Binary search

#include <stdio.h>

// Function to perform binary search

int binarySearch(int nums[], int size, int target) {

int left = 0;

int right = size - 1;

while (left <= right) {

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

// Check if target is present at mid

if (nums[mid] == target) {

return mid; }

// If target is greater, ignore the left half

if (nums[mid] < target) {

left = mid + 1;

} else {

// If target is smaller, ignore the right half

right = mid - 1;

}

} // Target is not present in the array

return -1;

}int main() {

int nums[] = {2, 3, 4, 10, 40};

int target = 10;

int size = sizeof(nums) / sizeof(nums[0]);

int result = binarySearch(nums, size, target);

if (result != -1) {

printf("Element is present at index %d\n", result);

} else {

printf("Element is not present in the array\n");

} return 0;}

Binomial coefficient

#include <stdio.h>

// Function to compute the Binomial Coefficient using Dynamic Programming

int binomialCoeff(int n, int k) {

int C[n+1][k+1];

int i, j;

// Calculate value of Binomial Coefficient in bottom-up manner

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

for (j = 0; j <= k; j++) {

// Base cases

if (j == 0 || j == i) {

C[i][j] = 1;

} else {

// Use previously computed values

C[i][j] = C[i-1][j-1] + C[i-1][j];}

} }

return C[n][k];

}

int main() {

int n = 8, k = 8;

printf("Binomial Coefficient C(%d, %d) is %d\n", n, k, binomialCoeff(n, k));

return 0;

}

Sum of subsets

#include <stdio.h>

void sumOfSubsets(int set[], int subset[], int n, int subsetSize, int total, int nodeIndex, int targetSum) { if (total == targetSum) {

// Print the current subset

printf("{ ");

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

printf("%d ", subset[i]);

}

printf("}\n");

return;

}

if (nodeIndex == n || total > targetSum) {

return;

}

// Include the nodeIndex element in the subset

subset[subsetSize] = set[nodeIndex];

sumOfSubsets(set, subset, n, subsetSize + 1, total + set[nodeIndex], nodeIndex + 1, targetSum);

// Exclude the nodeIndex element from the subset

sumOfSubsets(set, subset, n, subsetSize, total, nodeIndex + 1, targetSum);

}

int main() {

int set[] = {10, 7, 5, 18, 12, 20, 15};

int targetSum = 35;

int n = sizeof(set) / sizeof(set[0]);

int subset[n];

printf("Subsets of the given set that sum to %d are:\n", targetSum);

sumOfSubsets(set, subset, n, 0, 0, 0, targetSum);

return 0;}

Strassen’s Matrix

#include <stdio.h>

void strassenMultiply(int A[2][2], int B[2][2], int C[2][2]) {

int M1, M2, M3, M4, M5, M6, M7;

// Calculate M1 to M7

M1 = (A[0][0] + A[1][1]) \* (B[0][0] + B[1][1]);

M2 = (A[1][0] + A[1][1]) \* B[0][0];

M3 = A[0][0] \* (B[0][1] - B[1][1]);

M4 = A[1][1] \* (B[1][0] - B[0][0]);

M5 = (A[0][0] + A[0][1]) \* B[1][1];

M6 = (A[1][0] - A[0][0]) \* (B[0][0] + B[0][1]);

M7 = (A[0][1] - A[1][1]) \* (B[1][0] + B[1][1]);

// Calculate C[2][2]

C[0][0] = M1 + M4 - M5 + M7;

C[0][1] = M3 + M5;

C[1][0] = M2 + M4;

C[1][1] = M1 - M2 + M3 + M6;

}

void printMatrix(int matrix[2][2]) {

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

for (int j = 0; j < 2; j++) {

printf("%d ", matrix[i][j]);

}printf("\n");}}

int main() {

int A[2][2] = { {1, 2}, {3, 4} };

int B[2][2] = { {5, 6}, {7, 8} };

int C[2][2];

strassenMultiply(A, B, C);

printf("Resultant Matrix C is:\n");

printMatrix(C); return 0;

}

Find Max and Min value in the list using divide and conquer find its time complexity.

Testing Condition – Count the number of times in Comparison to find Min\_Max value in a list n for the given set of elements.

#include <stdio.h>

typedef struct {

int min;

int max;

int comparisonCount; // to count the number of comparisons

} MinMax;

// Function to find the min and max values using divide and conquer

MinMax findMinMax(int arr[], int low, int high) {

MinMax result, leftResult, rightResult;

// If there is only one element

if (low == high) {

result.min = result.max = arr[low];

result.comparisonCount = 0;

return result;

} if (high == low + 1) {

result.comparisonCount = 1;

if (arr[low] < arr[high]) {

result.min = arr[low];

result.max = arr[high];

} else {

result.min = arr[high];

result.max = arr[low];

} return result;

}

int mid = (low + high) / 2;

leftResult = findMinMax(arr, low, mid);

rightResult = findMinMax(arr, mid + 1, high);

result.comparisonCount = leftResult.comparisonCount + rightResult.comparisonCount + 2;

if (leftResult.min < rightResult.min) {

result.min = leftResult.min;

} else { result.min = rightResult.min; }

if (leftResult.max > rightResult.max) {

result.max = leftResult.max;

} else {

result.max = rightResult.max;

}

return result;

}

int main() {

int arr[] = {1000, 11, 445, 1, 330, 3000};

int n = sizeof(arr) / sizeof(arr[0]);

MinMax result = findMinMax(arr, 0, n - 1);

printf("Minimum value in the list is: %d\n", result.min);

printf("Maximum value in the list is: %d\n", result.max);

printf("Number of comparisons made: %d\n", result.comparisonCount);

return 0;

}

Pascals triangle

#include <stdio.h>

// Function to generate Pascal's Triangle

void generatePascalsTriangle(int n) {

int triangle[n][n];

// Initialize all entries to 0

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

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

// First and last values in every row are 1

if (i == 0 || i == line) {

triangle[line][i] = 1;

} else {

// Other values are sum of values just above and left of above

triangle[line][i] = triangle[line - 1][i - 1] + triangle[line - 1][i];

}

printf("%d ", triangle[line][i]);

}

printf("\n");

}

}

int main() {

int n;

printf("Enter the number of rows in Pascal's Triangle: ");

scanf("%d", &n);

generatePascalsTriangle(n);

return 0;

}

1. Write a program to find the sum of digits. You are given a **0- indexed** array nums consisting of **positive** integers. You can choose two indices i and j, such that i != j, and the sum of digits of the number nums[i] is equal to that of nums[j]. Return *the* **maximum** *value of* nums[i] + nums[j] *that you can obtain over all possible indices* i *and* j *that satisfy the conditions.*

#include <stdio.h>

// Function to calculate the sum of digits of a number

int sumOfDigits(int num) {

int sum = 0;

while (num > 0) {

sum += num % 10;

num /= 10;

}

return sum;

}

// Function to find the maximum sum of two numbers with the same digit sum

int maxSumPairWithEqualDigitSum(int nums[], int size) {

int maxSum = -1;

int digitSum[100] = {0}; // Assuming digit sum ranges will be within a small constant range

// Calculate digit sums and find pairs with equal digit sums

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

int sum = sumOfDigits(nums[i]);

// Check if there's already a number with the same digit sum

if (digitSum[sum] != 0) {

// Update maxSum if the current pair sum is greater

if (nums[i] + digitSum[sum] > maxSum) {

maxSum = nums[i] + digitSum[sum];

} }

// Update the array to store the maximum number for each digit sum

if (nums[i] > digitSum[sum]) {

digitSum[sum] = nums[i];

}

} return maxSum;

}

int main() {

int nums[] = {51, 71, 17, 42};

int size = sizeof(nums) / sizeof(nums[0]);

int result = maxSumPairWithEqualDigitSum(nums, size);

if (result != -1) {

printf("The maximum sum of nums[i] + nums[j] with equal digit sums is: %d\n", result);

} else {

printf("No pairs found with equal digit sums.\n");

}

return 0;

}

1. Consider a two integer arrays nums1 and nums2, sorted in non-increasing order and two integers m and n, representing the number of elements in nums1 and nums2 respectively. Write a program to Merge them into a single array using Merge Sort. Derive time complexity of merge sort

#include <stdio.h>

// Function to merge two sorted arrays in non-increasing order

void merge(int nums1[], int m, int nums2[], int n, int merged[]) {

int i = 0, j = 0, k = 0;

// Merge arrays while elements are remaining in both

while (i < m && j < n) {

if (nums1[i] >= nums2[j]) {

merged[k++] = nums1[i++];

} else {

merged[k++] = nums2[j++];

} } // Copy remaining elements of nums1 if any

while (i < m) {

merged[k++] = nums1[i++];

}

// Copy remaining elements of nums2 if any

while (j < n) {

merged[k++] = nums2[j++];

}}

// Function to print the array

void printArray(int arr[], int size) {

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

printf("%d ", arr[i]);

}

printf("\n");}

int main() {

int nums1[] = {9, 7, 5, 3, 1};

int nums2[] = {10, 8, 6, 4, 2};

int m = sizeof(nums1) / sizeof(nums1[0]);

int n = sizeof(nums2) / sizeof(nums2[0]);

int merged[m + n];

merge(nums1, m, nums2, n, merged);

printf("Merged array in non-increasing order: \n");

printArray(merged, m + n);

return 0;

}

Floyd warshall

#include <stdio.h>

#define INF 99999

#define V 4 // Number of vertices in the graph

// Function to print the solution matrix

void printSolution(int dist[][V]) {

printf("Shortest distances between every pair of vertices:\n");

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

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

if (dist[i][j] == INF)

printf("%7s", "INF");

else

printf("%7d", dist[i][j]);

}

printf("\n");

}}

// Function to find all pairs shortest paths using Floyd's algorithm

void floydWarshall(int graph[][V]) {

int dist[V][V]; // Output matrix where dist[i][j] will be the shortest distance from i to j

// Initialize the solution matrix same as input graph matrix

for (int i = 0; i < V; i++)

for (int j = 0; j < V; j++)

dist[i][j] = graph[i][j];

// Floyd's algorithm

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

// Pick all vertices as source one by one

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

// Pick all vertices as destination for the above picked source

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

// If vertex k is on the shortest path from i to j, then update the value of dist[i][j]

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

printSolution(dist);

}

int main() {

// Example adjacency matrix representing the graph

int graph[V][V] = {{0, 5, INF, 10},

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0}};

// Call the function to find all pairs shortest paths

floydWarshall(graph);

return 0;

}

1. Write a program to perform Knapsack problem using dynamic programming for the following set of object values.,

#include <stdio.h>

// Function to find maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to solve the Knapsack problem using dynamic programming

int knapsack(int W, int wt[], int val[], int n) {

int i, w;

int K[n + 1][W + 1];

// Build K[][] in bottom-up manner

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

for (w = 0; w <= W; w++) {

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i - 1] <= w)

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

else

K[i][w] = K[i - 1][w];

}

}

return K[n][W];

}

int main() {

int val[] = {60, 100, 120}; // Values of items

int wt[] = {10, 20, 30}; // Weights of items

int W = 50; // Knapsack capacity

int n = sizeof(val) / sizeof(val[0]); // Number of items

int max\_value = knapsack(W, wt, val, n);

printf("Maximum value that can be obtained: %d\n", max\_value);

return 0;

}

1. Write a program to find a minimum spanning tree using prims technique for the given graph.

#include <stdio.h>

#include <limits.h>

#define V 5 // Number of vertices in the graph

// Function to find the vertex with minimum key value

int minKey(int key[], int mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the constructed MST stored in parent[]

void printMST(int parent[], int graph[V][V]) {

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

}

// Function to construct and print MST for a graph represented using adjacency matrix representation

void primMST(int graph[V][V]) {

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

int mstSet[V]; // To represent set of vertices included in MST

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

// Always include first vertex in MST.

key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = 1;

// Update key value and parent index of the adjacent vertices of the picked vertex.

// Consider only those vertices which are not yet included in MST

for (int v = 0; v < V; v++) {

// graph[u][v] is non-zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the constructed MST

printMST(parent, graph);

}

// Driver program to test above functions

int main() {

// Example graph representation using adjacency matrix

int graph[V][V] = {{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}};

// Print the Minimum Spanning Tree

primMST(graph);

return 0;

}

Selection sort

#include <stdio.h>

// Function to perform Selection Sort

void selectionSort(int arr[], int n) {

int i, j, min\_idx;

// Iterate through all elements

for (i = 0; i < n - 1; i++) {

// Find the minimum element in unsorted array

min\_idx = i;

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

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

// Swap the found minimum element with the first element

if (min\_idx != i) {

int temp = arr[i];

arr[i] = arr[min\_idx];

arr[min\_idx] = temp;

}

}

}

// Function to print the array

void printArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Main function

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

// Perform Selection Sort

selectionSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

Pallindrom

#include <stdio.h>

// Function to check if a character is alphanumeric

int isAlphanumeric(char ch) {

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9');

}

// Function to convert uppercase letter to lowercase

char toLowercase(char ch) {

if (ch >= 'A' && ch <= 'Z') {

return ch + ('a' - 'A');

}

return ch;

}

// Function to check if a string is a palindrome

int isPalindrome(char str[]) {

int left = 0;

int right = 0;

// Find the length of the string

while (str[right] != '\0') {

right++;

}

right--; // adjust to point to the last character

// Compare characters from both ends towards the center of the string

while (left < right) {

// Ignore non-alphanumeric characters

while (!isAlphanumeric(str[left]) && left < right) {

left++;

}

while (!isAlphanumeric(str[right]) && left < right) {

right--;

}

// Convert characters to lowercase for case insensitivity

char left\_char = toLowercase(str[left]);

char right\_char = toLowercase(str[right]);

// Compare characters

if (left\_char != right\_char) {

return 0; // Not a palindrome

}

left++;

right--;

}

return 1; // Palindrome

}

int main() {

char str[100];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin);

// Remove newline character from fgets

str[strlen(str) - 1] = '\0';

// Check if the string is a palindrome

if (isPalindrome(str)) {

printf("'%s' is a palindrome.\n", str);

} else {

printf("'%s' is not a palindrome.\n", str);

}

return 0;

}

N QUEEN Problem

#include <stdio.h>

#include <stdlib.h>

#define MAX\_N 20 // Maximum value of n for the chessboard size

int board[MAX\_N][MAX\_N]; // Chessboard representation

// Function to check if placing a queen at board[row][col] is safe

int isSafe(int row, int col, int n) {

// Check if there's a queen in the same column up to 'row'

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

if (board[i][col] == 1) {

return 0;

}

}

// Check upper-left diagonal

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 1) {

return 0;

}

}

// Check upper-right diagonal

for (int i = row, j = col; i >= 0 && j < n; i--, j++) {

if (board[i][j] == 1) {

return 0;

}

}

return 1; // Safe to place queen at board[row][col]

}

// Function to solve N-Queens problem using backtracking

void solveNQueens(int row, int n, int \*count) {

if (row == n) {

// Found a solution, print the board

printf("Solution %d:\n", ++(\*count));

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

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

printf("%c ", board[i][j] ? 'Q' : '.');

}

printf("\n");

}

printf("\n");

return;

}

// Try placing queen in each column of current row

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

if (isSafe(row, col, n)) {

// Place queen

board[row][col] = 1;

// Recur to place rest of the queens

solveNQueens(row + 1, n, count);

// Backtrack

board[row][col] = 0;

}

}

}

// Main function to get solutions for N-Queens problem

void nQueens(int n) {

int count = 0;

solveNQueens(0, n, &count);

if (count == 0) {

printf("No solutions found for n = %d.\n", n);

} else {

printf("Total solutions found: %d\n", count);

}

}

// Driver code

int main() {

int n;

printf("Enter the size of the chessboard (n): ");

scanf("%d", &n);

if (n <= 0 || n > MAX\_N) {

printf("Invalid input. Please enter a value between 1 and %d.\n", MAX\_N);

return 1;

}

nQueens(n);

return 0;

}

Binary search tree

#include <stdio.h>

#include <limits.h>

#define MAX\_N 10

// Function to find minimum cost of constructing BST from keys[i] to keys[j]

int optimalBST(int keys[], int freq[], int n) {

// dp[i][j] will store the minimum cost of constructing BST from keys[i] to keys[j]

int dp[MAX\_N][MAX\_N] = {0};

// root[i][j] will store the root of optimal BST for keys[i] to keys[j]

int root[MAX\_N][MAX\_N] = {0};

// Initialize base cases: Single keys

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

dp[i][i] = freq[i];

root[i][i] = i;

}

// Build the dp table

for (int len = 2; len <= n; len++) { // Subtree lengths

for (int i = 0; i <= n - len + 1; i++) { // Start index

int j = i + len - 1; // End index

dp[i][j] = INT\_MAX;

// Compute cost of subtree from keys[i] to keys[j]

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

int cost = ((k > i) ? dp[i][k - 1] : 0) +

((k < j) ? dp[k + 1][j] : 0) +

sum(freq, i, j); // sum of frequencies from i to j

if (cost < dp[i][j]) {

dp[i][j] = cost;

root[i][j] = k;

}

}

}

}

printf("Minimum cost of constructing optimal BST is: %d\n", dp[0][n - 1]);

return dp[0][n - 1];

}

// Function to calculate sum of frequencies from freq[i] to freq[j]

int sum(int freq[], int i, int j) {

int s = 0;

for (int k = i; k <= j; k++)

s += freq[k];

return s;

}

// Example usage

int main() {

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

int n = sizeof(keys) / sizeof(keys[0]);

optimalBST(keys, freq, n);

return 0;

}

Permutations of integer

#include <stdio.h>

#include <stdlib.h>

// Function to swap two integers in an array

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to print the array

void printArray(int arr[], int n) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

// Function to generate all permutations of an array

void generatePermutations(int arr[], int start, int end) {

if (start == end) {

printArray(arr, end + 1); // Print the current permutation

return;

}

for (int i = start; i <= end; i++) {

// Swap element at index start with element at index i

swap(&arr[start], &arr[i]);

// Recursively generate permutations for the remaining elements

generatePermutations(arr, start + 1, end);

// Backtrack: Undo the swap to restore the original array

swap(&arr[start], &arr[i]);

}

}

// Comparator function for qsort to sort integers in ascending order

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

// Function to initialize permutation generation

void permutation(int arr[], int n) {

// Sort the array to generate permutations in lexicographical order

qsort(arr, n, sizeof(int), compare);

// Generate permutations recursively

generatePermutations(arr, 0, n - 1);

}

// Main function

int main() {

int arr[] = {1, 2, 3};

int n = sizeof(arr) / sizeof(arr[0]);

printf("All permutations of the array:\n");

permutation(arr, n);

return 0;

}

1. Write a program to return all the possible subsets for a given integer array. Return the solution in any order.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 20 // Maximum size of the array

// Function to print the subset currently formed

void printSubset(int subset[], int subsetSize) {

printf("{ ");

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

printf("%d ", subset[i]);

}

printf("}\n");

}

// Function to recursively generate subsets

void generateSubsets(int arr[], int subset[], int index, int n, int subsetSize) {

// Base case: Print the subset when we reach the end of the array

if (index == n) {

printSubset(subset, subsetSize);

return;

}

// Include the current element in the subset and recur

subset[subsetSize] = arr[index];

generateSubsets(arr, subset, index + 1, n, subsetSize + 1);

// Exclude the current element from the subset and recur

generateSubsets(arr, subset, index + 1, n, subsetSize);

}

// Function to initialize subset generation

void subsets(int arr[], int n) {

int subset[MAX\_SIZE]; // Temporary array to store subsets

generateSubsets(arr, subset, 0, n, 0); // Start from index 0 with empty subset

}

// Main function

int main() {

int arr[] = {1, 2, 3};

int n = sizeof(arr) / sizeof(arr[0]);

printf("All subsets of the array:\n");

subsets(arr, n);

return 0;

}

1. Write a program to find the longest common subsequent problem for the two list

**Input:** S1 = “ABC”, S2 = “ACD”  
**Output:** 2  
**Explanation:** The longest subsequence

#include <stdio.h>

#include <string.h>

#define MAX\_LEN 100

// Function to find the length of Longest Common Subsequence (LCS)

int longestCommonSubsequence(char S1[], char S2[]) {

int m = strlen(S1);

int n = strlen(S2);

int dp[MAX\_LEN][MAX\_LEN];

// Initialize the dp array

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

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

if (i == 0 || j == 0)

dp[i][j] = 0;

else if (S1[i - 1] == S2[j - 1])

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

else

dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1];

}

}

return dp[m][n];

}

// Main function for testing

int main() {

char S1[] = "ABC";

char S2[] = "ACD";

int lcs\_length = longestCommonSubsequence(S1, S2);

printf("Longest Common Subsequence length: %d\n", lcs\_length);

return 0;

}

1. Given a number N and a time X unit, the task is to find the number of containers that are filled completely in X unit if containers are arranged in pyramid fashion as shown below.

input: N = 3, X = 5

#include <stdio.h>

// Function to calculate the number of completely filled containers

int countFilledContainers(int N, int X) {

int total\_containers = N \* (N + 1) / 2;

int filled\_containers = 0;

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

if (X >= i) {

filled\_containers += i;

X -= i;

} else {

break;

}

}

return filled\_containers;

}

// Main function for testing

int main() {

int N = 3; // Number of levels in the pyramid

int X = 5; // Total time units

int filled\_count = countFilledContainers(N, X);

printf("Number of completely filled containers: %d\n", filled\_count);

return 0;

}

1. You are given a pattern in the form of a string and a collection of words. Your task is to determine if the pattern string and the collection of words have the same order.

##### Sample Input 1 :

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_WORDS 100

#define MAX\_LENGTH 50

// Function to check if the pattern matches each word in the collection

bool doesPatternMatch(char pattern[], char words[][MAX\_LENGTH], int n) {

int len = strlen(pattern);

int pattern\_map[26] = {0}; // Assuming only lowercase English letters

// Map characters in pattern to their positions

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

pattern\_map[pattern[i] - 'a'] = i + 1; // Store position (1-based index)

}

// Check pattern against each word

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

char word[MAX\_LENGTH];

strcpy(word, words[k]);

int word\_len = strlen(word);

int word\_pattern[word\_len];

// Map characters in the word to their positions based on pattern\_map

for (int i = 0; i < word\_len; i++) {

word\_pattern[i] = pattern\_map[word[i] - 'a'];

}

// Compare word's pattern with pattern of pattern string

for (int i = 1; i < word\_len; i++) {

if (word\_pattern[i] < word\_pattern[i - 1]) {

return false; // Pattern does not match

}

}

}

return true; // All words match the pattern

}

// Main function for testing

int main() {

char pattern[] = "abc";

char words[MAX\_WORDS][MAX\_LENGTH] = {"apple", "banana", "cherry"};

int n = 3; // Number of words in the collection

if (doesPatternMatch(pattern, words, n)) {

printf("Pattern matches the order of words.\n");

} else {

printf("Pattern does not match the order of words.\n");

}

return 0;

}

1. Implement an algorithm to find the maximum and minimum elements in an array using the Divide and Conquer approach. Analyze its time and space complexities.

#include <stdio.h>

#include <limits.h>

// Structure to hold both min and max values

struct MinMax {

int min;

int max;

};

// Function to find the minimum and maximum elements using Divide and Conquer

struct MinMax findMinMax(int arr[], int low, int high) {

struct MinMax result, left, right;

int mid;

// Base case: If array contains only one element

if (low == high) {

result.min = arr[low];

result.max = arr[low];

return result;

}

// If there are two elements

if (high == low + 1) {

if (arr[low] > arr[high]) {

result.max = arr[low];

result.min = arr[high];

} else {

result.max = arr[high];

result.min = arr[low];

}

return result;

}

// Divide the array into two halves

mid = (low + high) / 2;

left = findMinMax(arr, low, mid);

right = findMinMax(arr, mid + 1, high);

// Compare minimums and maximums of two halves

if (left.min < right.min)

result.min = left.min;

else

result.min = right.min;

if (left.max > right.max)

result.max = left.max;

else

result.max = right.max;

return result;}

// Main function for testing

int main() {

int arr[] = {3, 7, 1, 9, 4, 5};

int n = sizeof(arr) / sizeof(arr[0]);

struct MinMax result = findMinMax(arr, 0, n - 1);

printf("Minimum element: %d\n", result.min);

printf("Maximum element: %d\n", result.max);

return 0;

}

1. Write a program for to count the inversion inn an array

Input: arr[] = {8, 4, 2, 1}

Output: 6

#include <stdio.h>

// Function to merge two sorted subarrays and count inversions

int mergeAndCount(int arr[], int temp[], int left, int mid, int right) {

int i = left; // Starting index for left subarray

int j = mid + 1; // Starting index for right subarray

int k = left; // Starting index to be sorted

int inv\_count = 0; // Initialize inversion count

while ((i <= mid) && (j <= right)) {

if (arr[i] <= arr[j]) {

temp[k++] = arr[i++];

} else {

// There are mid - i inversions, because all remaining elements in the left subarray

// (arr[i], arr[i+1], ..., arr[mid]) are greater than arr[j]

temp[k++] = arr[j++];

inv\_count += (mid - i + 1);

}

}

// Copy the remaining elements of left subarray, if any

while (i <= mid) {

temp[k++] = arr[i++];

}

// Copy the remaining elements of right subarray, if any

while (j <= right) {

temp[k++] = arr[j++];

}

// Copy the sorted subarray into Original array

for (i = left; i <= right; i++) {

arr[i] = temp[i];

}

return inv\_count;

}

// Function to divide the array into subarrays and recursively count inversions

int mergeSortAndCount(int arr[], int temp[], int left, int right) {

int mid, inv\_count = 0;

if (left < right) {

mid = (left + right) / 2;

inv\_count += mergeSortAndCount(arr, temp, left, mid);

inv\_count += mergeSortAndCount(arr, temp, mid + 1, right);

inv\_count += mergeAndCount(arr, temp, left, mid, right);

}

return inv\_count;

}

// Function to count inversions in the array

int countInversions(int arr[], int n) {

int temp[n];

return mergeSortAndCount(arr, temp, 0, n - 1);

}

// Main function for testing

int main() {

int arr[] = {8, 4, 2, 1}; int n = sizeof(arr) / sizeof(arr[0]);

int inversion\_count = countInversions(arr, n);

printf("Number of inversions: %d\n", inversion\_count);

return 0;

}