1.

#include <stdio.h>

// Function to count elements x such that x + 1 is also in arr

int countElements(int\* arr, int arrSize) {

int count = 0;

int hash[1001] = {0}; // Since arr[i] is in the range 0 <= arr[i] <= 1000

// Fill the hash table

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

hash[arr[i]] = 1;

}

// Count elements x such that x + 1 is also in arr

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

if (hash[arr[i] + 1]) {

count++;

}

}

return count;

}

int main() {

// Example 1

int arr1[] = {1, 2, 3};

int arrSize1 = sizeof(arr1) / sizeof(arr1[0]);

printf("Output: %d\n", countElements(arr1, arrSize1)); // Output: 2

// Example 2

int arr2[] = {1, 1, 3, 3, 5, 5, 7, 7};

int arrSize2 = sizeof(arr2) / sizeof(arr2[0]);

printf("Output: %d\n", countElements(arr2, arrSize2)); // Output: 0

return 0;

}

2. #include <stdio.h>

#include <string.h>

// Function to perform left shift by one

void leftShift(char\* s, int length) {

char temp = s[0];

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

s[i] = s[i + 1];

}

s[length - 1] = temp;

}

// Function to perform right shift by one

void rightShift(char\* s, int length) {

char temp = s[length - 1];

for (int i = length - 1; i > 0; i--) {

s[i] = s[i - 1];

}

s[0] = temp;

}

// Function to perform the shifts

void performShifts(char\* s, int shift[][2], int shiftCount) {

int length = strlen(s);

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

int direction = shift[i][0];

int amount = shift[i][1];

if (direction == 0) { // left shift

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

leftShift(s, length);

}

} else if (direction == 1) { // right shift

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

rightShift(s, length);

}

}

}

}

int main() {

// Example 1

char s1[] = "abc";

int shift1[][2] = {{0, 1}, {1, 2}};

int shiftCount1 = sizeof(shift1) / sizeof(shift1[0]);

performShifts(s1, shift1, shiftCount1);

printf("Output: %s\n", s1); // Output should be "cab"

// Example 2

char s2[] = "abcdefg";

int shift2[][2] = {{1, 1}, {1, 1}, {0, 2}, {1, 3}};

int shiftCount2 = sizeof(shift2) / sizeof(shift2[0]);

performShifts(s2, shift2, shiftCount2);

printf("Output: %s\n", s2); // Output should be "efgabcd"

return 0;

}

3.

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int\*\* mat;

int rows;

int cols;

} BinaryMatrix;

// Function to get the element at (row, col)

int get(BinaryMatrix\* binaryMatrix, int row, int col) {

return binaryMatrix->mat[row][col];

}

// Function to get the dimensions of the matrix

void dimensions(BinaryMatrix\* binaryMatrix, int\* dims) {

dims[0] = binaryMatrix->rows;

dims[1] = binaryMatrix->cols;

}

// Function to find the leftmost column with at least a 1

int leftMostColumnWithOne(BinaryMatrix\* binaryMatrix) {

int dims[2];

dimensions(binaryMatrix, dims);

int rows = dims[0];

int cols = dims[1];

int leftMostCol = cols; // Initialize to out of bounds

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

int low = 0, high = cols - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (get(binaryMatrix, i, mid) == 1) {

leftMostCol = (mid < leftMostCol) ? mid : leftMostCol;

high = mid - 1; // Try to find an earlier 1 in the row

} else {

low = mid + 1; // Move to the right part of the row

}

}

}

return (leftMostCol == cols) ? -1 : leftMostCol;

}

// Helper function to create a BinaryMatrix

BinaryMatrix\* createBinaryMatrix(int\*\* mat, int rows, int cols) {

BinaryMatrix\* binaryMatrix = (BinaryMatrix\*)malloc(sizeof(BinaryMatrix));

binaryMatrix->mat = mat;

binaryMatrix->rows = rows;

binaryMatrix->cols = cols;

return binaryMatrix;

}

// Helper function to free a BinaryMatrix

void freeBinaryMatrix(BinaryMatrix\* binaryMatrix) {

free(binaryMatrix);

}

int main() {

// Example 1

int mat1Data[2][2] = {{0, 0}, {1, 1}};

int\* mat1[2] = {mat1Data[0], mat1Data[1]};

BinaryMatrix\* binaryMatrix1 = createBinaryMatrix(mat1, 2, 2);

printf("Output: %d\n", leftMostColumnWithOne(binaryMatrix1)); // Output should be 0

freeBinaryMatrix(binaryMatrix1);

// Example 2

int mat2Data[2][2] = {{0, 0}, {0, 1}};

int\* mat2[2] = {mat2Data[0], mat2Data[1]};

BinaryMatrix\* binaryMatrix2 = createBinaryMatrix(mat2, 2, 2);

printf("Output: %d\n", leftMostColumnWithOne(binaryMatrix2)); // Output should be 1

freeBinaryMatrix(binaryMatrix2);

// Example 3

int mat3Data[2][2] = {{0, 0}, {0, 0}};

int\* mat3[2] = {mat3Data[0], mat3Data[1]};

BinaryMatrix\* binaryMatrix3 = createBinaryMatrix(mat3, 2, 2);

printf("Output: %d\n", leftMostColumnWithOne(binaryMatrix3)); // Output should be -1

freeBinaryMatrix(binaryMatrix3);

return 0;

}

4. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Define a structure for a linked list node (queue node)

typedef struct ListNode {

int value;

struct ListNode\* next;

} ListNode;

// Define a structure for the queue

typedef struct Queue {

ListNode \*front, \*rear;

} Queue;

// Define a structure for the hash map node

typedef struct HashNode {

int key;

int count;

struct HashNode\* next;

} HashNode;

// Define a structure for the hash map

#define HASH\_SIZE 1000

typedef struct HashMap {

HashNode\* buckets[HASH\_SIZE];

} HashMap;

// Function prototypes

void enqueue(Queue\* q, int value);

int dequeue(Queue\* q);

void hashMapPut(HashMap\* map, int key);

int hashMapGet(HashMap\* map, int key);

void hashMapFree(HashMap\* map);

unsigned int hashFunction(int key);

// Define the FirstUnique structure

typedef struct {

Queue\* queue;

HashMap\* map;

} FirstUnique;

// Function to create a new ListNode

ListNode\* createListNode(int value) {

ListNode\* newNode = (ListNode\*)malloc(sizeof(ListNode));

newNode->value = value;

newNode->next = NULL;

return newNode;

}

// Function to create a new Queue

Queue\* createQueue() {

Queue\* q = (Queue\*)malloc(sizeof(Queue));

q->front = q->rear = NULL;

return q;

}

// Function to enqueue a value to the queue

void enqueue(Queue\* q, int value) {

ListNode\* newNode = createListNode(value);

if (q->rear == NULL) {

q->front = q->rear = newNode;

return;

}

q->rear->next = newNode;

q->rear = newNode;

}

// Function to dequeue a value from the queue

int dequeue(Queue\* q) {

if (q->front == NULL) return INT\_MIN;

ListNode\* temp = q->front;

int value = temp->value;

q->front = q->front->next;

if (q->front == NULL) q->rear = NULL;

free(temp);

return value;

}

// Function to create a new HashMap

HashMap\* createHashMap() {

HashMap\* map = (HashMap\*)malloc(sizeof(HashMap));

for (int i = 0; i < HASH\_SIZE; i++) {

map->buckets[i] = NULL;

}

return map;

}

// Hash function to get the index for a key

unsigned int hashFunction(int key) {

return abs(key) % HASH\_SIZE;

}

// Function to put a key in the hash map

void hashMapPut(HashMap\* map, int key) {

unsigned int index = hashFunction(key);

HashNode\* node = map->buckets[index];

while (node != NULL) {

if (node->key == key) {

node->count++;

return;

}

node = node->next;

}

HashNode\* newNode = (HashNode\*)malloc(sizeof(HashNode));

newNode->key = key;

newNode->count = 1;

newNode->next = map->buckets[index];

map->buckets[index] = newNode;

}

// Function to get the count of a key in the hash map

int hashMapGet(HashMap\* map, int key) {

unsigned int index = hashFunction(key);

HashNode\* node = map->buckets[index];

while (node != NULL) {

if (node->key == key) {

return node->count;

}

node = node->next;

}

return 0;

}

// Function to free the hash map

void hashMapFree(HashMap\* map) {

for (int i = 0; i < HASH\_SIZE; i++) {

HashNode\* node = map->buckets[i];

while (node != NULL) {

HashNode\* temp = node;

node = node->next;

free(temp);

}

}

free(map);

}

// Function to create a FirstUnique object

FirstUnique\* firstUniqueCreate(int\* nums, int numsSize) {

FirstUnique\* obj = (FirstUnique\*)malloc(sizeof(FirstUnique));

obj->queue = createQueue();

obj->map = createHashMap();

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

enqueue(obj->queue, nums[i]);

hashMapPut(obj->map, nums[i]);

}

return obj;

}

// Function to show the first unique integer

int firstUniqueShowFirstUnique(FirstUnique\* obj) {

while (obj->queue->front != NULL && hashMapGet(obj->map, obj->queue->front->value) > 1) {

dequeue(obj->queue);

}

if (obj->queue->front == NULL) return -1;

return obj->queue->front->value;

}

// Function to add a value to the queue

void firstUniqueAdd(FirstUnique\* obj, int value) {

enqueue(obj->queue, value);

hashMapPut(obj->map, value);

}

// Function to free the FirstUnique object

void firstUniqueFree(FirstUnique\* obj) {

ListNode\* current = obj->queue->front;

while (current != NULL) {

ListNode\* temp = current;

current = current->next;

free(temp);

}

free(obj->queue);

hashMapFree(obj->map);

free(obj);

}

// Example usage

int main() {

int nums[] = {2, 3, 5};

FirstUnique\* firstUnique = firstUniqueCreate(nums, 3);

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 2

firstUniqueAdd(firstUnique, 5); // the queue is now [2, 3, 5, 5]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 2

firstUniqueAdd(firstUnique, 2); // the queue is now [2, 3, 5, 5, 2]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 3

firstUniqueAdd(firstUnique, 3); // the queue is now [2, 3, 5, 5, 2, 3]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return -1

firstUniqueFree(firstUnique);

return 0;

}

4. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Define a structure for a linked list node (queue node)

typedef struct ListNode {

int value;

struct ListNode\* next;

} ListNode;

// Define a structure for the queue

typedef struct Queue {

ListNode \*front, \*rear;

} Queue;

// Define a structure for the hash map node

typedef struct HashNode {

int key;

int count;

struct HashNode\* next;

} HashNode;

// Define a structure for the hash map

#define HASH\_SIZE 1000

typedef struct HashMap {

HashNode\* buckets[HASH\_SIZE];

} HashMap;

// Function prototypes

void enqueue(Queue\* q, int value);

int dequeue(Queue\* q);

void hashMapPut(HashMap\* map, int key);

int hashMapGet(HashMap\* map, int key);

void hashMapFree(HashMap\* map);

unsigned int hashFunction(int key);

// Define the FirstUnique structure

typedef struct {

Queue\* queue;

HashMap\* map;

} FirstUnique;

// Function to create a new ListNode

ListNode\* createListNode(int value) {

ListNode\* newNode = (ListNode\*)malloc(sizeof(ListNode));

newNode->value = value;

newNode->next = NULL;

return newNode;

}

// Function to create a new Queue

Queue\* createQueue() {

Queue\* q = (Queue\*)malloc(sizeof(Queue));

q->front = q->rear = NULL;

return q;

}

// Function to enqueue a value to the queue

void enqueue(Queue\* q, int value) {

ListNode\* newNode = createListNode(value);

if (q->rear == NULL) {

q->front = q->rear = newNode;

return;

}

q->rear->next = newNode;

q->rear = newNode;

}

// Function to dequeue a value from the queue

int dequeue(Queue\* q) {

if (q->front == NULL) return INT\_MIN;

ListNode\* temp = q->front;

int value = temp->value;

q->front = q->front->next;

if (q->front == NULL) q->rear = NULL;

free(temp);

return value;

}

// Function to create a new HashMap

HashMap\* createHashMap() {

HashMap\* map = (HashMap\*)malloc(sizeof(HashMap));

for (int i = 0; i < HASH\_SIZE; i++) {

map->buckets[i] = NULL;

}

return map;

}

// Hash function to get the index for a key

unsigned int hashFunction(int key) {

return abs(key) % HASH\_SIZE;

}

// Function to put a key in the hash map

void hashMapPut(HashMap\* map, int key) {

unsigned int index = hashFunction(key);

HashNode\* node = map->buckets[index];

while (node != NULL) {

if (node->key == key) {

node->count++;

return;

}

node = node->next;

}

HashNode\* newNode = (HashNode\*)malloc(sizeof(HashNode));

newNode->key = key;

newNode->count = 1;

newNode->next = map->buckets[index];

map->buckets[index] = newNode;

}

// Function to get the count of a key in the hash map

int hashMapGet(HashMap\* map, int key) {

unsigned int index = hashFunction(key);

HashNode\* node = map->buckets[index];

while (node != NULL) {

if (node->key == key) {

return node->count;

}

node = node->next;

}

return 0;

}

// Function to free the hash map

void hashMapFree(HashMap\* map) {

for (int i = 0; i < HASH\_SIZE; i++) {

HashNode\* node = map->buckets[i];

while (node != NULL) {

HashNode\* temp = node;

node = node->next;

free(temp);

}

}

free(map);

}

// Function to create a FirstUnique object

FirstUnique\* firstUniqueCreate(int\* nums, int numsSize) {

FirstUnique\* obj = (FirstUnique\*)malloc(sizeof(FirstUnique));

obj->queue = createQueue();

obj->map = createHashMap();

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

enqueue(obj->queue, nums[i]);

hashMapPut(obj->map, nums[i]);

}

return obj;

}

// Function to show the first unique integer

int firstUniqueShowFirstUnique(FirstUnique\* obj) {

while (obj->queue->front != NULL && hashMapGet(obj->map, obj->queue->front->value) > 1) {

dequeue(obj->queue);

}

if (obj->queue->front == NULL) return -1;

return obj->queue->front->value;

}

// Function to add a value to the queue

void firstUniqueAdd(FirstUnique\* obj, int value) {

enqueue(obj->queue, value);

hashMapPut(obj->map, value);

}

// Function to free the FirstUnique object

void firstUniqueFree(FirstUnique\* obj) {

ListNode\* current = obj->queue->front;

while (current != NULL) {

ListNode\* temp = current;

current = current->next;

free(temp);

}

free(obj->queue);

hashMapFree(obj->map);

free(obj);

}

// Example usage

int main() {

int nums[] = {2, 3, 5};

FirstUnique\* firstUnique = firstUniqueCreate(nums, 3);

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 2

firstUniqueAdd(firstUnique, 5); // the queue is now [2, 3, 5, 5]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 2

firstUniqueAdd(firstUnique, 2); // the queue is now [2, 3, 5, 5, 2]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return 3

firstUniqueAdd(firstUnique, 3); // the queue is now [2, 3, 5, 5, 2, 3]

printf("First unique: %d\n", firstUniqueShowFirstUnique(firstUnique)); // return -1

firstUniqueFree(firstUnique);

return 0;

}

5. #include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition for a binary tree node.

typedef struct TreeNode {

int val;

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

// Function to create a new tree node

TreeNode\* createTreeNode(int val) {

TreeNode\* newNode = (TreeNode\*)malloc(sizeof(TreeNode));

newNode->val = val;

newNode->left = newNode->right = NULL;

return newNode;

}

// Helper function to check if a given array is a valid sequence from root to leaf

bool isValidSequenceHelper(TreeNode\* root, int\* arr, int arrSize, int index) {

if (root == NULL) return false;

if (index >= arrSize || root->val != arr[index]) return false;

// If this is a leaf node

if (root->left == NULL && root->right == NULL) {

return index == arrSize - 1;

}

// Check the left and right subtree

return isValidSequenceHelper(root->left, arr, arrSize, index + 1) ||

isValidSequenceHelper(root->right, arr, arrSize, index + 1);

}

// Function to check if a given array is a valid sequence from root to leaf

bool isValidSequence(TreeNode\* root, int\* arr, int arrSize) {

return isValidSequenceHelper(root, arr, arrSize, 0);

}

// Function to free the memory allocated for the binary tree

void freeTree(TreeNode\* root) {

if (root == NULL) return;

freeTree(root->left);

freeTree(root->right);

free(root);

}

// Example usage

int main() {

// Create the binary tree

TreeNode\* root = createTreeNode(0);

root->left = createTreeNode(1);

root->right = createTreeNode(0);

root->left->left = createTreeNode(0);

root->left->right = createTreeNode(1);

root->right->left = createTreeNode(0);

root->left->left->right = createTreeNode(1);

root->left->right->left = createTreeNode(0);

root->left->right->right = createTreeNode(0);

// Define the array

int arr[] = {0, 1, 0, 1};

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

// Check if the array is a valid sequence

if (isValidSequence(root, arr, arrSize)) {

printf("Output: true\n");

} else {

printf("Output: false\n");

}

// Free the memory allocated for the binary tree

freeTree(root);

return 0;

}

6. #include <stdio.h>

#include <stdbool.h>

void kidsWithCandies(int\* candies, int candiesSize, int extraCandies, bool\* result) {

int maxCandies = 0;

// Find the maximum number of candies any kid currently has

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

if (candies[i] > maxCandies) {

maxCandies = candies[i];

}

}

// Determine if each kid can have the greatest number of candies with extraCandies

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

if (candies[i] + extraCandies >= maxCandies) {

result[i] = true;

} else {

result[i] = false;

}

}

}

// Function to print the boolean array

void printBooleanArray(bool\* array, int size) {

printf("[");

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

if (i > 0) {

printf(",");

}

printf(array[i] ? "true" : "false");

}

printf("]\n");

}

int main() {

// Example 1

int candies1[] = {2, 3, 5, 1, 3};

int candiesSize1 = sizeof(candies1) / sizeof(candies1[0]);

int extraCandies1 = 3;

bool result1[candiesSize1];

kidsWithCandies(candies1, candiesSize1, extraCandies1, result1);

printf("Output: ");

printBooleanArray(result1, candiesSize1); // Output: [true,true,true,false,true]

// Example 2

int candies2[] = {4, 2, 1, 1, 2};

int candiesSize2 = sizeof(candies2) / sizeof(candies2[0]);

int extraCandies2 = 1;

bool result2[candiesSize2];

kidsWithCandies(candies2, candiesSize2, extraCandies2, result2);

printf("Output: ");

printBooleanArray(result2, candiesSize2); // Output: [true,false,false,false,false]

// Example 3

int candies3[] = {12, 1, 12};

int candiesSize3 = sizeof(candies3) / sizeof(candies3[0]);

int extraCandies3 = 10;

bool result3[candiesSize3];

kidsWithCandies(candies3, candiesSize3, extraCandies3, result3);

printf("Output: ");

printBooleanArray(result3, candiesSize3); // Output: [true,false,true]

return 0;

}

7. #include <stdio.h>

#include <string.h>

// Function to replace digits in the number

int replaceDigits(int num, int x, int y) {

char str[12];

snprintf(str, sizeof(str), "%d", num);

for (int i = 0; str[i] != '\0'; i++) {

if (str[i] == '0' + x) {

str[i] = '0' + y;

}

}

// Avoid leading zeros

if (str[0] == '0') return num;

return atoi(str);

}

int maxDifference(int num) {

int a = num, b = num;

char str[12];

snprintf(str, sizeof(str), "%d", num);

// Find the first non-9 digit for maximum number

for (int i = 0; str[i] != '\0'; i++) {

if (str[i] != '9') {

a = replaceDigits(num, str[i] - '0', 9);

break;

}

}

// For minimum number, consider first digit separately if it's not 1

if (str[0] != '1') {

b = replaceDigits(num, str[0] - '0', 1);

} else {

for (int i = 1; str[i] != '\0'; i++) {

if (str[i] != '0' && str[i] != '1') {

b = replaceDigits(num, str[i] - '0', 0);

break;

}

}

}

return a - b;

}

int main() {

int num1 = 555;

int num2 = 9;

printf("Input: %d\nOutput: %d\n", num1, maxDifference(num1)); // Output: 888

printf("Input: %d\nOutput: %d\n", num2, maxDifference(num2)); // Output: 8

return 0;

}

8. #include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

// Function to compare characters for qsort

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

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

}

// Function to check if one sorted string can break another sorted string

bool canBreak(char\* s1, char\* s2, int n) {

bool s1CanBreakS2 = true;

bool s2CanBreakS1 = true;

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

if (s1[i] < s2[i]) {

s1CanBreakS2 = false;

}

if (s2[i] < s1[i]) {

s2CanBreakS1 = false;

}

}

return s1CanBreakS2 || s2CanBreakS1;

}

// Main function to check if one permutation of s1 can break one permutation of s2

bool checkIfCanBreak(char\* s1, char\* s2) {

int n = strlen(s1);

// Sort both strings

qsort(s1, n, sizeof(char), compareChars);

qsort(s2, n, sizeof(char), compareChars);

return canBreak(s1, s2, n);

}

// Example usage

int main() {

char s1[] = "abc";

char s2[] = "xya";

if (checkIfCanBreak(s1, s2)) {

printf("Output: true\n");

} else {

printf("Output: false\n");

}

char s3[] = "abe";

char s4[] = "acd";

if (checkIfCanBreak(s3, s4)) {

printf("Output: true\n");

} else {

printf("Output: false\n");

}

char s5[] = "leetcodee";

char s6[] = "interview";

if (checkIfCanBreak(s5, s6)) {

printf("Output: true\n");

} else {

printf("Output: false\n");

}

return 0;

}

9. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MOD 1000000007

// A function to count the number of ways to assign hats using dynamic programming and bitmasking

int countWays(int\*\* hats, int\* hatsSize, int n) {

int dp[1 << n];

memset(dp, 0, sizeof(dp));

dp[0] = 1;

for (int hat = 1; hat <= 40; hat++) {

int new\_dp[1 << n];

memcpy(new\_dp, dp, sizeof(dp));

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

for (int j = 0; j < hatsSize[person]; j++) {

if (hats[person][j] == hat) {

for (int mask = 0; mask < (1 << n); mask++) {

if (!(mask & (1 << person))) {

new\_dp[mask | (1 << person)] = (new\_dp[mask | (1 << person)] + dp[mask]) % MOD;

}

}

}

}

}

memcpy(dp, new\_dp, sizeof(dp));

}

return dp[(1 << n) - 1];

}

int main() {

// Example 1

int hats1[3][3] = {{3, 4}, {4, 5}, {5}};

int\* hats1Ptrs[3] = {hats1[0], hats1[1], hats1[2]};

int hatsSize1[3] = {2, 2, 1};

printf("Output: %d\n", countWays(hats1Ptrs, hatsSize1, 3)); // Output: 1

// Example 2

int hats2[2][3] = {{3, 5, 1}, {3, 5}};

int\* hats2Ptrs[2] = {hats2[0], hats2[1]};

int hatsSize2[2] = {3, 2};

printf("Output: %d\n", countWays(hats2Ptrs, hatsSize2, 2)); // Output: 4

// Example 3

int hats3[4][4] = {{1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}};

int\* hats3Ptrs[4] = {hats3[0], hats3[1], hats3[2], hats3[3]};

int hatsSize3[4] = {4, 4, 4, 4};

printf("Output: %d\n", countWays(hats3Ptrs, hatsSize3, 4)); // Output: 24

return 0;

}

10. #include <stdio.h>

// Function to swap two elements

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to reverse elements from start to end

void reverse(int\* nums, int start, int end) {

while (start < end) {

swap(&nums[start], &nums[end]);

start++;

end--;

}

}

// Function to find the next permutation

void nextPermutation(int\* nums, int numsSize) {

if (numsSize <= 1) return;

int i = numsSize - 2;

// Find the first decreasing element

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

i--;

}

if (i >= 0) {

int j = numsSize - 1;

// Find the element just larger than nums[i]

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

j--;

}

// Swap them

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

}

// Reverse the elements from i+1 to the end of the array

reverse(nums, i + 1, numsSize - 1);

}

// Helper function to print the array

void printArray(int\* nums, int numsSize) {

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

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

}

printf("\n");

}

int main() {

int nums1[] = {1, 2, 3};

int numsSize1 = sizeof(nums1) / sizeof(nums1[0]);

nextPermutation(nums1, numsSize1);

printf("Next permutation: ");

printArray(nums1, numsSize1); // Output: 1 3 2

int nums2[] = {3, 2, 1};

int numsSize2 = sizeof(nums2) / sizeof(nums2[0]);

nextPermutation(nums2, numsSize2);

printf("Next permutation: ");

printArray(nums2, numsSize2); // Output: 1 2 3

int nums3[] = {1, 1, 5};

int numsSize3 = sizeof(nums3) / sizeof(nums3[0]);

nextPermutation(nums3, numsSize3);

printf("Next permutation: ");

printArray(nums3, numsSize3); // Output: 1 5 1

return 0;

}