1) Find the missing number in a given integer array of 1 to 500.

WTD: Examine an array expected to contain consecutive integers from 1 to 500. Identify any integer that is missing from this sequence.
(e.g.: I/P: [1,2,4,5]; O/P: 3)

```c
#include <stdio.h>

int findMissingNumber(int arr[], int n) {
    // Calculate the expected sum of consecutive integers from 1 to 500
    int expectedSum = (500 * (500 + 1)) / 2; // Sum of an arithmetic series

    // Calculate the actual sum of the elements in the array
    int actualSum = 0;
    for (int i = 0; i < n; i++) {
        actualSum += arr[i];
    }

    // The missing number is the difference between the expected sum and the actual sum
    int missingNumber = expectedSum - actualSum;
    return missingNumber;
}

int main() {
    int arr[] = {1, 2, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    int missingNumber = findMissingNumber(arr, n);

    printf("Missing Number: %d\n", missingNumber);

    return 0;
}
```

2) Find the duplicate number on a given integer array.

WTD: Inspect the provided array. Determine if there's any integer that appears more frequently than it should, signifying a duplicate.

(e.g.: I/P: [3,1,3,4,2]; O/P: 3)

```
#include <stdio.h>

int findDuplicate(int arr[], int n) {
    // Initialize a hash set to store seen elements
    int hashSet[501] = {0}; // Assuming the array contains numbers from 1
to 500

    for (int i = 0; i < n; i++) {
        if (hashSet[arr[i]]) {
            // If the element is already in the hash set, it's a duplicate
            return arr[i];
        }
        // Mark the element as seen
        hashSet[arr[i]] = 1;
    }

    // If no duplicate is found, return -1
    return -1;
}

int main() {
    int arr[] = {3, 1, 3, 4, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    int duplicate = findDuplicate(arr, n);

    if (duplicate != -1) {
        printf("Duplicate Number: %d\n", duplicate);
    } else {
        printf("No duplicate found.\n");
    }
    return 0;
}
```

3) Find the largest and smallest number in an unsorted integer array.

WTD: Navigate through the elements of the unsorted array, continuously updating the largest and smallest values found to identify the extremities in the array.

(e.g.: I/P: [34, 15, 88, 2]; O/P: Max: 88, Min: 2)

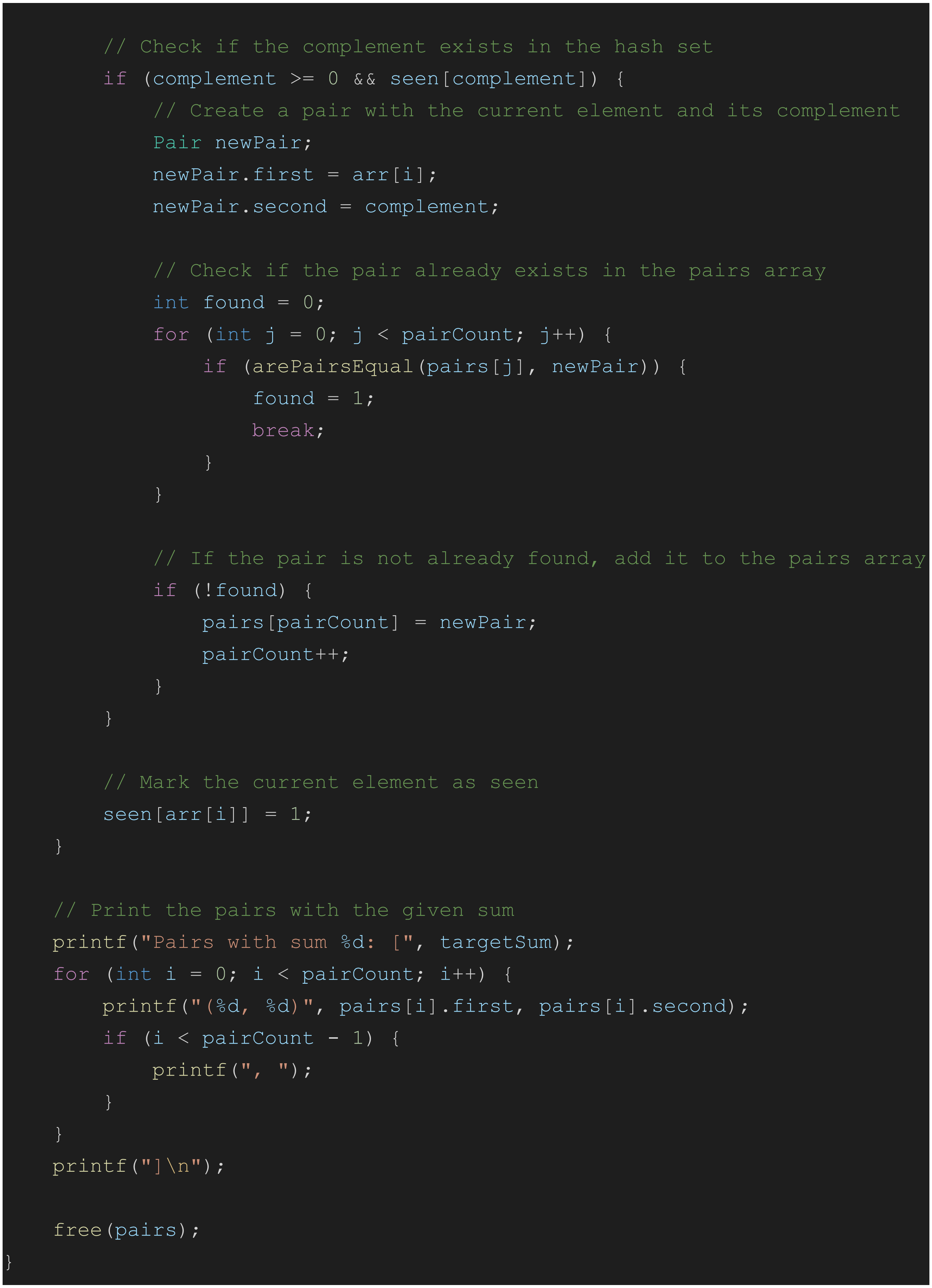
```
#include <stdio.h>

void findMinMax(int arr[], int n, int *min, int *max) {
    // Initialize min and max with the first element of the array
    *min = *max = arr[0];

    // Iterate through the array to update min and max
    for (int i = 1; i < n; i++) {
        if (arr[i] < *min) {
            // Update min if a smaller value is found
            *min = arr[i];
        } else if (arr[i] > *max) {
            // Update max if a larger value is found
            *max = arr[i];
        }
    }
}

int main() {
    int arr[] = {34, 15, 88, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    int min, max;
    findMinMax(arr, n, &min, &max);

    printf("Max: %d, Min: %d\n", max, min);
    return 0;
}

```

4) Find all pairs of an integer array whose sum is equal to a given number.

WTD: Explore combinations of integer pairs in the array. Check if the sum of any of these pairs matches a specified target number.

(e.g.: I/P: [2,4,3,5,6,-2,4,7,8,9], Sum: 7; O/P: [2,5],[4,3])

```
#include <stdio.h>
#include <stdlib.h>

// Structure to represent a pair of integers
typedef struct {
    int first;
    int second;
} Pair;

// Function to compare two pairs for equality
int arePairsEqual(Pair p1, Pair p2) {
    return (p1.first == p2.first && p1.second == p2.second) ||
           (p1.first == p2.second && p1.second == p2.first);
}

// Function to find all pairs with a given sum
void findPairsWithSum(int arr[], int n, int targetSum) {
    // Create a hash set to store seen elements
    int seen[1000] = {0}; // Assuming a reasonable range of input values

    // Create an array to store pairs
    Pair *pairs = malloc(sizeof(Pair) * n);
    int pairCount = 0;

    // Iterate through the array
    for (int i = 0; i < n; i++) {
        int complement = targetSum - arr[i];
```



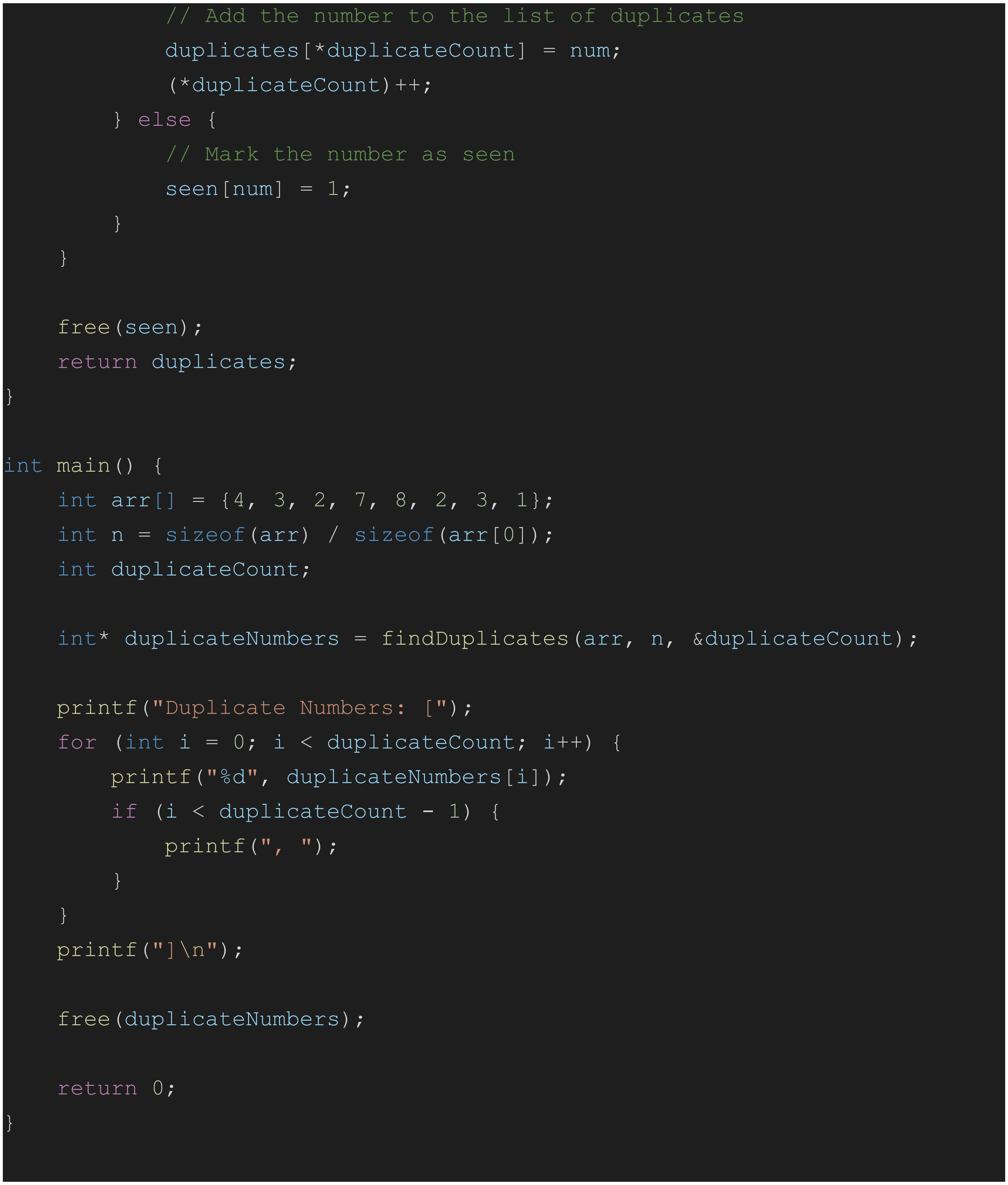
```
        // Check if the complement exists in the hash set
        if (complement >= 0 && seen[complement]) {
            // Create a pair with the current element and its complement
            Pair newPair;
            newPair.first = arr[i];
            newPair.second = complement;

            // Check if the pair already exists in the pairs array
            int found = 0;
            for (int j = 0; j < pairCount; j++) {
                if (arePairsEqual(pairs[j], newPair)) {
                    found = 1;
                    break;
                }
            }

            // If the pair is not already found, add it to the pairs array
            if (!found) {
                pairs[pairCount] = newPair;
                pairCount++;
            }
        }

        // Mark the current element as seen
        seen[arr[i]] = 1;
    }

    // Print the pairs with the given sum
    printf("Pairs with sum %d: [", targetSum);
    for (int i = 0; i < pairCount; i++) {
        printf("(%d, %d)", pairs[i].first, pairs[i].second);
        if (i < pairCount - 1) {
            printf(", ");
        }
    }
    printf("]\n");

    free(pairs);
}
```

```
int main() {
    int arr[] = {2, 4, 3, 5, 6, -2, 4, 7, 8, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int targetSum = 7;

    findPairsWithSum(arr, n, targetSum);

    return 0;
}
```

5) Find duplicate numbers in an array if it contains multiple duplicates.

WTD: Examine the array to identify numbers that appear more than once. Compile a list of these repetitive numbers.
 (e.g.: I/P: [4,3,2,7,8,2,3,1]; O/P: [2,3])

```
#include <stdio.h>
#include <stdlib.h>

// Function to find duplicate numbers in an array
int* findDuplicates(int arr[], int n, int* duplicateCount) {
    // Create a hash set to store seen elements
    int* seen = (int*)calloc(1000, sizeof(int)); // Assuming a reasonable range of input values

    // Create an array to store duplicate numbers
    int* duplicates = (int*)malloc(n * sizeof(int));
    *duplicateCount = 0;

    // Iterate through the array
    for (int i = 0; i < n; i++) {
        int num = arr[i];

        // Check if the number is already in the hash set
        if (seen[num]) {
```



```
            // Add the number to the list of duplicates
            duplicates[*duplicateCount] = num;
            (*duplicateCount)++;
        } else {
            // Mark the number as seen
            seen[num] = 1;
        }
    }

    free(seen);
    return duplicates;
}

int main() {
    int arr[] = {4, 3, 2, 7, 8, 2, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int duplicateCount;

    int* duplicateNumbers = findDuplicates(arr, n, &duplicateCount);

    printf("Duplicate Numbers: [");
    for (int i = 0; i < duplicateCount; i++) {
        printf("%d", duplicateNumbers[i]);
        if (i < duplicateCount - 1) {
            printf(", ");
        }
    }
    printf("]\n");

    free(duplicateNumbers);

    return 0;
}
```

6) Sort an array using the quicksort algorithm.

WTD: Implement the quicksort sorting technique on the provided array to rearrange its elements in ascending order.

(e.g.: I/P: [64, 34, 25, 12, 22, 11, 90]; O/P: [11, 12, 22, 25, 34, 64, 90])

```
#include <stdio.h>

// Function to swap two elements in an array
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the last element as the pivot
    int i = (low - 1);      // Index of the smaller element

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment the index of the smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to perform Quicksort
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array and get the pivot index
        int pivotIndex = partition(arr, low, high);

        // Recursively sort the elements before and after the pivot
        quicksort(arr, low, pivotIndex - 1);
        quicksort(arr, pivotIndex + 1, high);
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Input Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    quicksort(arr, 0, n - 1);

    printf("Sorted Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

7) Remove duplicates from an array without using any library.

WTD: Navigate through the array, identifying and removing any repetitive occurrences of numbers, ensuring each number appears only once.
(e.g.: I/P: [1,1,2,2,3,4,4]; O/P: [1,2,3,4])

```
#include <stdio.h>

void removeDuplicates(int arr[], int *n) {
    if (*n <= 1) {
        return; // Nothing to remove if the array has 0 or 1 elements
    }

    int uniqueIndex = 1; // Index to track unique elements

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

        // Check if the current element is a duplicate of any previous
elements
        for (int j = 0; j < uniqueIndex; j++) {
            if (arr[i] == arr[j]) {
                isDuplicate = 1;
                break;
            }
        }

        // If it's not a duplicate, add it to the unique elements section
        if (!isDuplicate) {
            arr[uniqueIndex] = arr[i];
            uniqueIndex++;
        }
    }

    // Update the size of the array to reflect the number of unique
elements
    *n = uniqueIndex;
}

int main() {
    int arr[] = {1, 1, 2, 2, 3, 4, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    removeDuplicates(arr, &n);
    printf("Array with Duplicates Removed: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

8) Determine the intersection of two integer arrays.

WTD: Compare every element of the two arrays, listing down the common integers that appear in both.
(e.g.: I/P: [1,2,4,5,6], [2,3,5,7]; O/P: [2,5])

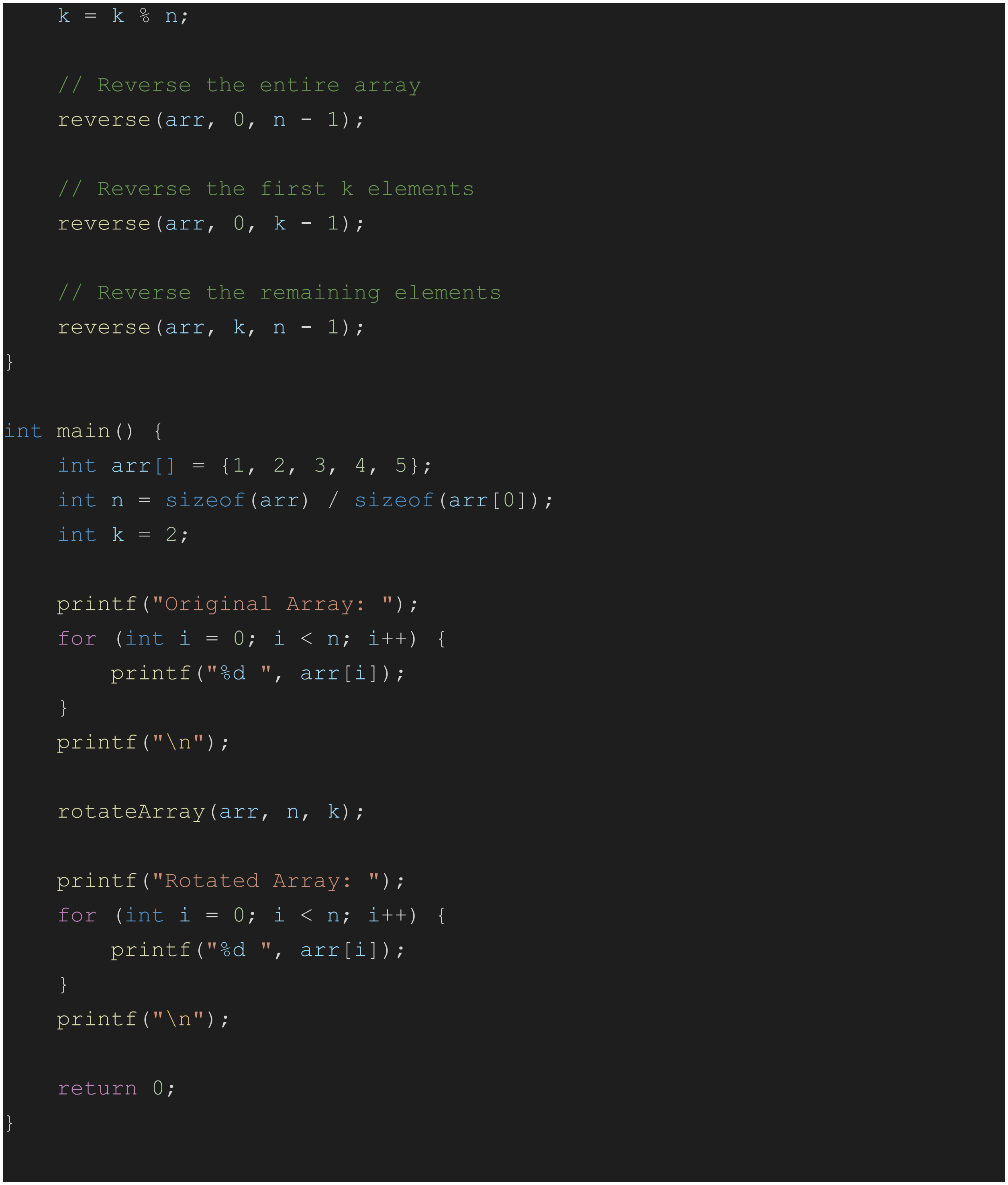
```
#include <stdio.h>

// Function to check if an element is in the array
int isInArray(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return 1; // Element found in the array
        }
    }
    return 0; // Element not found in the array
}

// Function to find the intersection of two arrays
void findIntersection(int arr1[], int n1, int arr2[], int n2) {
    printf("Intersection: [");
    for (int i = 0; i < n1; i++) {
        if (isInArray(arr2, n2, arr1[i])) {
            printf("%d", arr1[i]);
            if (i < n1 - 1) {
                printf(", ");
            }
        }
    }
    printf("]\n");
}

int main() {
    int arr1[] = {1, 2, 4, 5, 6};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    int arr2[] = {2, 3, 5, 7};
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    findIntersection(arr1, n1, arr2, n2);

    return 0;
}
```

9) Rotate an array to the right by k steps.

WTD: Modify the array by moving its elements to the right, wrapping them around when they reach the end, for a specified number of steps.

(e.g.: I/P: [1,2,3,4,5], k=2; O/P: [4,5,1,2,3])

```
#include <stdio.h>

// Function to reverse an array or a subarray
void reverse(int arr[], int start, int end) {
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Function to rotate an array to the right by k steps
void rotateArray(int arr[], int n, int k) {
    // Handle the case where k is greater than the array size
```

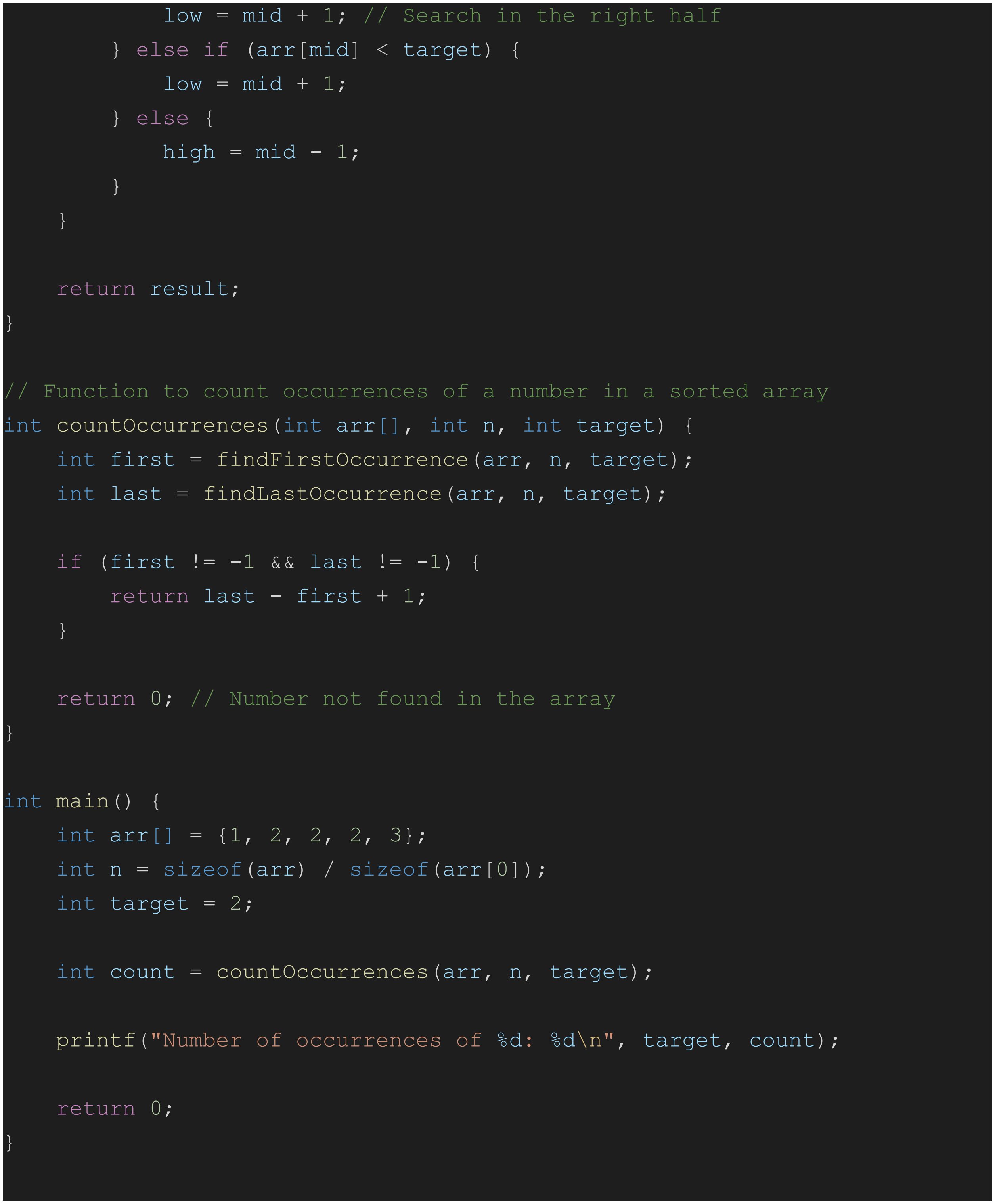


```
    k = k % n;

    // Reverse the entire array
    reverse(arr, 0, n - 1);

    // Reverse the first k elements
    reverse(arr, 0, k - 1);

    // Reverse the remaining elements
    reverse(arr, k, n - 1);
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;

    printf("Original Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    rotateArray(arr, n, k);

    printf("Rotated Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

10) Count occurrences of a number in a sorted array.

WTD: For a given number and a sorted array, iterate through the array to count the number of times that particular number appears.
(e.g.: I/P: [1, 2, 2, 2, 3], 2; O/P: 3)

```
#include <stdio.h>

// Function to find the first occurrence of a number in a sorted array
int findFirstOccurrence(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;
    int result = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            result = mid;
            high = mid - 1; // Search in the left half
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return result;
}

// Function to find the last occurrence of a number in a sorted array
int findLastOccurrence(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;
    int result = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            result = mid;
```



```
            low = mid + 1; // Search in the right half
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return result;
}

// Function to count occurrences of a number in a sorted array
int countOccurrences(int arr[], int n, int target) {
    int first = findFirstOccurrence(arr, n, target);
    int last = findLastOccurrence(arr, n, target);

    if (first != -1 && last != -1) {
        return last - first + 1;
    }

    return 0; // Number not found in the array
}

int main() {
    int arr[] = {1, 2, 2, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 2;

    int count = countOccurrences(arr, n, target);

    printf("Number of occurrences of %d: %d\n", target, count);

    return 0;
}
```

11) Find the "Kth" max and min element of an array.

WTD: Sort the array and retrieve the kth largest and kth smallest numbers.
(e.g.: I/P: [7, 10, 4, 3, 20, 15], K=3; O/P: 7)

```c
#include <stdio.h>

// Function to perform Quicksort
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {
            if (arr[j] >= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        int pivotIndex = i + 1;

        quicksort(arr, low, pivotIndex - 1);
        quicksort(arr, pivotIndex + 1, high);
    }
}

// Function to find the kth maximum element
int findKthMax(int arr[], int n, int k) {
    if (k <= 0 || k > n) {
        return -1; // Invalid k value
    }
    quicksort(arr, 0, n - 1);
    return arr[k - 1];
}

// Function to find the kth minimum element
int findKthMin(int arr[], int n, int k) {
    if (k <= 0 || k > n) {
        return -1; // Invalid k value
    }

    quicksort(arr, 0, n - 1);
    return arr[n - k];
}

int main() {
    int arr[] = {7, 10, 4, 3, 20, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;

    int kthMax = findKthMax(arr, n, k);
    int kthMin = findKthMin(arr, n, k);

    printf("Kth Max: %d\n", kthMax);
    printf("Kth Min: %d\n", kthMin);

    return 0;
}
```

12) Move all zeros to the left of an array while maintaining the order of other numbers.

WTD: Reorder the array by moving all zero values to the leftmost positions while ensuring the relative order of the non-zero numbers remains unchanged.
(e.g.: I/P: [1,2,0,4,3,0,5,0]; O/P: [0,0,0,1,2,4,3,5])

```
#include <stdio.h>

// Function to move all zeros to the left of the array
```

```c
void moveZerosToLeft(int arr[], int n) {
    int nonZeroIndex = n - 1; // Index to track the position of non-zero elements

    // Iterate the array from right to left
    for (int i = n - 1; i >= 0; i--) {
        if (arr[i] != 0) {
            // Move non-zero elements to the rightmost positions
            arr[nonZeroIndex] = arr[i];
            nonZeroIndex--;
        }
    }

    // Fill the remaining leftmost positions with zeros
    while (nonZeroIndex >= 0) {
        arr[nonZeroIndex] = 0;
        nonZeroIndex--;
    }
}

int main() {
    int arr[] = {1, 2, 0, 4, 3, 0, 5, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    moveZerosToLeft(arr, n);

    printf("Modified Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

13) Merge two sorted arrays to produce one sorted array.

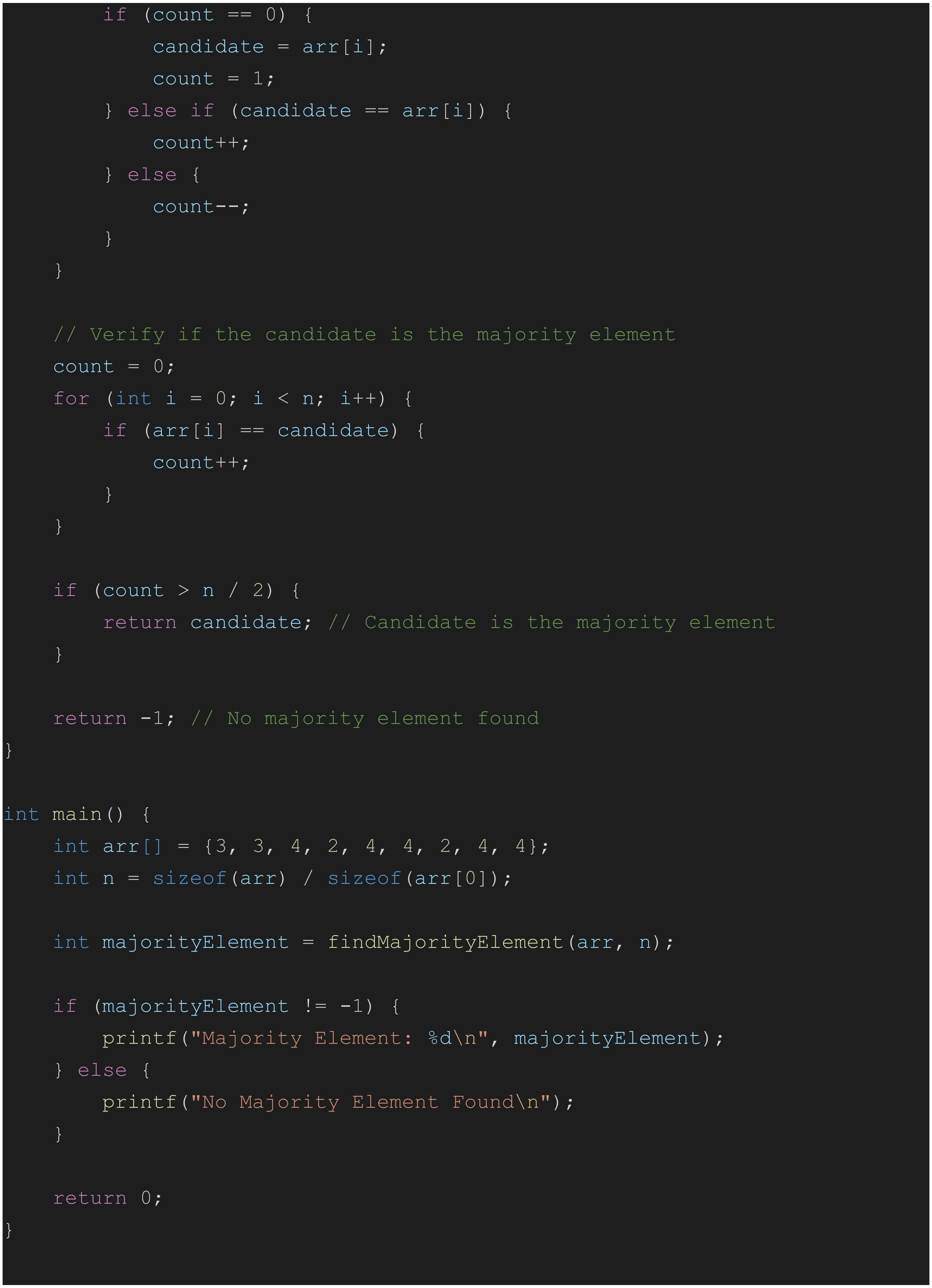
WTD: Sequentially compare the elements of two sorted arrays, combining them into a single array that remains sorted.
(e.g.: I/P: [1,3,5], [2,4,6]; O/P: [1,2,3,4,5,6])

```c
#include <stdio.h>

// Function to merge two sorted arrays into one sorted array
void mergeSortedArrays(int arr1[], int n1, int arr2[], int n2, int
mergedArr[]) {
    int i = 0, j = 0, k = 0;

    while (i < n1 && j < n2) {
        if (arr1[i] <= arr2[j]) {
            mergedArr[k] = arr1[i];
            i++;
        } else {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements from arr1
    while (i < n1) {
        mergedArr[k] = arr1[i];
        i++;
        k++;
    }

    // Copy any remaining elements from arr2
    while (j < n2) {
        mergedArr[k] = arr2[j];
        j++;
        k++;
    }
}

int main() {
    int arr1[] = {1, 3, 5};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    int arr2[] = {2, 4, 6};
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    int mergedArr[n1 + n2];

    mergeSortedArrays(arr1, n1, arr2, n2, mergedArr);

    printf("Merged Array: ");
    for (int i = 0; i < n1 + n2; i++) {
        printf("%d ", mergedArr[i]);
    }
    printf("\n");

    return 0;
}
```

14) Find the majority element in an array (appears more than n/2 times).

WTD: Traverse the array and maintain a count of each number. Identify if there's any number that appears more than half the length of the array.
(e.g.: I/P: [3,3,4,2,4,4,2,4,4]; O/P: 4)

```
#include <stdio.h>

// Function to find the majority element
int findMajorityElement(int arr[], int n) {
    int candidate = arr[0];
    int count = 1;

    // Find a candidate for the majority element
    for (int i = 1; i < n; i++) {
```



```
        if (count == 0) {
            candidate = arr[i];
            count = 1;
        } else if (candidate == arr[i]) {
            count++;
        } else {
            count--;
        }
    }

    // Verify if the candidate is the majority element
    count = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == candidate) {
            count++;
        }
    }

    if (count > n / 2) {
        return candidate; // Candidate is the majority element
    }

    return -1; // No majority element found
}

int main() {
    int arr[] = {3, 3, 4, 2, 4, 4, 2, 4, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    int majorityElement = findMajorityElement(arr, n);

    if (majorityElement != -1) {
        printf("Majority Element: %d\n", majorityElement);
    } else {
        printf("No Majority Element Found\n");
    }

    return 0;
}
```

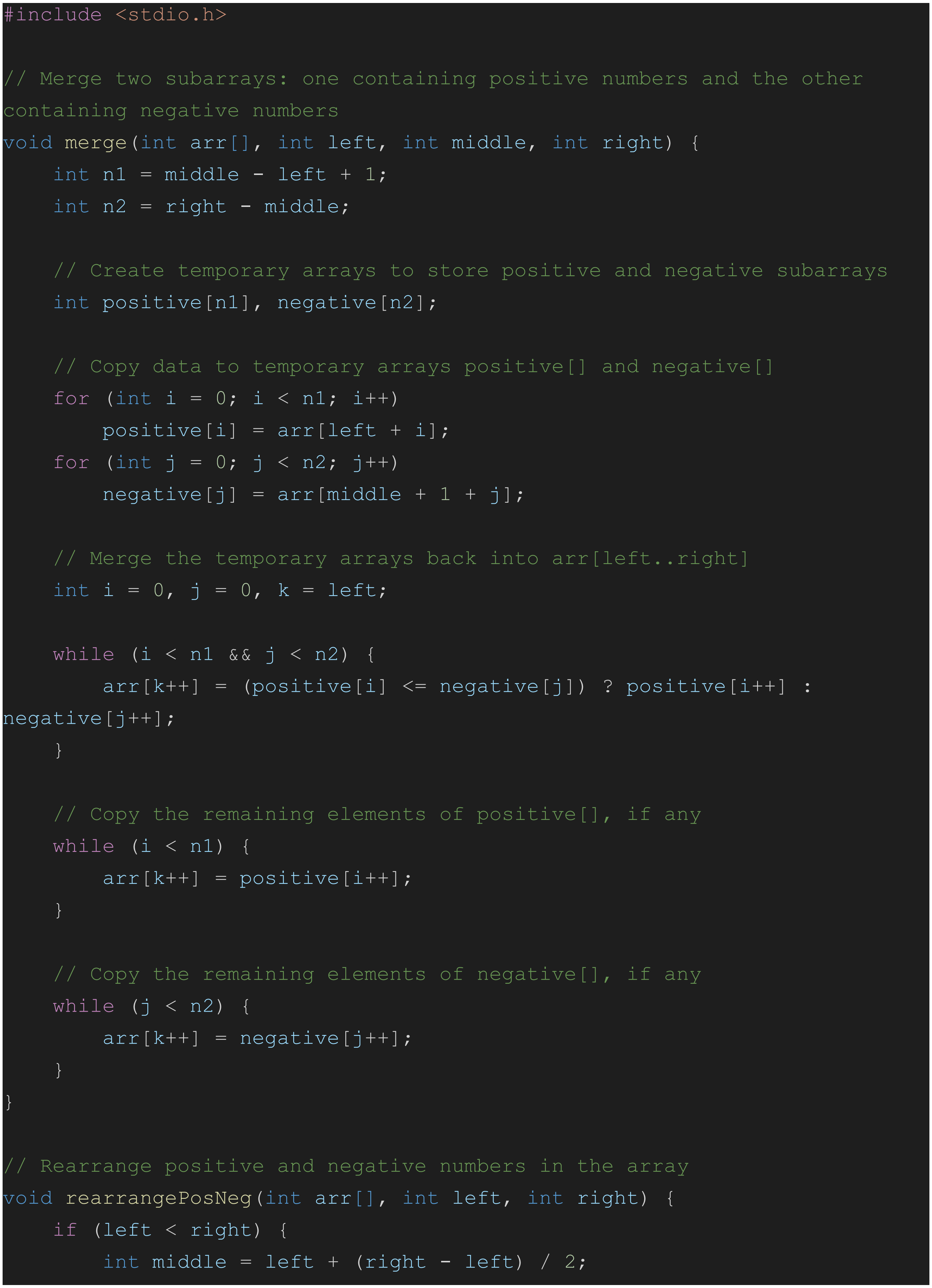
15) Find the two repeating elements in a given array.

WTD: Investigate the array and find two numbers that each appear more than once.
(e.g.: I/P: [4, 2, 4, 5, 2, 3, 1]; O/P: [4,2])

```
#include <stdio.h>
void printTwoRepeatNumber (int arr [], int size)
{
    int i, j;
    printf("Repeating elements are ");
    for (i = 0; i < size; i++)
    {
        for (j = i + 1; j < size; j++)
        {
            if (arr [i] == arr [j])
            {
                printf("%d ", arr [i]);
                break;
            }
        }
    }
}
int main ()
{
    int arr [] = { 4, 2, 4, 5, 2, 3, 1 };
    int arr_size = sizeof(arr) / sizeof(arr [0]);
    printTwoRepeatNumber (arr, arr_size);
    return 0;
}
```

16) Rearrange positive and negative numbers in an array.

WTD: Sort the array such that all the positive numbers appear before the negative ones, while maintaining their original sequence.
(e.g.: I/P: [-1, 2, -3, 4, 5, 6, -7, 8, 9]; O/P: [4,-3,5,-1,6,-7,2,8,9])



```c
#include <stdio.h>

// Merge two subarrays: one containing positive numbers and the other
containing negative numbers
void merge(int arr[], int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    // Create temporary arrays to store positive and negative subarrays
    int positive[n1], negative[n2];

    // Copy data to temporary arrays positive[] and negative[]
    for (int i = 0; i < n1; i++)
        positive[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        negative[j] = arr[middle + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        arr[k++] = (positive[i] <= negative[j]) ? positive[i++] :
negative[j++];
    }

    // Copy the remaining elements of positive[], if any
    while (i < n1) {
        arr[k++] = positive[i++];
    }

    // Copy the remaining elements of negative[], if any
    while (j < n2) {
        arr[k++] = negative[j++];
    }
}

// Rearrange positive and negative numbers in the array
void rearrangePosNeg(int arr[], int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        // Recursively sort the two halves
        rearrangePosNeg(arr, left, middle);
        rearrangePosNeg(arr, middle + 1, right);

        // Merge the sorted halves
        merge(arr, left, middle, right);
    }
}

int main() {
    int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

    rearrangePosNeg(arr, 0, n - 1);

    printf("Rearranged Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

17) Find if there's a subarray with zero sum.

WTD: Explore the array's subarrays (subsets of consecutive elements) to determine if there exists any subarray that sums up to zero.
(e.g.: I/P: [4, 2, -3, 1, 6]; O/P: True)

```
#include <stdio.h>
#include <stdbool.h>

// Function to find if there's a subarray with zero sum
bool hasZeroSumSubarray(int arr[], int n) {
    // Create a hash table to store cumulative sums and their positions
    int sum = 0;
    int hashTable[10000] = {0}; // Assuming the sum won't exceed 10000

    for (int i = 0; i < n; i++) {
        sum += arr[i];

        // If the current sum is zero or has been encountered before, there is a zero-sum subarray
        if (sum == 0 || hashTable[sum] == 1) {
            return true;
        }

        // Mark the current sum as encountered
        hashTable[sum] = 1;
    }

    return false;
}

int main() {
    int arr[] = {4, 2, -3, 1, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    if (hasZeroSumSubarray(arr, n)) {
        printf("Subarray with zero sum exists\n");
    } else {
        printf("Subarray with zero sum does not exist\n");
    }

    return 0;
}
```

18) Find the equilibrium index of an array (where the sum of elements on the left is equal to sum on the right).

WTD: Examine the array to find an index where the sum of all elements to its left is equal to the sum of all elements to its right.
(e.g.: I/P: [-7, 1, 5, 2, -4, 3, 0]; O/P: 3)

```c
#include <stdio.h>

// Function to find the equilibrium index of an array
int findEquilibriumIndex(int arr[], int n) {
    int totalSum = 0;
    int leftSum = 0;

    // Calculate the total sum of the array
    for (int i = 0; i < n; i++) {
        totalSum += arr[i];
    }

    for (int i = 0; i < n; i++) {
        // Subtract the current element from the total sum to get the right sum
        totalSum -= arr[i];

        // If left sum equals right sum, it's an equilibrium index
        if (leftSum == totalSum) {
            return i;
        }

        // Otherwise, add the current element to the left sum
        leftSum += arr[i];
    }

    // No equilibrium index found
    return -1;
}

int main() {
    int arr[] = {-7, 1, 5, 2, -4, 3, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    int equilibriumIndex = findEquilibriumIndex(arr, n);

    if (equilibriumIndex != -1) {
        printf("Equilibrium Index: %d\n", equilibriumIndex);
    } else {
        printf("No Equilibrium Index Found\n");
    }

    return 0;
}
```

19)Find the longest consecutive subsequence in an array.

WTD: Examine the array to find the longest stretch of numbers that appear in increasing consecutive order.

(e.g.: I/P: [1, 9, 3, 10, 4, 20, 2]; O/P: [1, 2, 3, 4])

```
#include <stdio.h>
#include <stdbool.h>

// Function to find the longest consecutive subsequence in an array
void findLongestConsecutiveSubsequence(int arr[], int n) {
    int longestLength = 0;
    int currentLength = 0;
    int startOfSubsequence = 0;
    int endOfSubsequence = 0;

    // Create a hash set to store elements
    bool hashSet[10000] = {false}; // Assuming the range of elements won't
exceed 10000

    // Fill the hash set with elements from the array
    for (int i = 0; i < n; i++) {
        hashSet[arr[i]] = true;
    }

    // Iterate through the array to find the longest consecutive
subsequence
    for (int i = 0; i < n; i++) {
        int currentElement = arr[i];
        if (!hashSet[currentElement - 1]) {
            int j = currentElement;
            while (hashSet[j]) {
                j++;
            }

            currentLength = j - currentElement;

            if (currentLength > longestLength) {
                longestLength = currentLength;
                startOfSubsequence = currentElement;
                endOfSubsequence = j - 1;
            }
        }
    }

    // Print the longest consecutive subsequence
    printf("Longest Consecutive Subsequence: ");
    for (int i = startOfSubsequence; i <= endOfSubsequence; i++) {
        printf("%d ", i);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 9, 3, 10, 4, 20, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    findLongestConsecutiveSubsequence(arr, n);

    return 0;
}
```

20) Rearrange array such that arr[i] becomes arr[arr[i]].

WTD: Transform the array such that the number at each index corresponds to the number found at the index from the original array specified by the current number.

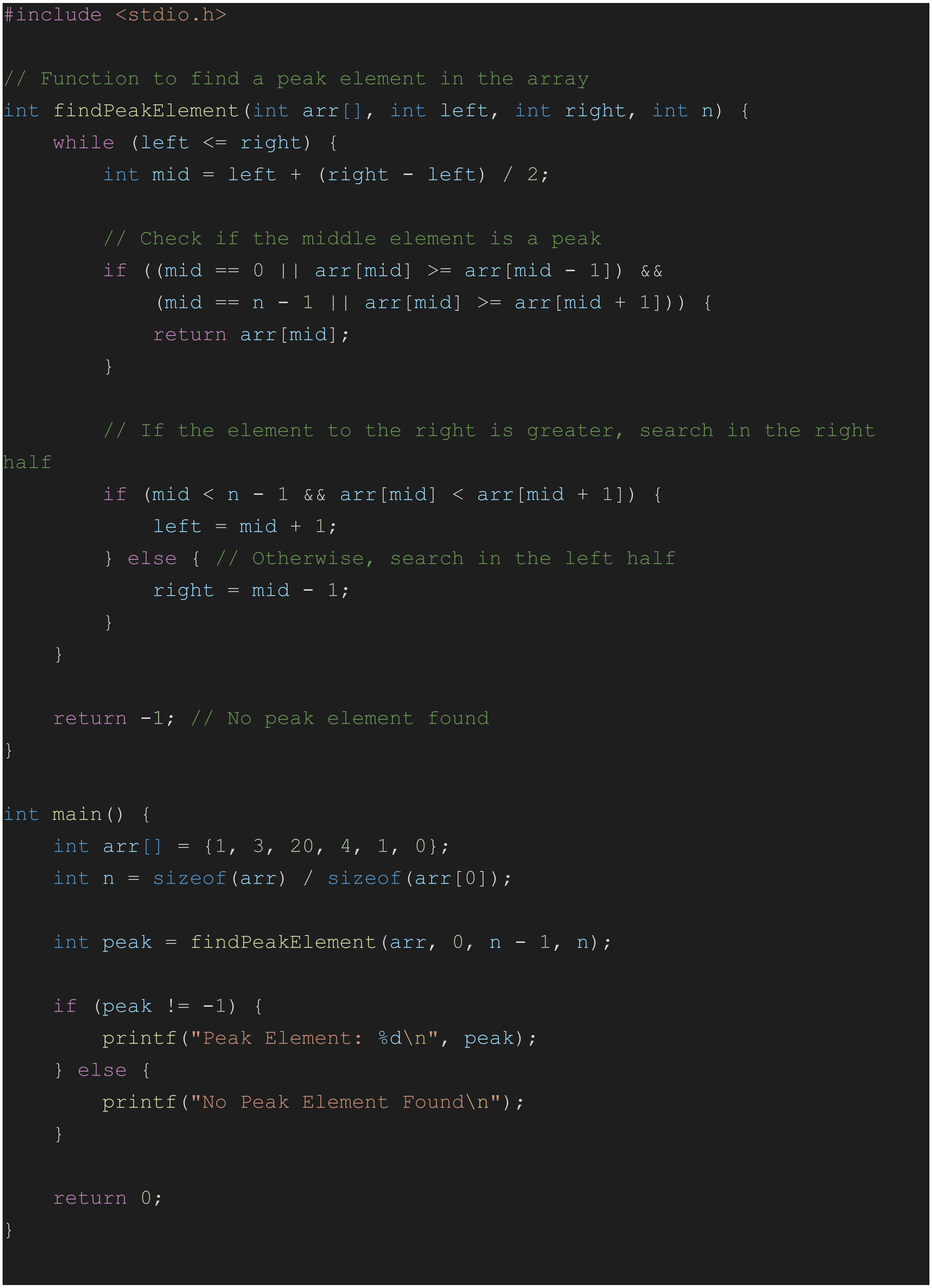
(e.g.: I/P: [0, 1, 2, 3]; O/P: [0, 1, 2, 3])

```c
#include <stdio.h>

// Function to rearrange the array as described
void rearrangeArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] += (arr[arr[i]] % n) * n;
    }

    for (int i = 0; i < n; i++) {
        arr[i] /= n;
    }
}

int main() {
    int arr[] = {0, 1, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    rearrangeArray(arr, n);

    printf("Rearranged Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

21) Find the peak element in an array (greater than or equal to its neighbors).

WTD: Scrutinize the array to find an element that is both larger than its predecessor and its successor.

(e.g.: I/P: [1, 3, 20, 4, 1, 0]; O/P: 20)



```
#include <stdio.h>

// Function to find a peak element in the array
int findPeakElement(int arr[], int left, int right, int n) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if the middle element is a peak
        if ((mid == 0 || arr[mid] >= arr[mid - 1]) &&
            (mid == n - 1 || arr[mid] >= arr[mid + 1])) {
            return arr[mid];
        }

        // If the element to the right is greater, search in the right
half
        if (mid < n - 1 && arr[mid] < arr[mid + 1]) {
            left = mid + 1;
        } else { // Otherwise, search in the left half
            right = mid - 1;
        }
    }

    return -1; // No peak element found
}

int main() {
    int arr[] = {1, 3, 20, 4, 1, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    int peak = findPeakElement(arr, 0, n - 1, n);

    if (peak != -1) {
        printf("Peak Element: %d\n", peak);
    } else {
        printf("No Peak Element Found\n");
    }

    return 0;
}
```

22) Compute the product of an array except self.

WTD: For every index in the array, calculate the product of all numbers except for the number at that index.

(e.g.: I/P: [1,2,3,4]; O/P: [24,12,8,6])

```
#include <stdio.h>

// Function to compute the product of an array except self
void productExceptSelf(int arr[], int n, int result[]) {
    int leftProduct[n];
    int rightProduct[n];

    // Initialize prefix and suffix product arrays
    leftProduct[0] = 1;
    rightProduct[n - 1] = 1;

    // Calculate prefix products
    for (int i = 1; i < n; i++) {
        leftProduct[i] = leftProduct[i - 1] * arr[i - 1];
    }

    // Calculate suffix products
    for (int i = n - 2; i >= 0; i--) {
        rightProduct[i] = rightProduct[i + 1] * arr[i + 1];
    }

    // Calculate the final result by multiplying prefix and suffix
products
    for (int i = 0; i < n; i++) {
        result[i] = leftProduct[i] * rightProduct[i];
    }
}

int main() {
    int arr[] = {1, 2, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    int result[n];
    productExceptSelf(arr, n, result);

    printf("Result: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");

    return 0;
}
```

23) Compute the leaders in an array.

WTD: Traverse the array from right to left, finding numbers that remain the largest compared to all numbers on their right.
(e.g.: I/P: [16,17,4,3,5,2]; O/P: [17,5,2])

```
#include <stdio.h>
#include <stdbool.h>

// Function to compute and print the leaders in an array
void findLeaders(int arr[], int n) {
    int leader = arr[n - 1];
    printf("Leaders: %d ", leader);

    for (int i = n - 2; i >= 0; i--) {
        if (arr[i] >= leader) {
            leader = arr[i];
            printf("%d ", leader);
        }
    }

    printf("\n");
}

int main() {
    int arr[] = {16, 17, 4, 3, 5, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    findLeaders(arr, n);

    return 0;
}
```

24) Find if an array can be divided into pairs whose sum is divisible by k.

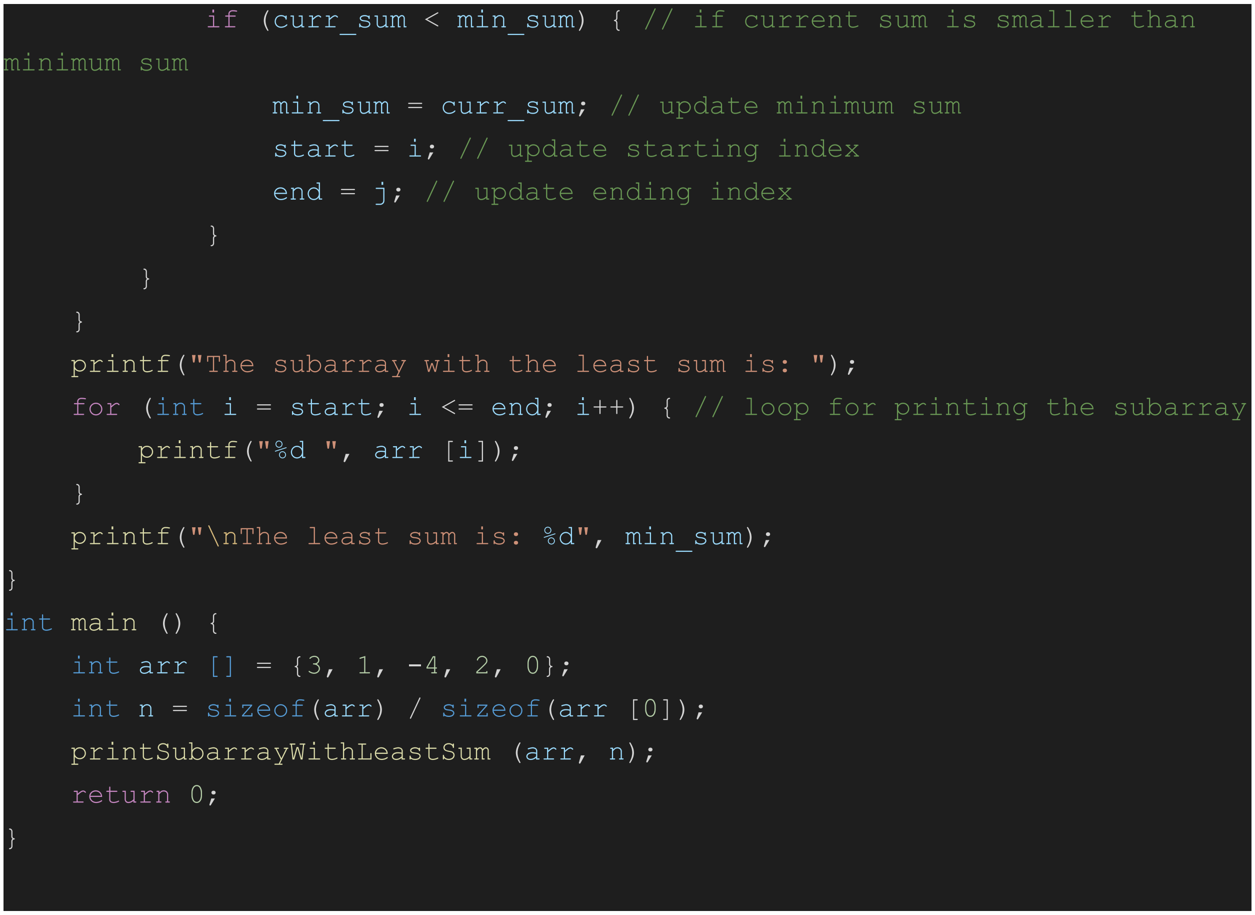
WTD: Examine the array to see if it can be segmented into pairs such that the sum of each pair's numbers is divisible by a specific number, k.
(e.g.: I/P: [9, 7, 5, -3], k=6; O/P: True)

```
#include <stdio.h>
#include <stdbool.h>

// Function to check if an array can be divided into pairs with sum
divisible by k
bool canDivideArray(int arr[], int n, int k) {
    // Create an array to store the count of remainders
    int remainders[k];
    for (int i = 0; i < k; i++) {
        remainders[i] = 0;
    }

    // Count the remainders of elements
    for (int i = 0; i < n; i++) {
        int remainder = (arr[i] % k + k) % k; // Ensure positive remainder
        remainders[remainder]++;
    }

    // Check if each remainder can form pairs
    if (remainders[0] % 2 != 0) {
        return false; // The remainder 0 must have an even count
    }

    for (int i = 1; i <= k / 2; i++) {
        if (remainders[i] != remainders[k - i]) {
            return false; // Remainders i and k-i must have equal counts
        }
    }

    return true;
}

int main() {
    int arr[] = {9, 7, 5, -3};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 6;

    bool canDivide = canDivideArray(arr, n, k);

    if (canDivide) {
        printf("Array can be divided into pairs with sum divisible by %d: True\n", k);
    } else {
        printf("Array cannot be divided into pairs with sum divisible by %d: False\n", k);
    }

    return 0;
}
```

25) Find the subarray with the least sum.

WTD: Investigate all possible subarrays of the given array, finding the one with the smallest sum. (e.g.: I/P: [3,1,-4,2,0]; O/P: -4)

```
#include <stdio.h>
void printSubarrayWithLeastSum (int arr [], int n) {
    int min_sum = arr [0]; // initialize minimum sum as first element
    int start = 0; // initialize starting index as 0
    int end = 0; // initialize ending index as 0
    for (int i = 0; i < n; i++) { // loop for each element of the array
        int curr_sum = 0; // initialize current sum as 0
        for (int j = i; j < n; j++) { // loop for each subarray starting from i
            curr_sum += arr [j]; // add current element to current sum
```



```c
            if (curr_sum < min_sum) { // if current sum is smaller than
minimum sum
                min_sum = curr_sum; // update minimum sum
                start = i; // update starting index
                end = j; // update ending index
            }
        }
    }
    printf("The subarray with the least sum is: ");
    for (int i = start; i <= end; i++) { // loop for printing the subarray
        printf("%d ", arr [i]);
    }
    printf("\nThe least sum is: %d", min_sum);
}
int main () {
    int arr [] = {3, 1, -4, 2, 0};
    int n = sizeof(arr) / sizeof(arr [0]);
    printSubarrayWithLeastSum (arr, n);
    return 0;
}
```