DAA-Assignment

Aiml-epsilon

Pranav rao

2211cs020420

**1Q.**[**Find the Index of the First Occurrence in a String**](https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/)

class Solution {

public:

int strStr(string haystack, string needle) {

size\_t position = haystack.find(needle);

return (position != string::npos) ? position : -1;

}

};

**2Q.BItwaise and of number of range**

class Solution {

public:

int rangeBitwiseAnd(int left, int right) {

while (left < right) {

right = right & (right - 1);

}

return right;

}

};

3Q.SQRT(x):

class Solution {

public:

int mySqrt(int x) {

if (x < 2) return x;

int left = 0, right = x, ans = 0;

while (left <= right) {

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

if (mid \* mid <= x) {

ans = mid;

left = mid + 1;

} else {

right = mid - 1;

}

}

return ans;

}

};

4Q.valid-parentheses

#include <stack>

#include <unordered\_map>

#include <string>

class Solution {

public:

bool isValid(std::string s) {

std::stack<char> stack;

std::unordered\_map<char, char> matchingBrackets = {

{')', '('},

{'}', '{'},

{']', '['}

};

for (char c : s) {

if (matchingBrackets.count(c)) {

if (!stack.empty() && stack.top() == matchingBrackets[c]) {

stack.pop();

} else {

return false;

}

} else {

stack.push(c);

}

}

return stack.empty();

}

};

5Q.merge-two-sorted-lists

/\*\*

\* 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) {

if (!list1) return list2;

if (!list2) return list1;

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

list1->next = mergeTwoLists(list1->next, list2);

return list1;

} else {

list2->next = mergeTwoLists(list1, list2->next);

return list2;

}

}

};

**6Q.**[**Remove Duplicates from Sorted List**](https://leetcode.com/problems/remove-duplicates-from-sorted-list/)

/\*\*

\* 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\* deleteDuplicates(ListNode\* head) {

ListNode\* current = head;

while (current != nullptr && current->next != nullptr) {

if (current->val == current->next->val) {

current->next = current->next->next; // Skip the duplicate node

} else {

current = current->next; // Move to the next node

}

}

return head;

}

};

7Q.Find peak element element

class Solution {

public:

int findPeakElement(vector<int>& nums) {

int left = 0, right = nums.size() - 1;

while (left < right) {

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

// Compare the middle element with its next element

if (nums[mid] > nums[mid + 1]) {

// Peak is in the left half

right = mid;

} else {

// Peak is in the right half

left = mid + 1;

}

}

return left; // 'left' and 'right' will eventually converge to the peak

}

};

8Q.BInary tree inorder

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* };

\*/

class Solution {

public:

void inorderHelper(TreeNode\* root, vector<int>& result) {

if (root == nullptr) return;

// Traverse left subtree

inorderHelper(root->left, result);

// Visit current node

result.push\_back(root->val);

// Traverse right subtree

inorderHelper(root->right, result);

}

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> result;

inorderHelper(root, result);

return result;

}

};

9Q N-queens

class Solution {

public:

vector<vector<string>> solveNQueens(int n) {

vector<vector<string>> result;

vector<string> board(n, string(n, '.')); // Initialize the board

vector<int> cols(n, 0); // Track columns where queens are placed

vector<int> diag1(2 \* n - 1, 0); // Track diagonals (r - c) for left diagonal

vector<int> diag2(2 \* n - 1, 0); // Track diagonals (r + c) for right diagonal

backtrack(result, board, 0, n, cols, diag1, diag2);

return result;

}

private:

void backtrack(vector<vector<string>>& result, vector<string>& board, int row, int n,

vector<int>& cols, vector<int>& diag1, vector<int>& diag2) {

if (row == n) { // If all queens are placed

result.push\_back(board); // Add the current board configuration to the result

return;

}

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

int d1 = row - col + (n - 1); // Calculate index for the left diagonal

int d2 = row + col; // Calculate index for the right diagonal

// Check if the column or diagonals are already attacked

if (cols[col] || diag1[d1] || diag2[d2]) continue;

// Place the queen

board[row][col] = 'Q';

cols[col] = diag1[d1] = diag2[d2] = 1;

// Recur for the next row

backtrack(result, board, row + 1, n, cols, diag1, diag2);

// Backtrack: Remove the queen and reset the variables

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

cols[col] = diag1[d1] = diag2[d2] = 0;

}

}

};

10Q.largest-number

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

// Custom comparison function for sorting

bool compare(string &a, string &b) {

return a + b > b + a; // We want the larger concatenated result first

}

class Solution {

public:

string largestNumber(vector<int>& nums) {

// Convert integers to stringsa

vector<string> str\_nums;

for (int num : nums) {

str\_nums.push\_back(to\_string(num));

}

// Sort using the custom comparator

sort(str\_nums.begin(), str\_nums.end(), compare);

// Edge case: if the largest number is "0", return "0"

if (str\_nums[0] == "0") {

return "0";

}

// Join all strings to form the largest number

string result = "";

for (const string& s : str\_nums) {

result += s;

}

return result;

}

};