represent the same integer "1".

Example 3:

Input: version1 = "1.0", version2 = "1.0.0.0"

Output: 0

Explanation:

version1 has less revisions, which means every missing revision are treated as "0".

Constraints:

1 <= version1.length, version2.length <= 500

version1 and version2 only contain digits and '.'.

version1 and version2 are valid version numbers.

All the given revisions in version1 and version2 can be stored in a 32-bit integer.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int compareVersion(string version1, string version2) {

vector<int> v1 = split(version1);

vector<int> v2 = split(version2);

int n = max(v1.size(), v2.size());

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

int num1 = i < v1.size() ? v1[i] : 0;

int num2 = i < v2.size() ? v2[i] : 0;

if (num1 < num2) return -1;

if (num1 > num2) return 1;

}

return 0;

}

private:

vector<int> split(string &s) {

vector<int> nums;

string token;

stringstream ss(s);

while (getline(ss, token, '.')) {

nums.push\_back(stoi(token)); // stoi handles leading zeros

}

return nums;

}

};

**Question-57**

Given two integers representing the numerator and denominator of a fraction, return *the fraction in string format*.

If the fractional part is repeating, enclose the repeating part in parentheses.

If multiple answers are possible, return **any of them**.

It is **guaranteed** that the length of the answer string is less than 104 for all the given inputs.

**Example 1:**

**Input:** numerator = 1, denominator = 2

**Output:** "0.5"

**Example 2:**

**Input:** numerator = 2, denominator = 1

**Output:** "2"

**Example 3:**

**Input:** numerator = 4, denominator = 333

**Output:** "0.(012)"

**Constraints:**

* -231 <= numerator, denominator <= 231 - 1
* denominator != 0

**Solution:**

class Solution {

public:

string fractionToDecimal(long long numerator, long long denominator) {

if (numerator == 0) return "0";

string result;

// Handle negative numbers

if ((numerator < 0) ^ (denominator < 0)) result += "-";

// Convert to absolute values

long long n = llabs(numerator);

long long d = llabs(denominator);

// Integer part

result += to\_string(n / d);

long long remainder = n % d;

if (remainder == 0) return result; // No fractional part

result += ".";

// Map to track remainder positions

unordered\_map<long long, int> seen;

while (remainder != 0) {

if (seen.count(remainder)) {

// Insert "(" at the first occurrence of this remainder

result.insert(seen[remainder], "(");

result += ")";

break;

}

seen[remainder] = result.size();

remainder \*= 10;

result += to\_string(remainder / d);

remainder %= d;

}

return result;

}

};

**Question-58:**

Given an integer array nums, return *the number of triplets chosen from the array that can make triangles if we take them as side lengths of a triangle*.

**Example 1:**

**Input:** nums = [2,2,3,4]

**Output:** 3

**Explanation:** Valid combinations are:

2,3,4 (using the first 2)

2,3,4 (using the second 2)

2,2,3

**Example 2:**

**Input:** nums = [4,2,3,4]

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 1000
* 0 <= nums[i] <= 1000

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int triangleNumber(vector<int>& nums) {

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

int n = nums.size();

int count = 0;

for (int k = n - 1; k >= 2; k--) {

int i = 0, j = k - 1;

while (i < j) {

if (nums[i] + nums[j] > nums[k]) {

count += (j - i);

j--;

} else {

i++;

}

}

}

return count;

}

};

**Question-59**

Given an integer array nums, return *the largest perimeter of a triangle with a non-zero area, formed from three of these lengths*. If it is impossible to form any triangle of a non-zero area, return 0.

**Example 1:**

**Input:** nums = [2,1,2]

**Output:** 5

**Explanation:** You can form a triangle with three side lengths: 1, 2, and 2.

**Example 2:**

**Input:** nums = [1,2,1,10]

**Output:** 0

**Explanation:**

You cannot use the side lengths 1, 1, and 2 to form a triangle.

You cannot use the side lengths 1, 1, and 10 to form a triangle.

You cannot use the side lengths 1, 2, and 10 to form a triangle.

As we cannot use any three side lengths to form a triangle of non-zero area, we return 0.

**Constraints:**

* 3 <= nums.length <= 104
* 1 <= nums[i] <= 106

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int largestPerimeter(vector<int>& nums) {

sort(nums.begin(), nums.end(), greater<int>()); // sort in descending order

for (int i = 0; i < nums.size() - 2; i++) {

int a = nums[i], b = nums[i+1], c = nums[i+2];

if (b + c > a) { // triangle inequality

return a + b + c;

}

}

return 0;

}

};

int main() {

Solution sol;

vector<int> nums1 = {2,1,2};

cout << sol.largestPerimeter(nums1) << endl; // Output: 5

vector<int> nums2 = {1,2,1,10};

cout << sol.largestPerimeter(nums2) << endl; // Output: 0

return 0;

}

**Question-60**

You are given two integers numBottles and numExchange.

numBottles represents the number of full water bottles that you initially have. In one operation, you can perform one of the following operations:

* Drink any number of full water bottles turning them into empty bottles.
* Exchange numExchange empty bottles with one full water bottle. Then, increase numExchange by one.

Note that you cannot exchange multiple batches of empty bottles for the same value of numExchange. For example, if numBottles == 3 and numExchange == 1, you cannot exchange 3 empty water bottles for 3 full bottles.

Return *the****maximum****number of water bottles you can drink*.

**Example 1:**

A table with numbers and symbols

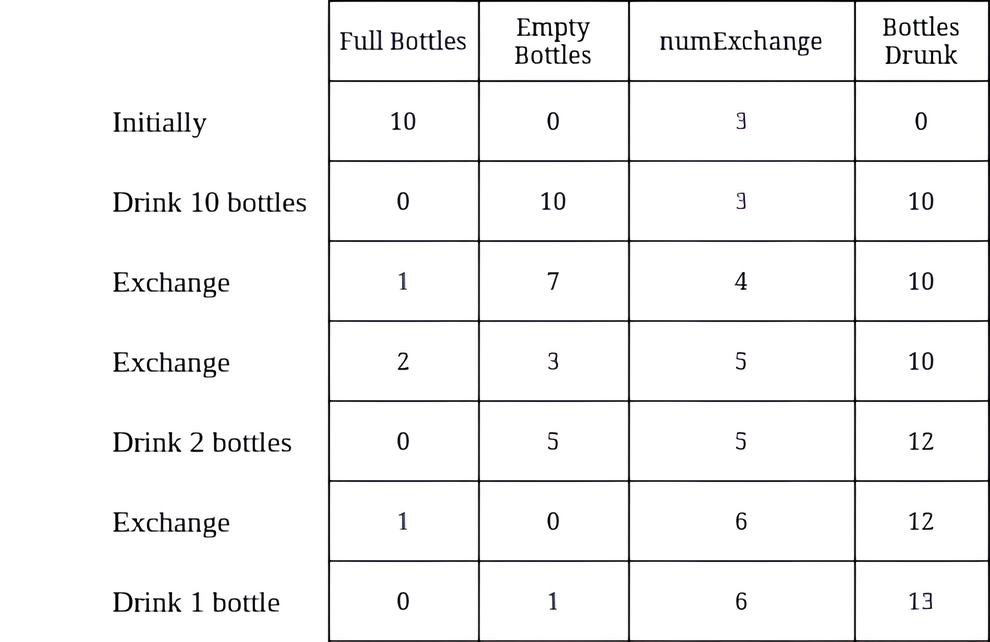
AI-generated content may be incorrect.

**Input:** numBottles = 13, numExchange = 6

**Output:** 15

**Explanation:** The table above shows the number of full water bottles, empty water bottles, the value of numExchange, and the number of bottles drunk.

**Example 2:**



**Input:** numBottles = 10, numExchange = 3

**Output:** 13

**Explanation:** The table above shows the number of full water bottles, empty water bottles, the value of numExchange, and the number of bottles drunk.

**Constraints:**

* 1 <= numBottles <= 100
* 1 <= numExchange <= 100

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int maxBottlesDrunk(int numBottles, int numExchange) {

int ans = 0;

int empty = 0;

while (numBottles > 0) {

// drink all full bottles

ans += numBottles;

empty += numBottles;

numBottles = 0;

// try to exchange

if (empty >= numExchange) {

empty -= numExchange; // use numExchange empties

numBottles = 1; // gain 1 new full bottle

numExchange++; // cost increases

}

}

return ans;

}

};

int main() {

Solution sol;

cout << sol.maxBottlesDrunk(13, 6) << endl; // Output: 15

cout << sol.maxBottlesDrunk(10, 3) << endl; // Output: 13

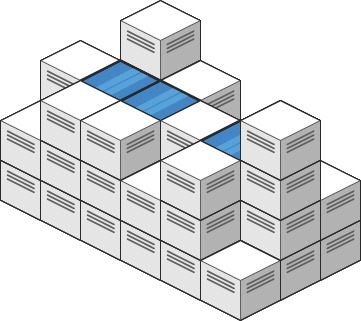
return 0;

}

**Question-61**

Given an m x n integer matrix heightMap representing the height of each unit cell in a 2D elevation map, return *the volume of water it can trap after raining*.

**Example 1:**



**Input:** heightMap = [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]

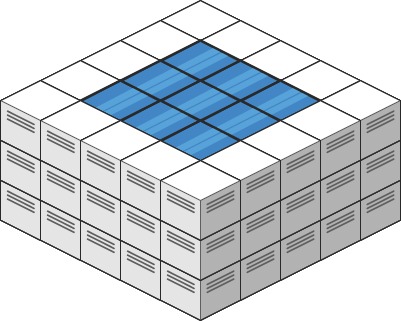
**Output:** 4

**Explanation:** After the rain, water is trapped between the blocks.

We have two small ponds 1 and 3 units trapped.

The total volume of water trapped is 4.

**Example 2:**



**Input:** heightMap = [[3,3,3,3,3],[3,2,2,2,3],[3,2,1,2,3],[3,2,2,2,3],[3,3,3,3,3]]

**Output:** 10

**Constraints:**

* m == heightMap.length
* n == heightMap[i].length
* 1 <= m, n <= 200
* 0 <= heightMap[i][j] <= 2 \* 104

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int trapRainWater(vector<vector<int>>& heightMap) {

int m = heightMap.size();

int n = heightMap[0].size();

if (m <= 2 || n <= 2) return 0; // no space to trap water

vector<vector<bool>> visited(m, vector<bool>(n, false));

priority\_queue<

pair<int, pair<int,int>>,

vector<pair<int, pair<int,int>>>,

greater<>> minHeap;

// Step 1: Push all boundary cells into heap

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

for (int j : {0, n-1}) {

minHeap.push({heightMap[i][j], {i,j}});

visited[i][j] = true;

}

}

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

for (int i : {0, m-1}) {

if (!visited[i][j]) {

minHeap.push({heightMap[i][j], {i,j}});

visited[i][j] = true;

}

}

}

// Step 2: BFS with min-heap

int trapped = 0;

vector<int> dirs = {0,1,0,-1,0};

while (!minHeap.empty()) {

auto [h, cell] = minHeap.top(); minHeap.pop();

int x = cell.first, y = cell.second;

for (int k = 0; k < 4; k++) {

int nx = x + dirs[k], ny = y + dirs[k+1];

if (nx < 0 || nx >= m || ny < 0 || ny >= n || visited[nx][ny]) continue;

visited[nx][ny] = true;

trapped += max(0, h - heightMap[nx][ny]);

minHeap.push({max(h, heightMap[nx][ny]), {nx,ny}});

}

}

return trapped;

}

};

int main() {

Solution sol;

vector<vector<int>> h1 = {{1,4,3,1,3,2},{3,2,1,3,2,4},{2,3,3,2,3,1}};

cout << sol.trapRainWater(h1) << endl; // Output: 4

vector<vector<int>> h2 = {

{3,3,3,3,3},

{3,2,2,2,3},

{3,2,1,2,3},

{3,2,2,2,3},

{3,3,3,3,3}

};

cout << sol.trapRainWater(h2) << endl; // Output: 10

}

**Question-62**

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

**Notice** that you may not slant the container.

**Example 1:**

A blue and black graph

AI-generated content may be incorrect.

**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Constraints:**

* n == height.length
* 2 <= n <= 105
* 0 <= height[i] <= 104

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int maxArea(vector<int>& height) {

int left = 0, right = height.size() - 1;

int maxArea = 0;

while (left < right) {

int h = min(height[left], height[right]);

int width = right - left;

maxArea = max(maxArea, h \* width);

// Move the smaller line inward

if (height[left] < height[right])

left++;

else

right--;

}

return maxArea;

}

};

int main() {

Solution sol;

vector<int> height1 = {1,8,6,2,5,4,8,3,7};

cout << sol.maxArea(height1) << endl; // Output: 49

vector<int> height2 = {1,1};

cout << sol.maxArea(height2) << endl; // Output: 1

}

**Question-63**

Given an unsorted integer array nums. Return the *smallest positive integer* that is *not present* in nums.

You must implement an algorithm that runs in O(n) time and uses O(1) auxiliary space.

**Example 1:**

**Input:** nums = [1,2,0]

**Output:** 3

**Explanation:** The numbers in the range [1,2] are all in the array.

**Example 2:**

**Input:** nums = [3,4,-1,1]

**Output:** 2

**Explanation:** 1 is in the array but 2 is missing.

**Example 3:**

**Input:** nums = [7,8,9,11,12]

**Output:** 1

**Explanation:** The smallest positive integer 1 is missing.

**Constraints:**

* 1 <= nums.length <= 105
* -231 <= nums[i] <= 231 - 1

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int firstMissingPositive(vector<int>& nums) {

int n = nums.size();

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

// Place nums[i] at the correct position

while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i]) {

swap(nums[i], nums[nums[i] - 1]);

}

}

// Find the first index where nums[i] != i+1

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

if (nums[i] != i + 1)

return i + 1;

}

// If all numbers 1..n are present

return n + 1;

}

};

int main() {

Solution sol;

vector<int> nums1 = {1, 2, 0};

cout << sol.firstMissingPositive(nums1) << endl; // Output: 3

vector<int> nums2 = {3, 4, -1, 1};

cout << sol.firstMissingPositive(nums2) << endl; // Output: 2

vector<int> nums3 = {7, 8, 9, 11, 12};

cout << sol.firstMissingPositive(nums3) << endl; // Output: 1

}

**Question-64**

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:

Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]

Output: 9

Constraints:

n == height.length

1 <= n <= 2 \* 104

0 <= height[i] <= 105

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int trap(vector<int>& height) {

int n = height.size();

int left = 0, right = n - 1;

int leftMax = 0, rightMax = 0;

int water = 0;

while (left < right) {

if (height[left] < height[right]) {

if (height[left] >= leftMax)

leftMax = height[left];

else

water += leftMax - height[left];

left++;

} else {

if (height[right] >= rightMax)

rightMax = height[right];

else

water += rightMax - height[right];

right--;

}

}

return water;

}

};

int main() {

Solution sol;

vector<int> height1 = {0,1,0,2,1,0,1,3,2,1,2,1};

cout << sol.trap(height1) << endl; // Output: 6

vector<int> height2 = {4,2,0,3,2,5};

cout << sol.trap(height2) << endl; // Output: 9

}

**Question-65**

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

Note: You must not use any built-in BigInteger library or convert the inputs to integer directly.

Example 1:

Input: num1 = "2", num2 = "3"

Output: "6”

Example 2:

Input: num1 = "123", num2 = "456"

Output: "56088"

Constraints:

1 <= num1.length, num2.length <= 200

num1 and num2 consist of digits only.

Both num1 and num2 do not contain any leading zero, except the number 0 itself.

**Solution:**

class Solution {

public:

string multiply(string num1, string num2) {

// If either number is 0, product is 0

if (num1 == "0" || num2 == "0")

return "0";

int n1 = num1.size(), n2 = num2.size();

vector<int> res(n1 + n2, 0);

// Reverse both strings to multiply from least significant digit

reverse(num1.begin(), num1.end());

reverse(num2.begin(), num2.end());

// Multiply each digit and add at proper position

for (int i = 0; i < n1; i++) {

for (int j = 0; j < n2; j++) {

int mul = (num1[i] - '0') \* (num2[j] - '0');

res[i + j] += mul;

// Handle carry

res[i + j + 1] += res[i + j] / 10;

res[i + j] %= 10;

}

}

// Remove leading zeros

while (res.size() > 1 && res.back() == 0)

res.pop\_back();

// Convert result array to string

string result;

for (int i = res.size() - 1; i >= 0; i--)

result.push\_back(res[i] + '0');

return result;

}

};

**Question-66**

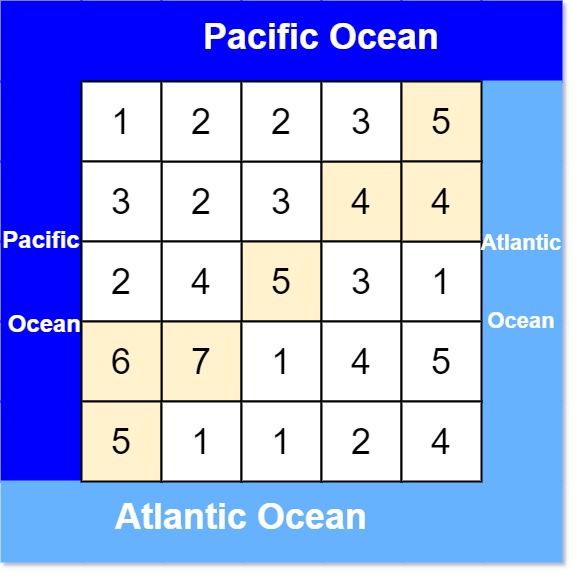
There is an m x n rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the **height above sea level** of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a****2D list****of grid coordinates*result*where*result[i] = [ri, ci]*denotes that rain water can flow from cell*(ri, ci)*to****both****the Pacific and Atlantic oceans*.

**Example 1:**



**Input:** heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]

**Output:** [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

**Explanation:** The following cells can flow to the Pacific and Atlantic oceans, as shown below:

[0,4]: [0,4] -> Pacific Ocean

  [0,4] -> Atlantic Ocean

[1,3]: [1,3] -> [0,3] -> Pacific Ocean

  [1,3] -> [1,4] -> Atlantic Ocean

[1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean

  [1,4] -> Atlantic Ocean

[2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean

  [2,2] -> [2,3] -> [2,4] -> Atlantic Ocean

[3,0]: [3,0] -> Pacific Ocean

  [3,0] -> [4,0] -> Atlantic Ocean

[3,1]: [3,1] -> [3,0] -> Pacific Ocean

  [3,1] -> [4,1] -> Atlantic Ocean

[4,0]: [4,0] -> Pacific Ocean

[4,0] -> Atlantic Ocean

Note that there are other possible paths for these cells to flow to the Pacific and Atlantic oceans.

**Example 2:**

**Input:** heights = [[1]]

**Output:** [[0,0]]

**Explanation:** The water can flow from the only cell to the Pacific and Atlantic oceans.

**Constraints:**

* m == heights.length
* n == heights[r].length
* 1 <= m, n <= 200
* 0 <= heights[r][c] <= 105

**Solution:**

class Solution {

public:

int m, n;

vector<vector<int>> directions = {{1,0},{-1,0},{0,1},{0,-1}};

void dfs(vector<vector<int>>& heights, vector<vector<bool>>& visited, int i, int j) {

visited[i][j] = true;

for (auto& dir : directions) {

int x = i + dir[0];

int y = j + dir[1];

// Move only if inside bounds, not visited, and height is >= previous cell

if (x >= 0 && y >= 0 && x < m && y < n &&

!visited[x][y] && heights[x][y] >= heights[i][j]) {

dfs(heights, visited, x, y);

}

}

}

vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {

m = heights.size();

n = heights[0].size();

vector<vector<bool>> pacific(m, vector<bool>(n, false));

vector<vector<bool>> atlantic(m, vector<bool>(n, false));

// Pacific: top row & left column

for (int i = 0; i < m; i++) dfs(heights, pacific, i, 0);

for (int j = 0; j < n; j++) dfs(heights, pacific, 0, j);

// Atlantic: bottom row & right column

for (int i = 0; i < m; i++) dfs(heights, atlantic, i, n - 1);

for (int j = 0; j < n; j++) dfs(heights, atlantic, m - 1, j);

// Collect cells reachable by both oceans

vector<vector<int>> result;

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

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

if (pacific[i][j] && atlantic[i][j]) {

result.push\_back({i, j});

}

}

}

return result;

}

};

**Question-67**

ou are given an n x n integer matrix grid where each value grid[i][j] represents the elevation at that point (i, j).

It starts raining, and water gradually rises over time. At time t, the water level is t, meaning **any** cell with elevation less than equal to t is submerged or reachable.

You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return *the minimum time until you can reach the bottom right square*(n - 1, n - 1)*if you start at the top left square*(0, 0).

**Example 1:**

A grid of numbers and symbols

AI-generated content may be incorrect.

**Input:** grid = [[0,2],[1,3]]

**Output:** 3

Explanation:

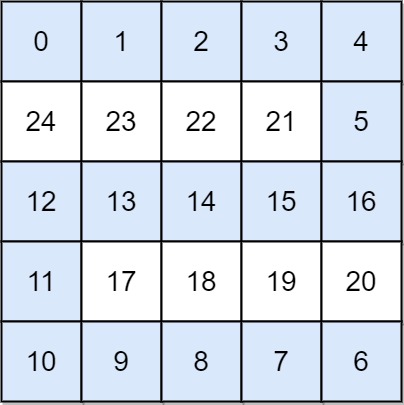
At time 0, you are in grid location (0, 0).

You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than t = 0.

You cannot reach point (1, 1) until time 3.

When the depth of water is 3, we can swim anywhere inside the grid.

**Example 2:**



**Input:** grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

**Output:** 16

**Explanation:** The final route is shown.

We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

**Constraints:**

* n == grid.length
* n == grid[i].length
* 1 <= n <= 50
* 0 <= grid[i][j] < n2
* Each value grid[i][j] is **unique**.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

int swimInWater(vector<vector<int>>& grid) {

int n = grid.size();

vector<vector<bool>> visited(n, vector<bool>(n, false));

priority\_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;

// {time, row, col}

pq.push({grid[0][0], 0, 0});

visited[0][0] = true;

int dirs[4][2] = {{1,0},{-1,0},{0,1},{0,-1}};

int res = 0;

while (!pq.empty()) {

auto cur = pq.top(); pq.pop();

int t = cur[0], x = cur[1], y = cur[2];

res = max(res, t);

// If we reached bottom-right

if (x == n - 1 && y == n - 1) return res;

for (auto& d : dirs) {

int nx = x + d[0], ny = y + d[1];

if (nx >= 0 && ny >= 0 && nx < n && ny < n && !visited[nx][ny]) {

visited[nx][ny] = true;

pq.push({grid[nx][ny], nx, ny});

}

}

}

return -1; // Should never happen

}

};

**Question-68**

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

* '?' Matches any single character.
* '\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "\*"

**Output:** true

**Explanation:** '\*' matches any sequence.

**Example 3:**

**Input:** s = "cb", p = "?a"

**Output:** false

**Explanation:** '?' matches 'c', but the second letter is 'a', which does not match 'b'.

**Constraints:**

* 0 <= s.length, p.length <= 2000
* s contains only lowercase English letters.
* p contains only lowercase English letters, '?' or '\*'.

**Solution:**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

bool isMatch(string s, string p) {

int m = s.size(), n = p.size();

vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

dp[0][0] = true;

// Handle leading '\*' in pattern

for (int j = 1; j <= n; ++j)

if (p[j - 1] == '\*')

dp[0][j] = dp[0][j - 1];

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

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

if (p[j - 1] == s[i - 1] || p[j - 1] == '?') {

dp[i][j] = dp[i - 1][j - 1];

}

else if (p[j - 1] == '\*') {

dp[i][j] = dp[i][j - 1] || dp[i - 1][j];

}

}

}

return dp[m][n];

}

};

**Question-69**

You are given a **0-indexed** array of integers nums of length n. You are initially positioned at index 0.

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at index i, you can jump to any index (i + j) where:

* 0 <= j <= nums[i] and
* i + j < n

Return *the minimum number of jumps to reach index*n - 1. The test cases are generated such that you can reach index n - 1.

**Example 1:**

**Input:** nums = [2,3,1,1,4]

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** nums = [2,3,0,1,4]

**Output:** 2

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 1000
* It's guaranteed that you can reach nums[n - 1].

**Solution:**

class Solution {

public:

int jump(vector<int>& nums) {

int n = nums.size();

if (n <= 1) return 0;

int jumps = 0;

int farthest = 0;

int end = 0;

// Traverse the array (except the last index)

for (int i = 0; i < n - 1; i++) {

farthest = max(farthest, i + nums[i]);

// If we have reached the end of the current jump range

if (i == end) {

jumps++;

end = farthest;

// If we can reach or go beyond the last index

if (end >= n - 1) break;

}

}

return jumps;

}

};

**Question-70**

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in **any order**.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

**Example 2:**

**Input:** nums = [0,1]

**Output:** [[0,1],[1,0]]

**Example 3:**

**Input:** nums = [1]

**Output:** [[1]]

**Constraints:**

* 1 <= nums.length <= 6
* -10 <= nums[i] <= 10
* All the integers of nums are **unique**.

**Solution:**

class Solution {

public:

void backtrack(vector<int>& nums, vector<vector<int>>& result, int start) {

// Base case: if we've fixed all positions, store the permutation

if (start == nums.size()) {

result.push\_back(nums);

return;

}

// Try placing each number at the current 'start' position

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

swap(nums[start], nums[i]); // choose

backtrack(nums, result, start + 1); // explore

swap(nums[start], nums[i]); // un-choose (backtrack)

}

}

vector<vector<int>> permute(vector<int>& nums) {

vector<vector<int>> result;

backtrack(nums, result, 0);

return result;

}

};

**Question-71**

Given a collection of numbers, nums, that might contain duplicates, return *all possible unique permutations****in any order****.*

**Example 1:**

**Input:** nums = [1,1,2]

**Output:**

[[1,1,2],

[1,2,1],

[2,1,1]]

**Example 2:**

**Input:** nums = [1,2,3]

**Output:** [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

**Constraints:**

* 1 <= nums.length <= 8
* -10 <= nums[i] <= 10

**Solution:**

class Solution {

public:

void backtrack(vector<int>& nums, vector<vector<int>>& result, vector<int>& curr, vector<bool>& used) {

if (curr.size() == nums.size()) {

result.push\_back(curr);

return;

}

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

// Skip used numbers

if (used[i]) continue;

// Skip duplicates — only use the first unused occurrence

if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) continue;

used[i] = true;

curr.push\_back(nums[i]);

backtrack(nums, result, curr, used);

curr.pop\_back();

used[i] = false;

}

}

vector<vector<int>> permuteUnique(vector<int>& nums) {

vector<vector<int>> result;

vector<int> curr;

vector<bool> used(nums.size(), false);

sort(nums.begin(), nums.end()); // Sort first to handle duplicates

backtrack(nums, result, curr, used);

return result;

}

};

**Question-72**

You are given an n x n 2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[7,4,1],[8,5,2],[9,6,3]]

**Example 2:**



**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

**Output:** [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

**Constraints:**

* n == matrix.length == matrix[i].length
* 1 <= n <= 20
* -1000 <= matrix[i][j] <= 1000

**Solution:**

class Solution {

public:

void rotate(vector<vector<int>>& matrix) {

int n = matrix.size();

// Step 1: Transpose the matrix (swap across diagonal)

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

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

swap(matrix[i][j], matrix[j][i]);

}

}

// Step 2: Reverse each row

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

reverse(matrix[i].begin(), matrix[i].end());

}

}

};

**Question-73**

Given an array of strings strs, group the anagrams together. You can return the answer in **any order**.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Explanation:**

* There is no string in strs that can be rearranged to form "bat".
* The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
* The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

**Solution:**

class Solution {

public:

vector<vector<string>> groupAnagrams(vector<string>& strs) {

unordered\_map<string, vector<string>> mp;

for (string s : strs) {

string key = s;

sort(key.begin(), key.end()); // Sort characters to form a key

mp[key].push\_back(s);

}

vector<vector<string>> result;

for (auto& entry : mp) {

result.push\_back(entry.second);

}

return result;

}

};

**Question-74**

Implement [pow(x, n)](http://www.cplusplus.com/reference/valarray/pow/), which calculates x raised to the power n (i.e., xn).

**Example 1:**

**Input:** x = 2.00000, n = 10

**Output:** 1024.00000

**Example 2:**

**Input:** x = 2.10000, n = 3

**Output:** 9.26100

**Example 3:**

**Input:** x = 2.00000, n = -2

**Output:** 0.25000

**Explanation:** 2-2 = 1/22 = 1/4 = 0.25

**Constraints:**

* -100.0 < x < 100.0
* -231 <= n <= 231-1
* n is an integer.
* Either x is not zero or n > 0.
* -104 <= xn <= 104

**Solution:**

class Solution {

public:

double myPow(double x, int n) {

long long N = n; // Use long long to handle overflow (n = INT\_MIN case)

if (N < 0) {

x = 1 / x; // For negative power, invert x

N = -N; // Make power positive

}

double result = 1.0;

while (N > 0) {

if (N % 2 == 1) // If current bit is set

result \*= x;

x \*= x; // Square the base

N /= 2; // Shift to next bit

}

return result;

}

};

**Question-75**

**Solution:**