**1. Find the index of the Occurrence String -**

#include <string>

using namespace std;

class Solution {

public:

int strStr(string haystack, string needle) {

// Find the position of the first occurrence of needle in haystack

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

// If found, return the position; otherwise, return -1

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

}

};

**3. Square Root –**

#include <iostream>

using namespace std;

class Solution {

public:

int mySqrt(int x) {

if (x == 0 || x == 1) return x; // Handle edge cases

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

while (left <= right) {

long long mid = left + (right - left) / 2; // Use long long to avoid overflow

long long square = mid \* mid;

if (square == x) {

return mid; // Perfect square

} else if (square < x) {

result = mid; // Update result

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

} else {

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

}

}

return result; // Return the largest integer whose square <= x

}

};

**4. Largest Number --**

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

class Solution {

public:

string largestNumber(vector<int>& nums) {

vector<string> strNums;

for (int num : nums) {

strNums.push\_back(to\_string(num));

}

auto compare = [](const string& a, const string& b) {

return a + b > b + a;

};

sort(strNums.begin(), strNums.end(), compare);

string result;

for (const string& str : strNums) {

result += str;

}

if (result[0] == '0') {

return "0";

}

return result;

}

};

**5.valid parentheses –**

#include <iostream>

#include <stack>

#include <string>

using namespace std;

class Solution {

public:

bool isValid(string s) {

stack<char> stack;

for (char c : s) {

// If the character is an opening bracket, push it onto the stack

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

stack.push(c);

}

// If the character is a closing bracket, check if it matches the top of the stack

else if (c == ')' || c == '}' || c == ']') {

if (stack.empty()) {

return false; // No opening bracket to match with

}

char top = stack.top();

stack.pop();

// Check if the top of the stack matches the corresponding opening bracket

if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {

return false;

}

}

}

// If the stack is empty, all brackets were matched, otherwise return false

return stack.empty();

}

};

**6.Merge Two sorted Lists –**

#include <iostream>

#include <vector>

// Include the ListNode header

using namespace std;

class Solution {

public:

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

ListNode\* dummy = new ListNode(0); // Create a dummy node to start the merged list

ListNode\* current = dummy; // Pointer to build the new list

// Traverse both lists

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

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

current->next = list1; // Attach list1 node to merged list

list1 = list1->next; // Move the list1 pointer forward

} else {

current->next = list2; // Attach list2 node to merged list

list2 = list2->next; // Move the list2 pointer forward

}

current = current->next; // Move the current pointer forward in the merged list

}

// If there are remaining nodes in list1 or list2, attach them

if (list1 != nullptr) {

current->next = list1;

} else if (list2 != nullptr) {

current->next = list2;

}

return dummy->next; // Return the merged list starting from the first node

}

};

// Helper function to create a linked list from a vector

ListNode\* createList(const vector<int>& nums) {

ListNode\* head = nullptr;

ListNode\* current = nullptr;

for (int num : nums) {

if (!head) {

head = new ListNode(num);

current = head;

} else {

current->next = new ListNode(num);

current = current->next;

}

}

return head;

}

// Helper function to print the linked list

void printList(ListNode\* head) {

while (head != nullptr) {

cout << head->val << " ";

head = head->next;

}

cout << endl;

}

**7. Remove duplicates from sorted List –**

#include <iostream>

using namespace std;

// Assume ListNode structure is already defined and precompiled.

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

ListNode\* current = head;

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

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

// Skip the duplicate node

current->next = current->next->next;

} else {

// Move to the next node

current = current->next;

}

}

return head;

}

};

// Helper function to print the linked list

void printList(ListNode\* head) {

while (head != nullptr) {

cout << head->val << " ";

head = head->next;

}

cout << endl;

}

**8. Finding Peek element –**

#include <iostream>

#include <vector>

using namespace std;

class Solution {

public:

int findPeakElement(vector<int>& nums) {

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

while (left < right) {

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

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

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

};

**9. Binary Tree : In order traversal –**

#include <iostream>

#include <vector>

using namespace std;

class Solution {

public:

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

if (root == nullptr) {

return;

}

// Traverse left subtree

inorderTraversalHelper(root->left, result);

// Visit the root node

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

// Traverse right subtree

inorderTraversalHelper(root->right, result);

}

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

vector<int> result;

inorderTraversalHelper(root, result);

return result;

}

};

**10. N-Queens --**

#include <vector>

#include <string>

using namespace std;

class Solution {

public:

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

vector<vector<string>> solutions;

vector<string> board(n, string(n, '.')); // Initialize an empty n x n board

vector<int> leftRow(n, 0), upperDiag(2 \* n - 1, 0), lowerDiag(2 \* n - 1, 0);

backtrack(0, n, board, solutions, leftRow, upperDiag, lowerDiag);

return solutions;

}

private:

void backtrack(int col, int n, vector<string>& board, vector<vector<string>>& solutions,

vector<int>& leftRow, vector<int>& upperDiag, vector<int>& lowerDiag) {

if (col == n) {

solutions.push\_back(board);

return;

}

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

if (leftRow[row] == 0 && upperDiag[row + col] == 0 && lowerDiag[row - col + n - 1] == 0) {

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

leftRow[row] = upperDiag[row + col] = lowerDiag[row - col + n - 1] = 1;

backtrack(col + 1, n, board, solutions, leftRow, upperDiag, lowerDiag);

// Undo the current placement

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

leftRow[row] = upperDiag[row + col] = lowerDiag[row - col + n - 1] = 0;

}

}

}

};