This document contains a basic understanding of logics to be applied in the programs given in Lab 01 (just for help).

Task1:

Explanation:

- **2D Array**: The array2D is a 3x3 array initialized with some integer values.
- **Column-Major Order**: In column-major order, we traverse the 2D array by columns first, then by rows.
- **Copying to 1D Array**: We use a nested loop where the outer loop iterates over columns, and the inner loop iterates over rows. The elements are then stored sequentially in the 1D array.
- **Output**: The program outputs the elements of the 1D array in the order they were copied.

```
// Copying elements in column-major order
for (int col = 0; col < cols; col++) {
    for (int row = 0; row < rows; row++) {
        array1D[index] = array2D[row][col];
        index++;
    }
}
```

```
// Output the 1D array
cout << "1D Array in Column Major Order: ";
for (int i = 0; i < rows * cols; i++) {
    cout << array1D[i] << " ";
}
```

Task 2:

Explanation:

- **Data Structure**:
- We have an array of student names and a 2D array to store their grades for each course. Missing grades are represented by -1.
- **GPA Calculation**:
- The calculateGPA function calculates the GPA by summing the grade points for all subjects (only valid grades) and dividing by the total number of credits for those subjects.
- **Input Data**:
- The 2D array grades stores the grades for each student. Missing grades are indicated by -1, which the program skips when calculating the GPA.
- **Output**:
- The program loops through each student, counts their valid grades, and calculates their GPA based on the provided grades.

```
// Function to calculate GPA for a single student
double calculateGPA(double grades[], int numberOfGrades, int creditHours) {
    double totalPoints = 0;
    for (int i = 0; i < numberOfGrades; i++) {
        totalPoints += grades[i] * creditHours;
    }
    return totalPoints / (numberOfGrades * creditHours);
}
    // Students' names
    string students[numStudents] = {"Ali", "Hiba", "Asma", "Zain", "Faisal"};

    // Grades for each student in each subject (assuming -1 for missing grades)
    double grades[numStudents][numCourses] = {
        {3.66, 3.33, 4.0, 3.0, 2.66}, // Ali
        {3.33, 3.0, 3.66, 3.0, -1},   // Hiba
        {4.0, 3.66, 2.66, -1, -1},    // Asma
        {2.66, 2.33, 4.0, -1, -1},    // Zain
```

```
    {3.33, 3.66, 4.0, 3.0, 3.33}  // Faisal
};

// Calculate and display GPA for each student
for (int i = 0; i < numStudents; i++) {
    // Count the number of valid grades for each student
    int validGradesCount = 0;
    for (int j = 0; j < numCourses; j++) {
        if (grades[i][j] != -1) {
            validGradesCount++;
        }
    }

    // Calculate GPA
    double gpa = calculateGPA(grades[i], validGradesCount, creditHours);

    // Display student's GPA
    cout << "GPA of " << students[i] << ": " << gpa <<
endl;
}
```

-

Task 3:

Explanation:

1. **Data Members:**
    - **int* nums;**: This is a pointer to an integer array that will store the numbers.
    - **int size;**: This keeps track of the current number of elements in the nums array.
    - **int capacity;**: This represents the current capacity of the nums array (the maximum number of elements it can hold before needing to be resized).
2. **Constructor:**
    - **MedianFinder()**:
        - Initializes size to 0, indicating the array is empty.
        - Sets capacity to 1, meaning the initial array can hold one element.
        - Allocates memory for the nums array with the initial capacity.
3. **Destructor:**
    - **~MedianFinder()**:
        - Frees the dynamically allocated memory for nums to prevent memory leaks.
4. **addNum(int num) Method:**
    - Adds a new number to the data structure while maintaining the array in sorted order.
    - **Resizing**:
        - If the array is full (size == capacity), it doubles the array's capacity.
        - It creates a new array with double the capacity and copies over the existing elements from the old array.
        - The old array is then deleted, and nums points to the new array.
    - **Insertion**:
        - The method then inserts the new number into the array in its correct sorted position by shifting larger elements to the right.
5. **findMedian() Method:**
    - Returns the median of the numbers in the array:
        - If the number of elements is even, the median is the average of the two middle elements.
        - If the number of elements is odd, the median is the middle element.

Example Usage in main():

- **Adding Numbers:**
    - medianFinder.addNum(1);
    - medianFinder.addNum(2);
    - After these two additions, the array looks like [1, 2].
- **Finding the Median:**
    - After adding 1 and 2, the median is calculated as (1 + 2) / 2.0 = 1.5.
    - Then, medianFinder.addNum(3); adds the number 3, making the array [1, 2, 3].
    - Now, the median is 2, since 2 is the middle element in the array.

```
class MedianFinder {
```

```
private:
    int* nums;
    int size;
    int capacity;

public:
    // Constructor
    MedianFinder()  {
        size=0;
        capacity=1;
        nums = new int[capacity];  // Initial capacity of 10
    }

    // Destructor to free the allocated memory
    ~MedianFinder() {
        delete[] nums;
    }

    // Adds a number to the data structure
    void addNum(int num) {
        // Resize the array if needed
        if (size == capacity) {
            // Creating new array of double size
        int* temp = new int[capacity * 2];

        capacity = capacity * 2;
        // copy element of old array in newly created array
        for (int i = 0; i < size; i++) {
            temp[i] = nums[i];
```

```
        }

        // Delete old array
        delete[] nums;

        // Assign newly created temp array to original array
        nums = temp;
        }

        // Insert the new number into the sorted position
        int i = size - 1;
        while (i >= 0 && nums[i] > num) {
            nums[i + 1] = nums[i];
            i--;
        }
        nums[i + 1] = num;
        size++;
    }

    // Returns the median of all elements so far
    double findMedian() {
        if (size % 2 == 0) {
            return (nums[size / 2 - 1] + nums[size / 2]) / 2.0;
        } else {
            return nums[size / 2];
        }
    }
};
```

// Example usage: in main function, call addNum() function as many times you want to add num then call findMedian() to find outthe median of numbers you added: print the output

```
   MedianFinder medianFinder;
   medianFinder.addNum(1);
   medianFinder.addNum(2);
cout << medianFinder.findMedian() << endl;  // Output: 1.5
   medianFinder.addNum(3);
```

Task 4:

Explanation:

- **Binary Search Algorithm**:
- The binary search algorithm works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
- **Runtime Complexity**:
- The binary search algorithm has a time complexity of O(log n), making it efficient for large datasets.
- **Code Flow**:
- **Initialization**: The search starts with the entire array (left = 0, right = nums.size() - 1).
- **Midpoint Calculation**: mid is calculated using the formula left + (right - left) / 2 to avoid overflow.
- **Comparison**: The middle element is compared with the target.
    - If nums[mid] equals the target, return mid.
    - If nums[mid] is less than the target, search in the right half (left = mid + 1).
    - If nums[mid] is greater than the target, search in the left half (right = mid - 1).
- **Loop**: This process continues until left exceeds right, indicating that the target is not in the array, and -1 is returned.
- **Examples**:
- **Example 1**: For nums = {-1,0,3,5,9,12} and target = 9, the output is 4 because 9 is at index 4.

Example 2: For nums = {-1,0,3,5,9,12} and target = 2, the output is -1 because 2 is not in the array.

```
int binarySearch(const int nums[], int size, int target) {
   int left = 0;
   int right = size - 1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      // Check if the target is at mid
      if (nums[mid] == target) {
         return mid;
      }
```

```
        // If the target is greater, ignore the left half
        if (nums[mid] < target) {
            left = mid + 1;
        }
        // If the target is smaller, ignore the right half
        else {
            right = mid - 1;
        }
    }
    // Target is not present in the array
    return -1;
}
int main() {
    int nums[] = {-1, 0, 3, 5, 9, 12};
    int size = sizeof(nums) / sizeof(nums[0]);  // Calculate the size of the array
    int target1 = 9;
    int target2 = 2;
    int result1 = binarySearch(nums, size, target1);
    int result2 = binarySearch(nums, size, target2);
    cout << "Index of " << target1 << " is: " << result1 << endl;
    cout << "Index of " << target2 << " is: " << result2 << endl;

    return 0;
}
```

Task 5:

Explanation:

The dimensions of the matrix (3x4) are specified explicitly in the code.

- **Array Size Handling:**
  The number of rows (m) and columns (n) are passed as additional parameters to the searchMatrix function. This is necessary because arrays in C++ don't carry size information with them.
- **Function Signature:**
  The function signature of searchMatrix now includes the matrix dimensions: bool searchMatrix(int matrix[][4], int m, int n, int target).

  Matrix Indexing:

  - The matrix is treated as a flattened 1D array, and the mid_value is accessed using matrix[mid / n][mid % n]. Here, mid / n gives the row index, and mid % n gives the column index

- **Binary Search:**
- The binary search algorithm is applied to the virtual 1D array with indices ranging from 0 to m*n - 1.
- At each step, calculate the middle index, then convert it to the corresponding row and column in the matrix.
- Compare the value at this position with the target:
  If it's equal, return true.
  If it's less, move the left boundary to mid + 1.
  If it's greater, move the right boundary to mid - 1.
- If the loop completes without finding the target, return false.

Example:

- **Input**: matrix = [[1, 3, 5, 7], [10, 11, 16, 20], [23, 30, 34, 60]], target = 3
- **Output**: true
- **Explanation**: The target 3 is found at position (0, 1) in the matrix, so the output is true.

```
#include <iostream>

using namespace std;


bool searchMatrix(int matrix[][4], int m, int n, int target) {

   if (m == 0 || n == 0) return false;
```

```
    int left = 0;
    int right = m * n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int mid_value = matrix[mid / n][mid % n];

        if (mid_value == target) {
            return true;
        } else if (mid_value < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}

int main() {
    // Define a 3x4 matrix
    int matrix[3][4] = {{1, 3, 5, 7},
                        {10, 11, 16, 20},
                        {23, 30, 34, 60}};

    int target = 3;
    bool result = searchMatrix(matrix, 3, 4, target);
    cout << (result ? "true" : "false") << endl;
```

```
    return 0;
}
```