1.#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

int val;

struct TreeNode \*left;

struct TreeNode \*right;

};

// Utility function to create a new tree node

struct TreeNode\* createNode(int val) {

struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

node->val = val;

node->left = node->right = NULL;

return node;

}

// Function to calculate the height of a binary tree

int height(struct TreeNode\* root) {

if (root == NULL) {

return -1;

}

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

// Function to find the node with a given value

struct TreeNode\* findNode(struct TreeNode\* root, int value) {

if (root == NULL || root->val == value) {

return root;

}

struct TreeNode\* left = findNode(root->left, value);

if (left != NULL) {

return left;

}

return findNode(root->right, value);

}

// Function to remove the subtree rooted at a given node

void removeSubtree(struct TreeNode\*\* root, int value) {

struct TreeNode\* node = findNode(\*root, value);

if (node != NULL) {

node->left = node->right = NULL;

}

}

int main() {

// Create the binary tree from the example

struct TreeNode\* root = createNode(1);

root->left = createNode(3);

root->right = createNode(4);

root->left->left = createNode(2);

root->right->left = createNode(6);

root->right->right = createNode(5);

root->right->left->right = createNode(7);

// Queries array

int queries[] = {4};

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

// Result array to store the heights

int answer[m];

// Process each query

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

int value = queries[i];

// Remove the subtree

removeSubtree(&root, value);

// Calculate the height of the modified tree

answer[i] = height(root);

// Restore the tree (recreate the subtree)

// In a real scenario, you would need to restore the tree from a saved state.

// Here, for simplicity, we recreate the tree as we know the original structure.

root->right->left = createNode(6);

root->right->right = createNode(5);

root->right->left->right = createNode(7);

}

// Print the result

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

printf("%d\n", answer[i]);

}

return 0;

}

2. #include <stdio.h>

#include <stdbool.h>

// Function to find the index of 0 (empty space) in the array

int findEmptySpace(int\* nums, int n) {

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

if (nums[i] == 0) {

return i;

}

}

return -1; // Should never happen since 0 is guaranteed to be in the array

}

// Function to count minimum moves to sort the array

int minMovesToSort(int\* nums, int n) {

bool visited[n];

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

visited[i] = false;

}

int moves = 0;

int emptyIndex = findEmptySpace(nums, n);

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

if (nums[i] == i || visited[i]) {

continue;

}

int cycle\_length = 0;

int j = i;

while (!visited[j]) {

visited[j] = true;

j = nums[j];

cycle\_length++;

}

if (cycle\_length > 1) {

moves += cycle\_length - 1;

}

}

if (emptyIndex != 0 && emptyIndex != n - 1) {

moves++;

}

return moves;

}

int main() {

int nums[] = {4, 2, 0, 3, 1};

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

int result = minMovesToSort(nums, n);

printf("Minimum number of moves: %d\n", result);

return 0;

}

3. #include <stdio.h>

// Function to apply operations on the array

void applyOperations(int\* nums, int n) {

// Apply the operations as described

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

if (nums[i] == nums[i + 1]) {

nums[i] \*= 2;

nums[i + 1] = 0;

}

}

// Shift all zeros to the end

int index = 0;

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

if (nums[i] != 0) {

nums[index++] = nums[i];

}

}

while (index < n) {

nums[index++] = 0;

}

}

// Function to print the array

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

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

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

}

printf("\n");

}

int main() {

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

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

applyOperations(nums, n);

printf("Resulting array: ");

printArray(nums, n);

return 0;

}

4. #include <stdio.h>

#include <stdlib.h>

// Function to find the maximum sum of distinct subarrays with length k

int maxSumOfDistinctSubarrays(int\* nums, int n, int k) {

int maxSum = 0;

int currentSum = 0;

int start = 0;

int count[100001] = {0}; // Assuming nums[i] <= 100000

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

currentSum += nums[end];

count[nums[end]]++;

while (count[nums[end]] > 1) {

count[nums[start]]--;

currentSum -= nums[start];

start++;

}

if (end - start + 1 == k) {

if (currentSum > maxSum) {

maxSum = currentSum;

}

count[nums[start]]--;

currentSum -= nums[start];

start++;

}

}

return maxSum;

}

int main() {

int nums[] = {1, 5, 4, 2, 9, 9, 9};

int k = 3;

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

int result = maxSumOfDistinctSubarrays(nums, n, k);

printf("Maximum sum of distinct subarrays with length %d is: %d\n", k, result);

return 0;

}

5. #include <stdio.h>

#include <stdlib.h>

typedef struct {

int cost;

int index;

} Worker;

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

Worker\* w1 = (Worker\*)a;

Worker\* w2 = (Worker\*)b;

if (w1->cost == w2->cost) {

return w1->index - w2->index;

}

return w1->cost - w2->cost;

}

int totalCostToHireKWorkers(int\* costs, int n, int k, int candidates) {

Worker\* firstCandidates = (Worker\*)malloc(sizeof(Worker) \* candidates);

Worker\* lastCandidates = (Worker\*)malloc(sizeof(Worker) \* candidates);

int firstCount = 0, lastCount = 0;

int firstIndex = 0, lastIndex = n - 1;

// Initialize first and last candidates

while (firstCount < candidates && firstIndex < n) {

firstCandidates[firstCount].cost = costs[firstIndex];

firstCandidates[firstCount].index = firstIndex;

firstCount++;

firstIndex++;

}

while (lastCount < candidates && lastIndex >= 0) {

lastCandidates[lastCount].cost = costs[lastIndex];

lastCandidates[lastCount].index = lastIndex;

lastCount++;

lastIndex--;

}

// Sort initial candidates by cost (and index for ties)

qsort(firstCandidates, firstCount, sizeof(Worker), compare);

qsort(lastCandidates, lastCount, sizeof(Worker), compare);

int totalCost = 0;

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

if (firstCount > 0 && (lastCount == 0 || firstCandidates[0].cost < lastCandidates[0].cost ||

(firstCandidates[0].cost == lastCandidates[0].cost && firstCandidates[0].index < lastCandidates[0].index))) {

// Hire from first candidates

totalCost += firstCandidates[0].cost;

// Shift all elements in the firstCandidates array to the left

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

firstCandidates[j - 1] = firstCandidates[j];

}

firstCount--;

// Add new candidate from the front part of the array if available

if (firstIndex <= lastIndex) {

firstCandidates[firstCount].cost = costs[firstIndex];

firstCandidates[firstCount].index = firstIndex;

firstCount++;

firstIndex++;

qsort(firstCandidates, firstCount, sizeof(Worker), compare);

}

} else {

// Hire from last candidates

totalCost += lastCandidates[0].cost;

// Shift all elements in the lastCandidates array to the left

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

lastCandidates[j - 1] = lastCandidates[j];

}

lastCount--;

// Add new candidate from the back part of the array if available

if (lastIndex >= firstIndex) {

lastCandidates[lastCount].cost = costs[lastIndex];

lastCandidates[lastCount].index = lastIndex;

lastCount++;

lastIndex--;

qsort(lastCandidates, lastCount, sizeof(Worker), compare);

}

}

}

free(firstCandidates);

free(lastCandidates);

return totalCost;

}

int main() {

int costs[] = {17, 12, 10, 2, 7, 2, 11, 20, 8};

int k = 3;

int candidates = 4;

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

int result = totalCostToHireKWorkers(costs, n, k, candidates);

printf("Total cost to hire %d workers: %d\n", k, result);

return 0;

}

6. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

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

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

}

int min(int a, int b) {

return (a < b) ? a : b;

}

int minimumTotalDistance(int\* robots, int robotSize, int\*\* factories, int factorySize) {

qsort(robots, robotSize, sizeof(int), compare);

int\* factoryPositions = (int\*)malloc(factorySize \* sizeof(int));

int\* factoryLimits = (int\*)malloc(factorySize \* sizeof(int));

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

factoryPositions[i] = factories[i][0];

factoryLimits[i] = factories[i][1];

}

qsort(factoryPositions, factorySize, sizeof(int), compare);

int\*\* dp = (int\*\*)malloc((robotSize + 1) \* sizeof(int\*));

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

dp[i] = (int\*)malloc((factorySize + 1) \* sizeof(int));

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

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

}

}

dp[0][0] = 0;

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

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

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

int distance = 0;

for (int k = 1; k <= factoryLimits[j - 1] && i - k >= 0; k++) {

distance += abs(robots[i - k] - factoryPositions[j - 1]);

dp[i][j] = min(dp[i][j], dp[i - k][j - 1] + distance);

}

}

}

int result = dp[robotSize][factorySize];

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

free(dp[i]);

}

free(dp);

free(factoryPositions);

free(factoryLimits);

return result;

}

int main() {

int robots[] = {0, 4, 6};

int robotSize = sizeof(robots) / sizeof(robots[0]);

int factoryArray[][2] = {{2, 2}, {6, 2}};

int\* factories[2];

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

factories[i] = factoryArray[i];

}

int factorySize = sizeof(factories) / sizeof(factories[0]);

int result = minimumTotalDistance(robots, robotSize, factories, factorySize);

printf("Minimum total distance: %d\n", result);

return 0;

}

7. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Function to compute the GCD of two numbers

int gcd(int a, int b) {

while (b != 0) {

int t = b;

b = a % b;

a = t;

}

return a;

}

// Function to find the minimum number of valid subarrays

int minValidSplitSubarrays(int\* nums, int numsSize) {

int\* dp = (int\*)malloc(numsSize \* sizeof(int));

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

dp[i] = INT\_MAX; // Initialize dp array with a large number

}

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

int g = 0;

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

g = gcd(g, nums[j]);

if (g > 1) {

if (j == 0) {

dp[i] = 1;

} else {

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

}

}

}

}

int result = (dp[numsSize - 1] == INT\_MAX) ? -1 : dp[numsSize - 1];

free(dp);

return result;

}

int main() {

int nums1[] = {2, 6, 3, 4, 3};

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

printf("Minimum subarrays for nums1: %d\n", minValidSplitSubarrays(nums1, numsSize1));

int nums2[] = {3, 5};

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

printf("Minimum subarrays for nums2: %d\n", minValidSplitSubarrays(nums2, numsSize2));

return 0;

}

8. #include <stdio.h>

#include <stdlib.h>

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

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

}

int distinctAverages(int\* nums, int numsSize) {

qsort(nums, numsSize, sizeof(int), compare);

double\* averages = (double\*)malloc(numsSize / 2 \* sizeof(double));

int avgCount = 0;

int left = 0;

int right = numsSize - 1;

while (left < right) {

double avg = (nums[left] + nums[right]) / 2.0;

int distinct = 1;

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

if (averages[i] == avg) {

distinct = 0;

break;

}

}

if (distinct) {

averages[avgCount++] = avg;

}

left++;

right--;

}

free(averages);

return avgCount;

}

int main() {

int nums1[] = {4, 1, 4, 0, 3, 5};

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

printf("Number of distinct averages for nums1: %d\n", distinctAverages(nums1, numsSize1));

int nums2[] = {1, 100};

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

printf("Number of distinct averages for nums2: %d\n", distinctAverages(nums2, numsSize2));

return 0;

}

9. #include <stdio.h>

#include <stdlib.h>

#define MOD 1000000007

int countGoodStrings(int low, int high, int zero, int one) {

int \*dp = (int \*)calloc(high + 1, sizeof(int));

dp[0] = 1;

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

if (i >= zero) {

dp[i] = (dp[i] + dp[i - zero]) % MOD;

}

if (i >= one) {

dp[i] = (dp[i] + dp[i - one]) % MOD;

}

}

int result = 0;

for (int i = low; i <= high; i++) {

result = (result + dp[i]) % MOD;

}

free(dp);

return result;

}

int main() {

int low1 = 3, high1 = 3, zero1 = 1, one1 = 1;

printf("Number of good strings (Example 1): %d\n", countGoodStrings(low1, high1, zero1, one1));

int low2 = 2, high2 = 3, zero2 = 1, one2 = 2;

printf("Number of good strings (Example 2): %d\n", countGoodStrings(low2, high2, zero2, one2));

return 0;

}

10. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <math.h> // Include math.h for fmax

#define MAXN 100005

typedef struct {

int \*data;

int size;

} Vector;

void initVector(Vector \*vec, int capacity) {

vec->data = (int \*)malloc(capacity \* sizeof(int));

vec->size = 0;

}

void pushBack(Vector \*vec, int value) {

vec->data[vec->size++] = value;

}

int n;

int amount[MAXN];

Vector adj[MAXN];

int bobTime[MAXN];

void dfsBob(int node, int parent, int time) {

bobTime[node] = time;

for (int i = 0; i < adj[node].size; i++) {

int next = adj[node].data[i];

if (next != parent) {

dfsBob(next, node, time + 1);

}

}

}

int dfsAlice(int node, int parent, int time) {

int profit = (bobTime[node] == -1 || bobTime[node] > time) ? amount[node] :

(bobTime[node] == time) ? amount[node] / 2 : 0;

int maxSubProfit = INT\_MIN;

for (int i = 0; i < adj[node].size; i++) {

int next = adj[node].data[i];

if (next != parent) {

maxSubProfit = fmax(maxSubProfit, dfsAlice(next, node, time + 1));

}

}

if (maxSubProfit == INT\_MIN) {

return profit;

} else {

return profit + maxSubProfit;

}

}

int mostProfitablePath(int edges[][2], int edgesSize, int amountArr[], int amountSize, int bobPos) {

n = amountSize;

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

amount[i] = amountArr[i];

initVector(&adj[i], n - 1);

bobTime[i] = -1;

}

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

int u = edges[i][0];

int v = edges[i][1];

pushBack(&adj[u], v);

pushBack(&adj[v], u);

}

dfsBob(bobPos, -1, 0);

return dfsAlice(0, -1, 0);

}

int main() {

int edges1[][2] = {{0, 1}, {1, 2}, {1, 3}, {3, 4}};

int amount1[] = {-2, 4, 2, -4, 6};

int bob1 = 3;

int edgesSize1 = sizeof(edges1) / sizeof(edges1[0]);

int amountSize1 = sizeof(amount1) / sizeof(amount1[0]);

printf("Most Profitable Path (Example 1): %d\n", mostProfitablePath(edges1, edgesSize1, amount1, amountSize1, bob1));

int edges2[][2] = {{0, 1}};

int amount2[] = {-7280, 2350};

int bob2 = 1;

int edgesSize2 = sizeof(edges2) / sizeof(edges2[0]);

int amountSize2 = sizeof(amount2) / sizeof(amount2[0]);

printf("Most Profitable Path (Example 2): %d\n", mostProfitablePath(edges2, edgesSize2, amount2, amountSize2, bob2));

return 0;

}