**Question 1:**

Implement a program that utilizes the quicksort algorithm to efficiently sort an array of integers? Begin by prompting the user to input the number of elements and dynamically allocating memory for the array. Request the user to input the array elements and display the original array. Implement the quicksort function to sort the array in ascending order. Finally, display the sorted array. Consider incorporating error handling for user inputs and optimizing the code for larger datasets.

**Input Format:**

- Enter the number of elements in the array: [user_input_n]
- Enter the elements of the array separated by spaces: [element_1] [element_2] ... [element_n]

**Output Format:**

**Before Quick Sort:**

- [element_1] [element_2] ... [element_n]

**After Quick Sort:**

- [sorted_element_1] [sorted_element_2] ... [sorted_element_n]

**Title for Question 1:** Efficient Integer Sorting with QuickSort

**Solution:**

```
#include <iostream>
// Function to perform quick sort
void quick_Sort(int *nums, int low, int high) {
    // Initializing indices and pivot
    int i = low;
    int j = high;
    int pivot = nums[(i + j) / 2];
    int temp;
    // Partitioning step of the quicksort algorithm
    while (i <= j) {
        // Finding elements less than the pivot on the left side
        while (nums[i] < pivot)
            i++;
        // Finding elements greater than the pivot on the right side
        while (nums[j] > pivot)
            j--;
        // Swapping elements to maintain order
        if (i <= j) {
            temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
            i++;
            j--;
        }
    }
    // Recursive calls for the remaining partitions
    if (j > low)
        quick_Sort(nums, low, j);
    if (i < high)
        quick_Sort(nums, i, high);
}
// Main function
int main() {
    // Prompting the user for the number of elements
    int n;
    //std::cout << "Enter the number of elements in the array: ";
    std::cin >> n;
    // Dynamically allocating an array for sorting
    int *nums = new int[n];
    // Prompting the user to input array elements
  //  std::cout << "Enter the elements of the array separated by spaces:
    for (int i = 0; i < n; ++i)
        std::cin >> nums[i];
    // Displaying the original numbers in the array
    std::cout << "Before Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << nums[i] << " ";
    // Calling the quicksort function
    quick_Sort(nums, 0, n - 1);
    // Displaying the sorted numbers after Quick Sort
    std::cout << "\nAfter Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << nums[i] << " ";
    // Deallocating the dynamically allocated array
    delete[] nums;
    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 5 34 12 56 7 23 | Before Quick Sort: 34 12 56 7 23 After Quick Sort: 7 12 23 34 56 |
| 2 | 8 9 22 4 18 37 6 15 50 | Before Quick Sort: 9 22 4 18 37 6 15 50 After Quick Sort: 4 6 9 15 18 22 37 50 |
| 3 | 3 -5 0 5 | Before Quick Sort: -5 0 5 After Quick Sort: -5 0 5 |
| 4 | 6 99 32 45 18 21 76 | Before Quick Sort: 99 32 45 18 21 76 After Quick Sort: 18 21 32 45 76 99 |
| 5 | 4 8 16 4 2 | Before Quick Sort: 8 16 4 2 After Quick Sort: 2 4 8 16 |
| 6 | 7 7 14 21 28 35 42 49 | Before Quick Sort: 7 14 21 28 35 42 49 After Quick Sort: 7 14 21 28 35 42 49 |

**White List:**

**Black List:**

---

**Question 2:**

Implement a program that dynamically handles user input for array size, allows users to input array elements, and incorporates the quicksort algorithm for sorting in either ascending or descending order? Start by prompting users for the array size, dynamically allocating memory, and requesting array elements. Implement the quicksort algorithm with options for ascending or descending order. Consider implementing error handling for invalid inputs to ensure program robustness.

**Input Format:**

- Enter the number of elements in the array: [user_input_n]
- Enter the elements of the array separated by spaces: [element_1] [element_2] ... [element_n]
- Enter 'a' for ascending or 'd' for descending order: [user_input_order]

**Output Format:**

**Before Quick Sort:**

- [element_1] [element_2] ... [element_n]

**After Quick Sort:**

- [sorted_element_1] [sorted_element_2] ... [sorted_element_n]

**Title for Question 2:** Dynamic Sorting Mastery

**Solution:**

```
#include <iostream>
#include<limits>
#include <iostream>
// Function to perform quick sort
void quick_Sort(int *nums, int low, int high, bool ascending) {
    int i = low, j = high;
    int pivot = nums[(low + high) / 2];
    int temp;
    while (i <= j) {
        if (ascending) {
            while (nums[i] < pivot)
                i++;
            while (nums[j] > pivot)
                j--;
        } else {
            while (nums[i] > pivot)
                i++;
            while (nums[j] < pivot)
                j--;
        }
        if (i <= j) {
            temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
            i++;
            j--;
        }
    }
    if (j > low)
        quick_Sort(nums, low, j, ascending);
    if (i < high)
        quick_Sort(nums, i, high, ascending);
}
// Rest of the code remains the same
// Main function
int main() {
    int n;
    while (!(std::cin >> n) || n <= 0) {
        std::cout << "Invalid input. Please enter a valid positive intege
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'
    }
    int *nums = new int[n];
    for (int i = 0; i < n; ++i)
        std::cin >> nums[i];
    char sortOrder;
    std::cin >> sortOrder;
    bool ascending = (sortOrder == 'a');
    std::cout << "Before Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << nums[i] << " ";
    quick_Sort(nums, 0, n - 1, ascending);
    std::cout << "\nAfter Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << nums[i] << " ";
    delete[] nums;
    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 6 15 7 3 22 10 5 a | Before Quick Sort: 15 7 3 22 10 5 After Quick Sort: 3 5 7 10 15 22 |
| 2 | 8 40 12 8 25 16 6 35 18 d | Before Quick Sort: 40 12 8 25 16 6 35 18 After Quick Sort: 40 35 25 18 16 12 8 6 |
| 3 | 5 9 2 14 7 5 a | Before Quick Sort: 9 2 14 7 5 After Quick Sort: 2 5 7 9 14 |
| 4 | 5 10 10 10 10 10 d | Before Quick Sort: 10 10 10 10 10 After Quick Sort: 10 10 10 10 10 |
| 5 | 7 3 18 7 11 6 25 14 a | Before Quick Sort: 3 18 7 11 6 25 14 After Quick Sort: 3 6 7 11 14 18 25 |
| 6 | 3 23 15 22 a | Before Quick Sort: 23 15 22 After Quick Sort: 15 22 23 |

**White List:**

**Black List:**

---

**Question 3:**

Implement a program incorporating an optimized quick sort algorithm with insertion sort for small arrays. dynamically handling user input for array size and elements. Explore the recursive nature of the optimized quick sort and its adaptability to handle arrays of varying sizes. Consider the intricacies involved in programmatically implementing the sorting algorithm.

**Input Format:**

- Enter the number of elements in the array: [user_input_n]
- Enter the elements of the array separated by spaces: [element_1] [element_2] ... [element_n]

**Output Format:**

Before Optimized Quick Sort:

- [element_1] [element_2] ... [element_n]

After Optimized Quick Sort:

- [sorted_element_1] [sorted_element_2] ... [sorted_element_n]

**Title for Question 3:** Optimized Quicksort Implementation with Insertion Sort for Small Arrays

**Solution:**

```
#include <iostream>


const int INSERTION_THRESHOLD = 10;  // Threshold for using insertion sor


// Function to perform insertion sort
void insertionSort(int *arr, int low, int high) {
    for (int i = low + 1; i <= high; ++i) {
        int key = arr[i];
        int j = i - 1;


        while (j >= low && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }


        arr[j + 1] = key;
    }
}


// Partition function (example, replace with your own implementation)
int partition(int *arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
```

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


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


// Optimized quicksort with insertion sort for small arrays
void optimizedQuickSort(int *arr, int low, int high) {
    while (low < high) {
        // Use insertion sort for small arrays
        if (high - low + 1 <= INSERTION_THRESHOLD) {
            insertionSort(arr, low, high);
            break;
        }


        int pivotIndex = partition(arr, low, high);


        // Recursively sort the smaller partition
        if (pivotIndex - low < high - pivotIndex) {
            optimizedQuickSort(arr, low, pivotIndex - 1);
            low = pivotIndex + 1;
        } else {
            optimizedQuickSort(arr, pivotIndex + 1, high);
            high = pivotIndex - 1;
        }
    }
}


// Main function for testing
int main() {
    int n;


    std::cin >> n;


    int *arr = new int[n];
    for (int i = 0; i < n; ++i)
        std::cin >> arr[i];
    std::cout << "Before Optimized Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << arr[i] << " ";
    optimizedQuickSort(arr, 0, n - 1);
    std::cout << "\nAfter Optimized Quick Sort:" << std::endl;
    for (int i = 0; i < n; ++i)
        std::cout << arr[i] << " ";
    delete[] arr;
    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 5 10 5 8 2 7 | Before Optimized Quick Sort: 10 5 8 2 7 After Optimized Quick Sort: 2 5 7 8 10 |
| 2 | 7 15 12 10 8 6 4 2 | Before Optimized Quick Sort: 15 12 10 8 6 4 2 After Optimized Quick Sort: 2 4 6 8 10 12 15 |
| 3 | 10 20 5 17 8 12 10 14 3 7 15 | Before Optimized Quick Sort: 20 5 17 8 12 10 14 3 7 15 After Optimized Quick Sort: 3 5 7 8 10 12 14 15 17 20 |
| 4 | 8 5 8 5 2 8 2 5 2 | Before Optimized Quick Sort: 5 8 5 2 8 2 5 2 After Optimized Quick Sort: 2 2 2 5 5 5 8 8 |
| 5 | 6 1 2 3 4 5 6 | Before Optimized Quick Sort: 1 2 3 4 5 6 After Optimized Quick Sort: 1 2 3 4 5 6 |
| 6 | 9 10 10 10 10 10 10 10 10 10 | Before Optimized Quick Sort: 10 10 10 10 10 10 10 10 10 After Optimized Quick Sort: 10 10 10 10 10 10 10 10 10 |

**White List:**

**Black List:**

---

**Question 4:**

Implement a program that employs the simplified Merge Sort algorithm to efficiently sort an array. Begin by prompting the user to input the size of the array dynamically, followed by the array elements. Utilize vectors for dynamic array handling. The program should display the original array and the sorted array after applying the Merge Sort algorithm.

**Input Format:**

- Enter the number of elements in the array: [size]
- Enter the elements of the array separated by spaces: [element1] [element2] ... [elementN]

**Output Format:**

**Original numbers:**

- [element1] [element2] ... [elementN]

**Sorted numbers:**

- [sorted_element1] [sorted_element2] ... [sorted_elementN]

**Title for Question 4:** Dynamic Merge Sort Implementation

**Solution:**

```cpp
#include <iostream>
#include <vector>
using namespace std;
// Function to merge two subarrays into one sorted array
void merge(vector<int>& arr, int p, int q, int r) {
    vector<int> left(arr.begin() + p, arr.begin() + q + 1);
    vector<int> right(arr.begin() + q + 1, arr.begin() + r + 1);
    int i = 0, j = 0, k = p;
    while (i < left.size() && j < right.size()) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < left.size()) {
        arr[k++] = left[i++];
    }
    while (j < right.size()) {
        arr[k++] = right[j++];
    }
}
// Recursive function to perform Merge Sort
void mergeSort(vector<int>& arr, int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergeSort(arr, p, q);
        mergeSort(arr, q + 1, r);
        merge(arr, p, q, r);
    }
}
int main() {
    int len;
    // Get user input for array size
    //cout << "Enter the number of elements in the array: ";
    cin >> len;
    // Get user input for array elements
    vector<int> arr(len);
    //cout << "Enter the elements of the array separated by spaces: ";
    for (int i = 0; i < len; i++) {
        cin >> arr[i];
    }
    cout << "Original numbers:\n";
    // Displaying the original numbers in the array
    for (int i = 0; i < len; i++) {
        cout << arr[i] << " ";
    }
    // Sorting the array using Merge Sort
    mergeSort(arr, 0, len - 1);
    cout << "\nSorted numbers:\n";
    // Displaying the sorted numbers after Merge Sort
    for (int i = 0; i < len; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 8 7 2 5 1 8 4 6 3 | Original numbers: 7 2 5 1 8 4 6 3 Sorted numbers: 1 2 3 4 5 6 7 8 |
| 2 | 6 1 2 3 4 5 6 | Original numbers: 1 2 3 4 5 6 Sorted numbers: 1 2 3 4 5 6 |
| 3 | 10 15 10 8 12 5 18 3 7 14 20 | Original numbers: 15 10 8 12 5 18 3 7 14 20 Sorted numbers: 3 5 7 8 10 12 14 15 18 20 |
| 4 | 5 10 10 10 10 10 | Original numbers: 10 10 10 10 10 Sorted numbers: 10 10 10 10 10 |
| 5 | 7 9 8 7 6 5 4 3 | Original numbers: 9 8 7 6 5 4 3 Sorted numbers: 3 4 5 6 7 8 9 |
| 6 | 15 27 10 18 22 15 30 12 8 25 14 20 5 35 17 29 | Original numbers: 27 10 18 22 15 30 12 8 25 14 20 5 35 17 29 Sorted numbers: 5 8 10 12 14 15 17 18 20 22 25 27 29 30 35 |

**White List:**

**Black List:**

---

**Question 5:**

Implement a program that utilizes the Merge Sort algorithm to efficiently merge two sorted arrays, arr1 and arr2, into a single sorted array. Explain the significance of using Merge Sort in this context, detailing the algorithm's divide-and-conquer strategy and the merging process.

**Input Format:**

- The user is prompted to enter the size of array arr1.
- The user is prompted to enter the size of array arr2.
- The user is asked to enter the elements of array arr1 separated by spaces.
- The user is asked to enter the elements of array arr2 separated by spaces.

**Output Format:**

- The merged and sorted array is displayed.

**Title for Question 5:** Merge Sort for Merging Two Sorted Arrays

**Solution:**

```
#include <iostream>
using namespace std;

// Function to merge two sorted subarrays into one sorted array
void merge(int arr[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int* left = new int[n1];
    int* right = new int[n2];
    for (int i = 0; i < n1; i++)
        left[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        right[j] = arr[q + 1 + j];

    int i = 0, j = 0, k = p;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }

    delete[] left;
    delete[] right;
}

// Recursive function to perform Merge Sort
void mergeSort(int arr[], int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergeSort(arr, p, q);
        mergeSort(arr, q + 1, r);
        merge(arr, p, q, r);
    }
}

// Function to merge two sorted arrays
int* mergeArrays(int* arr1, int m, int* arr2, int n) {
    int* mergedArray = new int[m + n];
    int i = 0, j = 0, k = 0;

    while (i < m && j < n) {
        if (arr1[i] < arr2[j]) {
            mergedArray[k] = arr1[i];
            i++;
        } else {
            mergedArray[k] = arr2[j];
            j++;
        }
        k++;
    }

    while (i < m) {
        mergedArray[k] = arr1[i];
        i++;
        k++;
    }

    while (j < n) {
        mergedArray[k] = arr2[j];
        j++;
        k++;
    }

    return mergedArray;
}

int main() {
    int m, n;
    cin >> m >> n;

    int* arr1 = new int[m];
    int* arr2 = new int[n];

    for (int i = 0; i < m; i++) {
        cin >> arr1[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> arr2[i];
    }

    int* mergedArray = mergeArrays(arr1, m, arr2, n);

    cout << "Merged and Sorted Array: ";
    for (int i = 0; i < m + n; i++) {
        cout << mergedArray[i] << " ";
    }
    cout << endl;

    // Cleaning up dynamically allocated memory
    delete[] arr1;
    delete[] arr2;
    delete[] mergedArray;

    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 3 4 2 5 8 1 4 7 9 | Merged and Sorted Array: 1 2 4 5 7 8 9 |
| 2 | 5 3 10 15 20 25 30 5 12 18 | Merged and Sorted Array: 5 10 12 15 18 20 25 30 |
| 3 | 2 2 3 6 1 4 | Merged and Sorted Array: 1 3 4 6 |
| 4 | 4 5 8 12 16 20 5 10 15 25 30 | Merged and Sorted Array: 5 8 10 12 15 16 20 25 30 |
| 5 | 1 3 3 7 14 21 | Merged and Sorted Array: 3 7 14 21 |
| 6 | 4 5 8 12 16 20 5 10 15 25 30 | Merged and Sorted Array: 5 8 10 12 15 16 20 25 30 |

**White List:**

**Black List:**

---

**Question 6:**

Implements the code for the Merge Sort algorithm to sort an array of integers in descending order. However, it uses a Variable Length Array (VLA) to declare the array. Discuss the implications of using VLA in this context and suggest alternative approaches to handle dynamic array sizes within the program.

**Input Format:**

- The first line contains a single integer N, the size of the array.
- The second line contains N space-separated integers, the elements of the array.

**Output Format:**

- The first line should print Given array is: followed by the elements of the given array separated by space.
- After an empty line, the next line should print Sorted array in descending order is: followed by the elements of the sorted array separated by space.

**Title for Question 6:** Merge sort in decrease order

**Solution:**

```
#include <iostream>
using namespace std;

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] >= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

int main() {
    int arr_size;
 //   cout << "Enter the number of elements in the array: ";
    cin >> arr_size;

    int arr[arr_size]; // Note: Variable Length Array (VLA) is used here,

  //  cout << "Enter " << arr_size << " integers:" << endl;
    for (int i = 0; i < arr_size; i++) {
        cin >> arr[i];
    }

    cout << "Given array is:"<<endl;
    printArray(arr, arr_size);
    cout<<endl;

    mergeSort(arr, 0, arr_size - 1);

    cout << "Sorted array in descending order is:"<<endl;
    printArray(arr, arr_size);
    return 0;
}
```

**TestCases:**

| S.No | Inputs | Outputs |
|---|---|---|
| 1 | 6 80 55 65 45 70 35 | Given array is: 80 55 65 45 70 35 Sorted array in descending order is: 80 70 65 55 45 35 |
| 2 | 5 5 2 8 1 9 | Given array is: 5 2 8 1 9 Sorted array in descending order is: 9 8 5 2 1 |
| 3 | 6 15 6 23 10 3 8 | Given array is: 15 6 23 10 3 8 Sorted array in descending order is: 23 15 10 8 6 3 |
| 4 | 8 100 75 125 50 25 23 1 5 | Given array is: 100 75 125 50 25 23 1 5 Sorted array in descending order is: 125 100 75 50 25 23 5 1 |
| 5 | 6 9 21 7 12 15 1 | Given array is: 9 21 7 12 15 1 Sorted array in descending order is: 21 15 12 9 7 1 |
| 6 | 5 50 20 40 10 30 | Given array is: 50 20 40 10 30 Sorted array in descending order is: 50 40 30 20 10 |

**White List:**

**Black List:**