1) Search for an element in an unsorted array.

WTD: Create a function that takes an unsorted array and a target element as input. Your task is to find the index of the target element using linear search. You can accomplish this by looping through the array and comparing each element with the target.
Algorithm: Linear search sequentially checks each element until the desired element is found or the array ends.
(e.g.: I/P: [5, 2, 9, 1, 5, 6], 9 ; O/P: 2)

```c
#include <stdio.h>

// Function to perform linear search in an unsorted array
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index if the target is found
        }
    }
    return -1; // Return -1 if the target is not found
}

int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 9;

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

2) Search for an element in a sorted array.

WTD: Implement a function that performs binary search to find a target element in a sorted array. You'll need to start by comparing the target with the middle element and adjust your search range accordingly. Divide and conquer by halving the array.
Algorithm: Binary search divides the array into halves and compares the middle element to the target.
(e.g.: I/P: [1, 2, 3, 4, 5], 4 ; O/P: 3)

```c
#include <stdio.h>

// Function to perform binary search in a sorted array
int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

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

        if (arr[mid] == target) {
            return mid; // Return the index if the target is found
        }

        if (arr[mid] < target) {
            left = mid + 1; // Adjust the left boundary
        } else {
            right = mid - 1; // Adjust the right boundary
        }
    }

    return -1; // Return -1 if the target is not found
}

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

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

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

3) Implement binary search recursively.

WTD: Create a function that uses recursion to implement binary search on a sorted array. Your function should make a recursive call after halving the array based on the comparison with the target element.
Algorithm: Recursive binary search reduces the problem size by half in each recursive call.
(e.g.: I/P: [1, 2, 3, 4, 5], 3 ; O/P: 2)

```
#include <stdio.h>

// Function to perform binary search recursively
int binarySearchRecursive(int arr[], int left, int right, int target) {
    if (left > right) {
        return -1; // Target not found
    }

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

    if (arr[mid] == target) {
        return mid; // Target found at index mid
    }

    if (arr[mid] < target) {
        return binarySearchRecursive(arr, mid + 1, right, target); //
Search in the right half
    } else {
        return binarySearchRecursive(arr, left, mid - 1, target); //
Search in the left half
    }
}
```

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

    int result = binarySearchRecursive(arr, 0, size - 1, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

4) Implement interpolation search on a sorted array.

WTD: Implement a function that employs interpolation search on a sorted array to find a target element. Use a formula to guess the probable position of the target element before conducting the search.
Algorithm: Interpolation search tries to find the position of the target value by using a probing formula.
(e.g.: I/P: [1, 2, 3, 4, 5], 4 ; O/P: 3)

```
#include <stdio.h>

// Function to perform interpolation search
int interpolationSearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right && target >= arr[left] && target <= arr[right]) {
        if (left == right) {
            if (arr[left] == target) {
                return left;
            }
            return -1; // Target not found
        }
```

```c
        // Calculate the probable position using interpolation formula
        int pos = left + ((double)(right - left) / (arr[right] -
arr[left])) * (target - arr[left]);

        if (arr[pos] == target) {
            return pos; // Target found
        }

        if (arr[pos] < target) {
            left = pos + 1;
        } else {
            right = pos - 1;
        }
    }

    return -1; // Target not found
}

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

    int result = interpolationSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

5) Use exponential search to find an element in a sorted array.

WTD: Develop a function that uses exponential search to find a target element in a sorted array. First exponentially grow the range where the target is likely to reside, and then apply binary search within that range. Algorithm: Exponential search involves two steps: finding a range where the element is likely to be and applying binary search.
(e.g.: I/P: [1, 2, 3, 4, 5], 3 ; O/P: 2)

```c
#include <stdio.h>

// Function to perform binary search within a range
int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Target found
        }

        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Target not found
}

// Function to perform exponential search
int exponentialSearch(int arr[], int size, int target) {
    if (arr[0] == target) {
        return 0; // Target found at the first element
    }

    int i = 1;
    while (i < size && arr[i] <= target) {
        i *= 2;
    }
    // Apply binary search within the found range
    return binarySearch(arr, i / 2, (i < size) ? i : size - 1, target);
}

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

    int result = exponentialSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

6) Implement jump search.

WTD: Write a function to implement jump search on a sorted array. Use a fixed step size to skip ahead and then perform a linear search within the step.
Algorithm: Jump search skips ahead by fixed steps and performs a linear search backward.
(e.g.: I/P: [1, 3, 5, 7, 9], 7 ; O/P: 3)

```c
#include <stdio.h>
#include <math.h>

// Function to perform jump search
int jumpSearch(int arr[], int size, int target) {
    int step = sqrt(size); // Set the step size
    int prev = 0;

    // Jump ahead by step size
    while (arr[(int)fmin(step, size) - 1] < target) {
        prev = step;
        step += sqrt(size);
        if (prev >= size) {
            return -1; // Element not found
        }
    }

    // Perform linear search within the step
    for (int i = prev; i < fmin(step, size); i++) {
        if (arr[i] == target) {
            return i; // Target found
        }
    }

    return -1; // Target not found
}

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;

    int result = jumpSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

7) Implement Fibonacci search on a sorted array.

WTD: Create a function that uses Fibonacci search to find a target element in a sorted array. Use Fibonacci numbers to divide the array into unequal parts.
Algorithm: Fibonacci search divides the array into unequal parts based on Fibonacci numbers.
(e.g.: I/P: [1, 2, 3, 4, 5], 2 ; O/P: 1)

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

// Function to find the minimum of two integers
int min(int x, int y) {
    return (x <= y) ? x : y;
}

// Function to perform Fibonacci search
int fibonacciSearch(int arr[], int size, int target) {
    int fibM_minus_2 = 0; // (m-2)th Fibonacci number
    int fibM_minus_1 = 1; // (m-1)th Fibonacci number
    int fibM = fibM_minus_1 + fibM_minus_2; // mth Fibonacci number

    // Find the smallest Fibonacci number greater than or equal to size
    while (fibM < size) {
        fibM_minus_2 = fibM_minus_1;
        fibM_minus_1 = fibM;
        fibM = fibM_minus_1 + fibM_minus_2;
    }

    int offset = -1; // Offset to indicate comparison range

    while (fibM > 1) {
        // Check if fibM_minus_2 is a valid index
        int i = min(offset + fibM_minus_2, size - 1);

        // If the target is greater than the current element, move the
offset and update Fibonacci numbers
        if (arr[i] < target) {
            fibM = fibM_minus_1;
            fibM_minus_1 = fibM_minus_2;
            fibM_minus_2 = fibM - fibM_minus_1;
            offset = i;
        }
        // If the target is smaller than the current element, update
Fibonacci numbers only
        else if (arr[i] > target) {
            fibM = fibM_minus_2;
            fibM_minus_1 = fibM_minus_1 - fibM_minus_2;
            fibM_minus_2 = fibM - fibM_minus_1;
        }
        // If the target is found, return the index
        else {
            return i;
        }
    }

    // If the target is not found, return -1
    if (fibM_minus_1 && arr[offset + 1] == target) {
        return offset + 1;
    }

    return -1;
}

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

    int result = fibonacciSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

8) Implement sentinel search on an unsorted array.

WTD: Write a function that employs sentinel search on an unsorted array. Add a sentinel element to the array to avoid checking boundaries. Algorithm: Sentinel search uses a sentinel value to avoid the boundary check.
(e.g.: I/P: [5, 2, 9, 1, 5, 6], 1 ; O/P: 3)

```c
#include <stdio.h>

// Function to perform sentinel search
int sentinelSearch(int arr[], int size, int target) {
    // Add the target as a sentinel element at the end of the array
    arr[size] = target;
    int i = 0;

    // Search for the target element
    while (arr[i] != target) {
        i++;
    }

    // If the search reached the last (sentinel) element, it means the
target was not found
    if (i == size) {
        return -1;
    }

    return i;
}

int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 1;

    int result = sentinelSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

9) Implement ternary search on a sorted array.

WTD: Implement a function for performing ternary search on a sorted array. Divide the array into three equal parts and use two midpoints for comparison.
Algorithm: Ternary search divides the array into three parts and compares the key with the two midpoints.
(e.g.: I/P: [1, 2, 3, 4, 5], 1 ; O/P: 0)

```c
#include <stdio.h>

// Function to perform ternary search
int ternarySearch(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid1 = left + (right - left) / 3;
        int mid2 = left + 2 * (right - left) / 3;

        if (arr[mid1] == target) {
            return mid1;
        }
        if (arr[mid2] == target) {
            return mid2;
        }

        if (target < arr[mid1]) {
            return ternarySearch(arr, left, mid1 - 1, target);
        } else if (target > arr[mid2]) {
            return ternarySearch(arr, mid2 + 1, right, target);
        } else {
            return ternarySearch(arr, mid1 + 1, mid2 - 1, target);
        }
    }

    return -1; // Target not found
}

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

    int result = ternarySearch(arr, 0, size - 1, target);
    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

10) Search an element in a sorted and rotated array.

WTD: Develop a function that searches for an element in a sorted but rotated array. Use a modified binary search that accounts for the rotation. Algorithm: Modified binary search. (e.g.: I/P: [3, 4, 5, 1, 2], 1 ; O/P: 3)

```c
#include <stdio.h>

// Function to perform binary search on a sorted and rotated array
int searchInRotatedArray(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Element found
        }

        // Check if the left half is sorted
        if (arr[left] <= arr[mid]) {
            // Check if the target is in the left half
            if (arr[left] <= target && target <= arr[mid]) {
                return searchInRotatedArray(arr, left, mid - 1, target);
            }
            // If not, search in the right half
            return searchInRotatedArray(arr, mid + 1, right, target);
        }

        // If the left half is not sorted, then the right half must be
sorted
```

```c
        // Check if the target is in the right half
        if (arr[mid] <= target && target <= arr[right]) {
            return searchInRotatedArray(arr, mid + 1, right, target);
        }
        // If not, search in the left half
        return searchInRotatedArray(arr, left, mid - 1, target);
    }

    return -1; // Element not found
}

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

    int result = searchInRotatedArray(arr, 0, size - 1, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

11) Find a peak element in an array.

WTD: Write a function to find a peak element in a given array. Use binary search to find an element that is greater than its neighbors. Algorithm: Use binary search to find a peak element. (e.g.: I/P: [1, 3, 5, 4, 2] ; O/P: 5)

```c
#include <stdio.h>

// Function to find a peak element in an array using binary search
int findPeakElement(int arr[], int left, int right) {
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] > arr[mid + 1]) {
            // The peak element must be in the left half (including mid)
            right = mid;
        } else {
            // The peak element must be in the right half (excluding mid)
            left = mid + 1;
        }
    }
    return left;
}

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

    int peakIndex = findPeakElement(arr, 0, size - 1);
    int peakElement = arr[peakIndex];

    printf("Peak element: %d\n", peakElement);

    return 0;
}
```

12) Find the missing number in an array of 1 to N.

WTD: Create a function that finds the missing number in an array that contains numbers from 1 to N. You can use binary search or sum formula to find the missing number.
Algorithm: Use binary search or mathematical formula.
(e.g.: I/P: [1, 2, 4, 6, 5, 7, 8] ; O/P: 3)

```
#include <stdio.h>

// Function to find the missing number using binary search
int findMissingNumberBinary(int arr[], int size) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
```

```c
        // Check if the mid element is in its correct position
        if (arr[mid] == mid + 1) {
            left = mid + 1; // Move to the right half
        } else {
            right = mid - 1; // Move to the left half
        }
    }

    return left + 1;
}

// Function to find the missing number using the mathematical formula
int findMissingNumberFormula(int arr[], int size) {
    int totalSum = ((size + 1) * (size + 2)) / 2;
    int arraySum = 0;

    for (int i = 0; i < size; i++) {
        arraySum += arr[i];
    }

    return totalSum - arraySum;
}

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

    int missingNumberBinary = findMissingNumberBinary(arr, size);
    int missingNumberFormula = findMissingNumberFormula(arr, size);

    printf("Missing number (Binary Search): %d\n", missingNumberBinary);
    printf("Missing number (Formula): %d\n", missingNumberFormula);

    return 0;
}
```

13) Find the first and last occurrence of an element in a sorted array.

WTD: Implement a function that finds both the first and last occurrences of a given element in a sorted array. Use a modified binary search algorithm to find the initial and final positions.
Algorithm: Modified binary search.
(e.g.: I/P: [2, 2, 2, 2, 2], 2 ; O/P: First = 0, Last = 4)

```
#include <stdio.h>

// Function to find the first occurrence of an element in a sorted array
int findFirstOccurrence(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    int firstOccurrence = -1;

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

        if (arr[mid] == target) {
            firstOccurrence = mid;
            right = mid - 1; // Continue searching in the left half
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return firstOccurrence;
}

// Function to find the last occurrence of an element in a sorted array
int findLastOccurrence(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    int lastOccurrence = -1;

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

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

```c
            lastOccurrence = mid;
            left = mid + 1; // Continue searching in the right half
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return lastOccurrence;
}

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

    int first = findFirstOccurrence(arr, size, target);
    int last = findLastOccurrence(arr, size, target);

    if (first != -1 && last != -1) {
        printf("First occurrence = %d\n", first);
        printf("Last occurrence = %d\n", last);
    } else {
        printf("Element not found in the array.\n");
    }

    return 0;
}
```

14) Find a fixed point (value equal to index) in a given array.

WTD: Write a function to find a fixed point (an element equal to its index) in a sorted array. Use binary search to efficiently find a fixed point. Algorithm: Use binary search for a sorted array. (e.g.: I/P: [-10, -5, 1, 3, 7, 9, 12, 17] ; O/P: 3)

```c
#include <stdio.h>

// Function to find a fixed point in a sorted array
```

```c
int findFixedPoint(int arr[], int size) {
    int left = 0, right = size - 1;

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

        if (arr[mid] == mid) {
            return mid; // Found a fixed point
        } else if (arr[mid] < mid) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }

    return -1; // No fixed point found
}

int main() {
    int arr[] = {-10, -5, 1, 3, 7, 9, 12, 17};
    int size = sizeof(arr) / sizeof(arr[0]);

    int fixedPoint = findFixedPoint(arr, size);

    if (fixedPoint != -1) {
        printf("Fixed point is found at index %d\n", fixedPoint);
    } else {
        printf("No fixed point found in the array.\n");
    }

    return 0;
}
```

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

WTD: Develop a function that counts the occurrences of a given element in a sorted array. Use a modified binary search to find the first and last occurrences, then calculate the count.
Algorithm: Modified binary search to find the first and last occurrences.

(e.g.: I/P: [2, 2, 2, 2, 2], 2 ; O/P: 5)

```c
#include <stdio.h>

// Function to find the first occurrence of a target element
int findFirstOccurrence(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    int result = -1;

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

        if (arr[mid] == target) {
            result = mid; // Update the result and continue searching in
the left half
            right = mid - 1;
        } else if (arr[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }

    return result;
}

// Function to find the last occurrence of a target element
int findLastOccurrence(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    int result = -1;

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

        if (arr[mid] == target) {
            result = mid; // Update the result and continue searching in
the right half
            left = mid + 1;
        } else if (arr[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }

    return result;
}

// Function to count occurrences of a target element in a sorted array
int countOccurrences(int arr[], int size, int target) {
    int firstOccurrence = findFirstOccurrence(arr, size, target);
    int lastOccurrence = findLastOccurrence(arr, size, target);

    if (firstOccurrence != -1 && lastOccurrence != -1) {
        return lastOccurrence - firstOccurrence + 1;
    } else {
        return 0;
    }
}

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

    int occurrences = countOccurrences(arr, size, target);

    printf("Occurrences of %d in the array: %d\n", target, occurrences);

    return 0;
}
```

16) Find the majority element in an array, if it exists.

WTD: Create a function to find the majority element in an array if it exists. Utilize the Boyer-Moore Voting Algorithm or a binary search on a sorted array.
Algorithm: Use the Boyer-Moore Voting Algorithm or binary search on a sorted array.
(e.g.: I/P: [3, 3, 4, 2, 4, 4, 2, 4, 4]; O/P: 4)

```c
#include <stdio.h>

// Function to find the majority element using Boyer-Moore Voting
Algorithm
int findMajorityElement(int arr[], int size) {
    int candidate = -1;
    int count = 0;

    // Find a candidate for the majority element
    for (int i = 0; i < size; 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 < size; i++) {
        if (arr[i] == candidate) {
            count++;
        }
    }

    if (count > size / 2) {
        return candidate;
    } else {
        return -1; // No majority element
    }
}

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

    int majorityElement = findMajorityElement(arr, size);
    if (majorityElement != -1) {
        printf("Majority element: %d\n", majorityElement);
    } else {
        printf("No majority element found.\n");
    }

    return 0;
}
```

17) Search for an element in a bitonic array (an array that is first increasing then decreasing).

WTD: Implement a function to search for an element in a bitonic array (an array that first increases and then decreases). First find the peak element, then conduct a binary search on the increasing and decreasing parts. Algorithm: First find the peak element, then perform binary search on the increasing and decreasing parts.
(e.g.: I/P: [1, 3, 8, 12, 4, 2]; Search Element: 8; O/P: 2)

```c
#include <stdio.h>

// Function to find the peak element (maximum) in a bitonic array
int findPeak(int arr[], int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1]) {
            return mid;
        } else if (arr[mid] > arr[mid + 1]) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1; // Should not happen for a valid bitonic array
}

// Function to perform binary search on the increasing part of the array
int binarySearchIncreasing(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

// Function to perform binary search on the decreasing part of the array
int binarySearchDecreasing(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}

// Function to search for an element in a bitonic array
int searchBitonicArray(int arr[], int size, int target) {
    int peakIndex = findPeak(arr, 0, size - 1);

    if (arr[peakIndex] == target) {
        return peakIndex;
    }

    int leftResult = binarySearchIncreasing(arr, 0, peakIndex - 1,
target);
    int rightResult = binarySearchDecreasing(arr, peakIndex + 1, size - 1,
target);

    if (leftResult != -1) {
        return leftResult;
    } else if (rightResult != -1) {
        return rightResult;
    } else {
        return -1; // Element not found in the bitonic array
    }
}

int main() {
    int arr[] = {1, 3, 8, 12, 4, 2};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 8;

    int result = searchBitonicArray(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the bitonic array\n", target);
    }

    return 0;
}
```

18) Search for an element in a matrix where rows and columns are sorted.

WTD: Develop a function to search for an element in a matrix where the rows and columns are sorted. Start from the top-right or bottom-left corner and incrementally move toward the target. Algorithm: Start from the top-right or bottom-left corner and move towards the target.
(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]], Search Element = 5; O/P: (1, 1))

```
#include <stdio.h>

// Function to search for an element in a sorted matrix
int searchInSortedMatrix(int matrix[][3], int rows, int cols, int target,
int *rowIndex, int *colIndex) {
    int row = 0;            // Start from the top row
    int col = cols - 1;     // Start from the rightmost column

    while (row < rows && col >= 0) {
        if (matrix[row][col] == target) {
            *rowIndex = row;
            *colIndex = col;
            return 1;  // Element found
        } else if (matrix[row][col] > target) {
            col--;      // Move left in the current row
        } else {
            row++;      // Move down to the next row
        }
    }

    return 0;  // Element not found
}

int main() {
    int matrix[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int rows = 3;
    int cols = 3;
    int target = 5;
    int rowIndex, colIndex;

    if (searchInSortedMatrix(matrix, rows, cols, target, &rowIndex,
&colIndex)) {
        printf("Element %d found at position (%d, %d) in the matrix.\n",
target, rowIndex, colIndex);
    } else {
        printf("Element %d not found in the matrix.\n", target);
    }

    return 0;
}
```

19) Count the number of times a sorted array is rotated.

WTD: Write a function that counts the number of times a sorted array is rotated. Use binary search to identify the pivot element, which indicates the rotation count. Algorithm: Use binary search to find the pivot element.
(e.g.: I/P: [15, 18, 2, 3, 6, 12]; O/P: 2)

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

```c
int countRotations(int arr[], int size) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        // If the array is already sorted, no rotations have occurred
        if (arr[left] <= arr[right]) {
            return left;
        }

        int mid = left + (right - left) / 2;
        int next = (mid + 1) % size; // Index of the next element

        // Check if the mid element is the pivot
        if (arr[mid] <= arr[next] && arr[mid] <= arr[(mid + size - 1) %
size]) {
            return mid;
        }

        // Decide whether to search the left or right half
        if (arr[mid] <= arr[right]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return -1; // In case of an empty array or invalid input
}

int main() {
    int arr[] = {15, 18, 2, 3, 6, 12};
    int size = sizeof(arr) / sizeof(arr[0]);

    int rotations = countRotations(arr, size);

    if (rotations != -1) {
        printf("The array is rotated %d times.\n", rotations);
    } else {
        printf("Invalid input or empty array.\n");
    }

    return 0;
}
```

20) Find the Kth maximum and minimum element in an array.

WTD: Implement a function to find the Kth maximum and minimum element in an array. Employ QuickSelect, sorting, or a min-max heap method. Algorithm: Use QuickSelect, sorting, or min-max heap methods.
(e.g.: I/P: [7, 10, 4, 3, 20, 15], K = 3; O/P: Max = 15, Min = 7)

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to perform partition (used in QuickSort)
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// QuickSort algorithm
void quickSort(int arr[], int low, int high) {
    if (low < high) {
```

```c
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to find the Kth maximum and minimum elements
void findKthMinMax(int arr[], int size, int k) {
    if (k > size) {
        printf("Invalid value of K.\n");
        return;
    }

    // Sort the array in ascending order using QuickSort
    quickSort(arr, 0, size - 1);

    // Kth minimum element is at index k-1
    int minK = arr[k - 1];

    // Kth maximum element is at index size-k
    int maxK = arr[size - k];

    printf("Kth Minimum: %d\n", minK);
    printf("Kth Maximum: %d\n", maxK);
}

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

    findKthMinMax(arr, size, k);

    return 0;
}
```

21) In a sorted array, find two numbers that add up to a specific target sum "N".

WTD: Create a function that, in a sorted array, finds two numbers that add up to a specific target sum "N". Use a two-pointer technique or binary search for an efficient solution. Algorithm: Use two-pointer technique or binary search.
(e.g.: I/P: [1, 2, 3, 4, 5], N = 9; O/P: (4, 5))

```c
#include <stdio.h>

void findTwoSum(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left < right) {
        int currentSum = arr[left] + arr[right];

        if (currentSum == target) {
            printf("I/P: [");
            for (int i = 0; i < size; i++) {
                printf("%d", arr[i]);
                if (i < size - 1) {
                    printf(", ");
                }
            }
            printf("], N = %d; O/P: (%d, %d)\n", target, arr[left],
arr[right]);
            return;
        } else if (currentSum < target) {
            left++;
        } else {
            right--;
        }
    }

    printf("No such pair found.\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 9;

    findTwoSum(arr, size, target);

    return 0;
}
```

22) In a sorted array of distinct elements, find the smallest missing element.

WTD: Write a function to find the smallest missing element in a sorted array of distinct elements. Use binary search to locate the smallest missing number efficiently. Algorithm: Use binary search to find the smallest missing number.
(e.g.: I/P: [0, 1, 2, 6, 9]; O/P: 3)

```
#include <stdio.h>

int findSmallestMissing(int arr[], int size) {
    int left = 0;
    int right = size - 1;

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

        // If the middle element is not equal to its index, the smallest
missing number is on the left.
        if (arr[mid] != mid) {
            right = mid - 1;
        } else {
            // Otherwise, it's on the right.
            left = mid + 1;
        }
    }

    // The smallest missing number is found when left > right.
    return left;
}
```

```
int main() {
    int arr[] = {0, 1, 2, 6, 9};
    int size = sizeof(arr) / sizeof(arr[0]);

    int smallestMissing = findSmallestMissing(arr, size);

    printf("I/P: [");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
        }
    }
    printf("]; O/P: %d\n", smallestMissing);

    return 0;
}
```

23) Search for an element in a nearly sorted array (elements may be swapped).

WTD: Implement a function to search for an element in a nearly sorted array (where elements may be slightly out of place). Use a modified binary search algorithm. Algorithm: Modified binary search.
(e.g.: I/P: [2, 1, 3, 5, 4], 2; O/P: 0)

```
#include <stdio.h>

int modifiedBinarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

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

        if (arr[mid] == target) {
            return mid; // Element found at index mid.
        }

        if (mid > left && arr[mid - 1] == target) {
            return mid - 1; // Element found at index mid - 1.
        }

        if (mid < right && arr[mid + 1] == target) {
            return mid + 1; // Element found at index mid + 1.
        }

        if (arr[mid] > target) {
            right = mid - 2; // Adjust the search range to the left.
        } else {
            left = mid + 2; // Adjust the search range to the right.
        }
    }

    return -1; // Element not found.
}

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

    int index = modifiedBinarySearch(arr, size, target);

    printf("I/P: [");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
        }
    }
    printf("], %d; O/P: %d\n", target, index);

    return 0;
}
```

24) Given an array of n unique numbers except for one number which is repeating and one number which is missing, find them.

WTD: Develop a function that identifies one number which is repeating and one which is missing in an array of n unique numbers. Leverage XOR bitwise operation or sorting techniques. Algorithm: Use XOR bitwise operation or sorting.
(e.g.: I/P: [3, 1, 3]; O/P: Repeating = 3, Missing = 2)

```
#include <stdio.h>

void findRepeatingAndMissing(int arr[], int size, int *repeating, int
*missing) {
    int XOR = 0;

    // XOR all elements in the array and all numbers from 1 to n
    for (int i = 0; i < size; i++) {
        XOR ^= arr[i];
        XOR ^= (i + 1);
    }

    // Find the rightmost set bit in XOR
    int rightmostSetBit = XOR & -XOR;

    int num1 = 0, num2 = 0;

    // Divide the numbers into two groups based on the rightmost set bit
    for (int i = 0; i < size; i++) {
        if (arr[i] & rightmostSetBit) {
            num1 ^= arr[i];
        } else {
            num2 ^= arr[i];
        }

        int j = i + 1;
        if (j & rightmostSetBit) {
            num1 ^= j;
        } else {
            num2 ^= j;
        }
```

```c
    }

    // Check which number is missing by comparing with the array
    for (int i = 0; i < size; i++) {
        if (arr[i] == num1) {
            *repeating = num1;
            *missing = num2;
            return;
        }
        if (arr[i] == num2) {
            *repeating = num2;
            *missing = num1;
            return;
        }
    }
}

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

    int repeating, missing;
    findRepeatingAndMissing(arr, size, &repeating, &missing);

    printf("I/P: [");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
        }
    }
    printf("]; O/P: Repeating = %d, Missing = %d\n", repeating, missing);

    return 0;
}
```

25) Given a sorted array of unknown length, find if an element exists.

WTD: Write a function to find if an element exists in a sorted array of unknown length. Utilize exponential search to define a suitable range for binary search. Algorithm: Use exponential search to find the high boundary for binary search.
(e.g.: I/P: [1, 2, 3, 4, 5, ...], Element = 5; O/P: 4)

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

// Define the function to perform exponential search
bool exponentialSearch(int arr[], int size, int target) {
    if (arr[0] == target) {
        return true; // Element found at the first position
    }

    int bound = 1;

    // Double the bound until it exceeds the array size or the element is
found
    while (bound < size && arr[bound] < target) {
        bound *= 2;
    }

    // Perform binary search within the identified range
    int left = bound / 2;
    int right = (bound < size) ? bound : (size - 1);

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

        if (arr[mid] == target) {
            return true; // Element found
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false; // Element not found
}

int main() {
    // Example sorted array (in this case, just an ascending sequence)
    int size = 100;
    int arr[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
    }

    int target = 5; // Element to search for

    if (exponentialSearch(arr, size, target)) {
        printf("I/P: [1, 2, 3, 4, 5, ....]; Element = %d; O/P: Found\n", target);
    } else {
        printf("I/P: [1, 2, 3, 4, 5, ....]; Element = %d; O/P: Not Found\n", target);
    }

    return 0;
}
```

—--Happy Learning —---