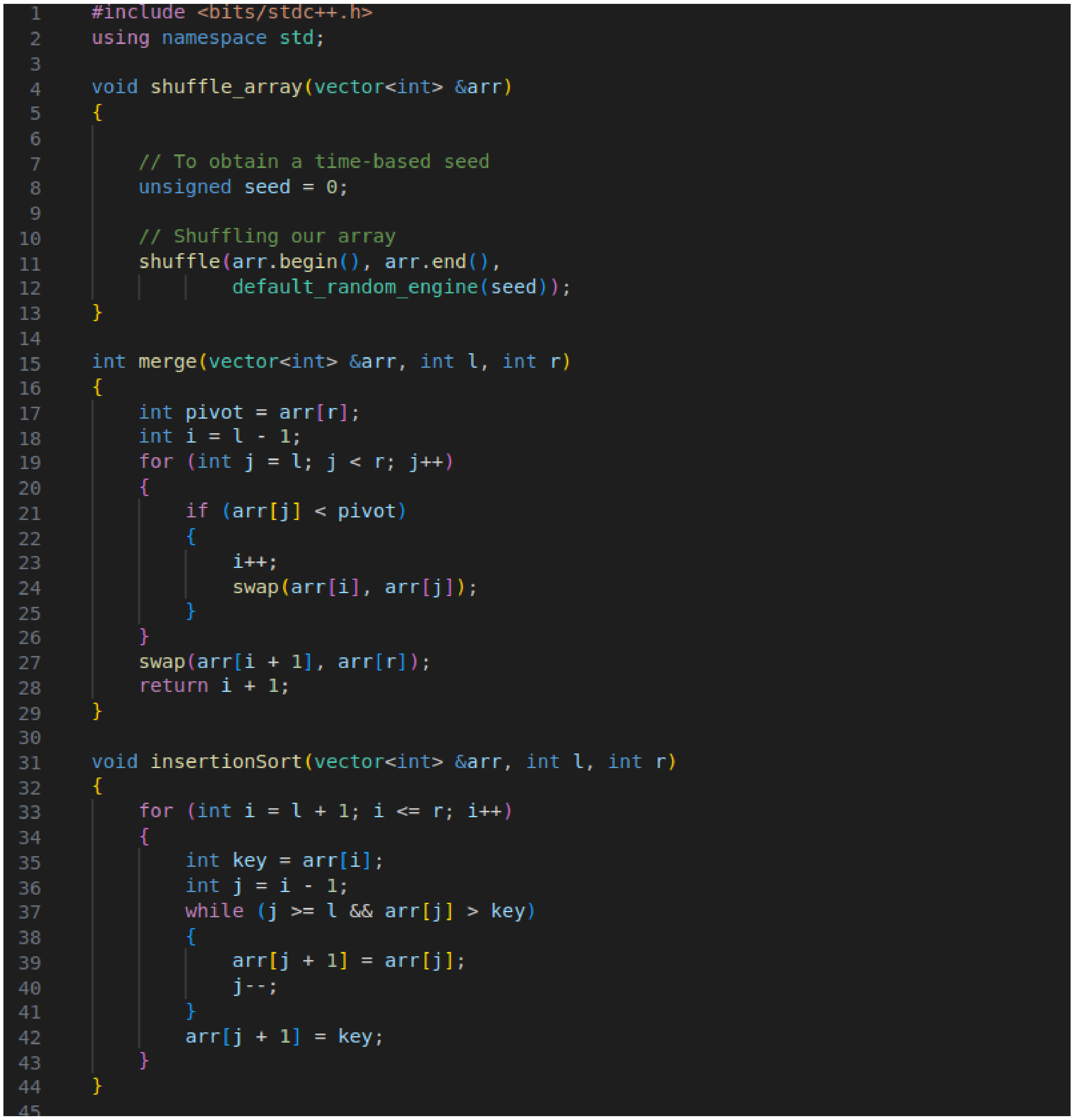
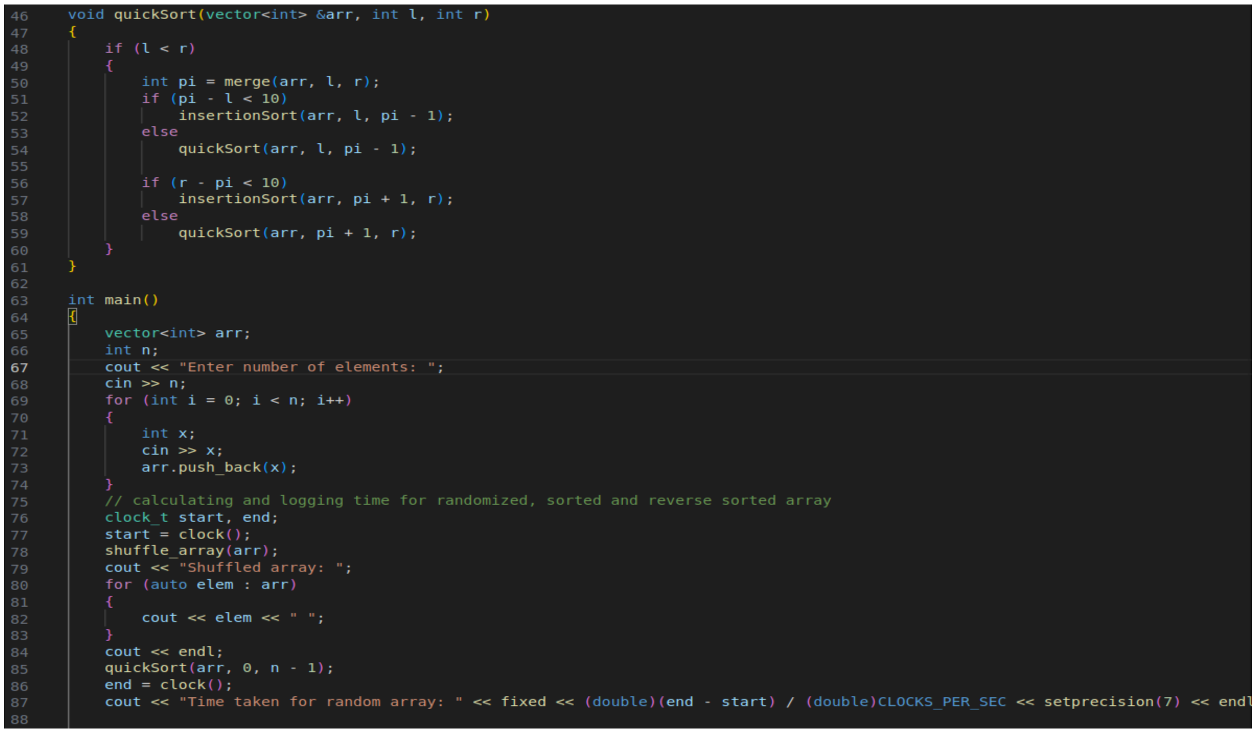
SE Assignment 2

Name: Vatsal Bhuva
Roll No.: IIT2022004
Section: A

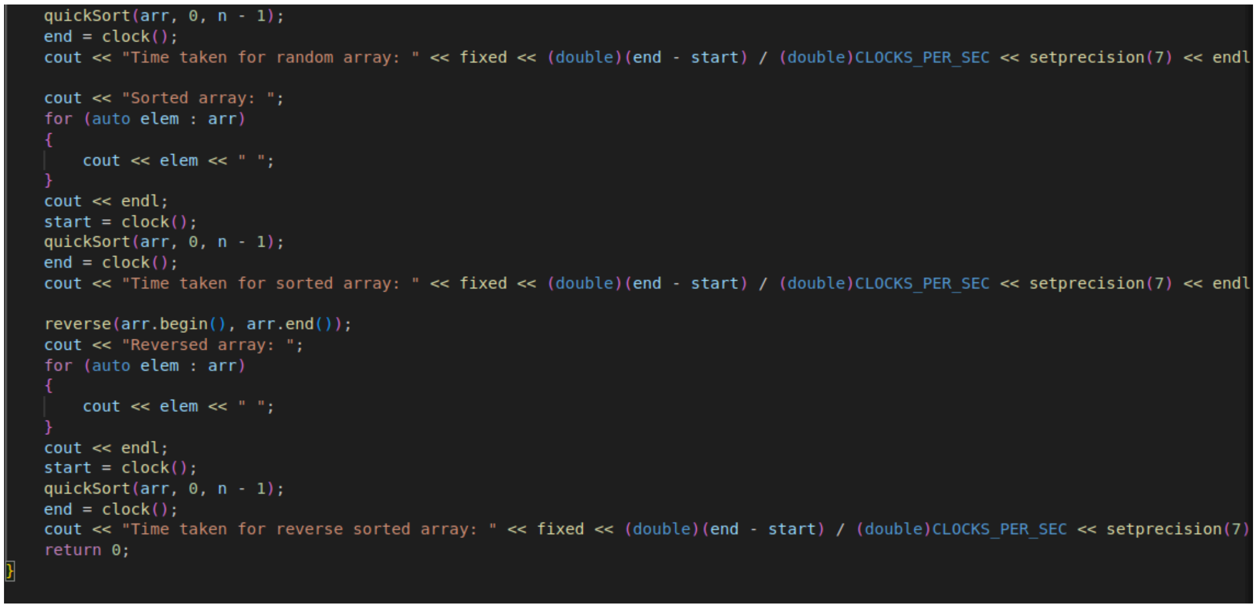
Code 1:



```cpp
#include <bits/stdc++.h>
using namespace std;

void shuffle_array(vector<int> &arr)
{

    // To obtain a time-based seed
    unsigned seed = 0;

    // Shuffling our array
    shuffle(arr.begin(), arr.end(),
            default_random_engine(seed));
}

int merge(vector<int> &arr, int l, int r)
{
    int pivot = arr[r];
    int i = l - 1;
    for (int j = l; j < r; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}

void insertionSort(vector<int> &arr, int l, int r)
{
    for (int i = l + 1; i <= r; i++)
    {
        int key = arr[i];
        int j = i - 1;
        while (j >= l && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

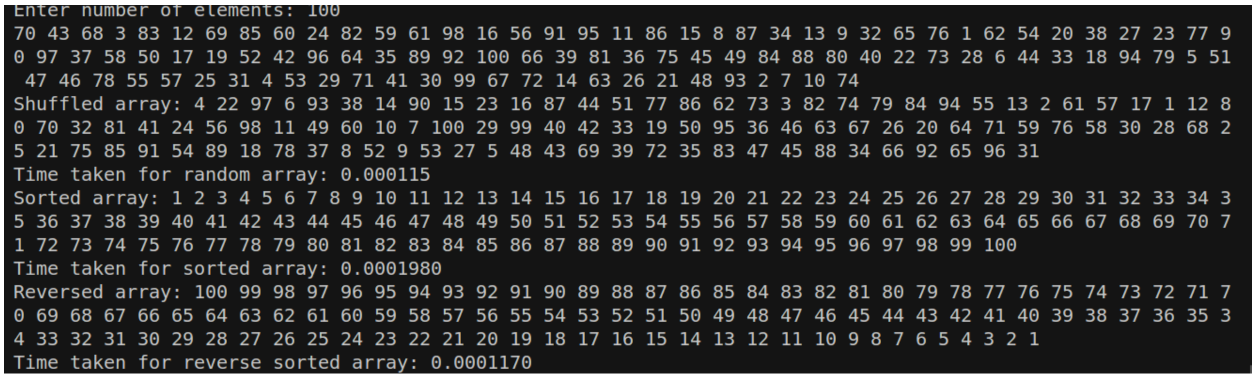


```cpp
void quickSort(vector<int> &arr, int l, int r)
{
    if (l < r)
    {
        int pi = merge(arr, l, r);
        if (pi - l < 10)
            insertionSort(arr, l, pi - 1);
        else
            quickSort(arr, l, pi - 1);

        if (r - pi < 10)
            insertionSort(arr, pi + 1, r);
        else
            quickSort(arr, pi + 1, r);
    }
}

int main()
{
    vector<int> arr;
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        arr.push_back(x);
    }
    // calculating and logging time for randomized, sorted and reverse sorted array
    clock_t start, end;
    start = clock();
    shuffle_array(arr);
    cout << "Shuffled array: ";
    for (auto elem : arr)
    {
        cout << elem << " ";
    }
    cout << endl;
    quickSort(arr, 0, n - 1);
    end = clock();
    cout << "Time taken for random array: " << fixed << (double)(end - start) / (double)CLOCKS_PER_SEC << setprecision(7) << endl
```

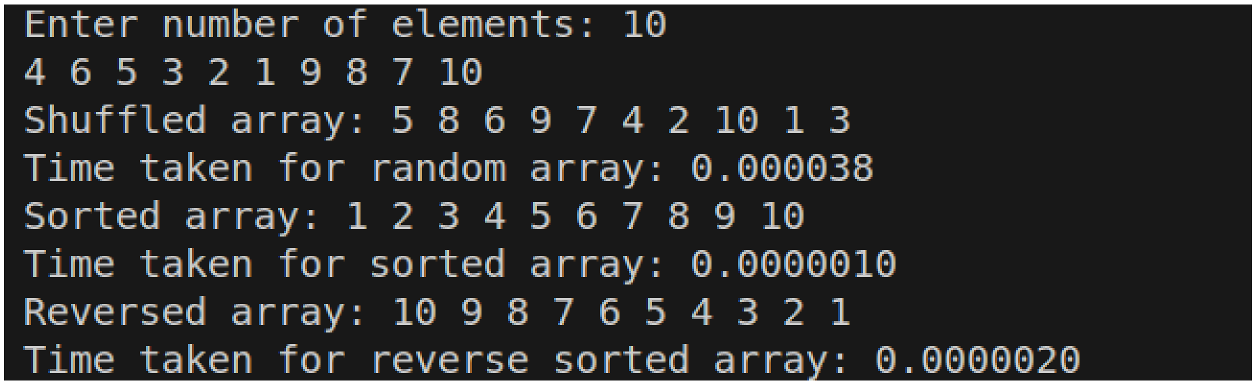


```cpp
    quickSort(arr, 0, n - 1);
    end = clock();
    cout << "Time taken for random array: " << fixed << (double)(end - start) / (double)CLOCKS_PER_SEC << setprecision(7) << endl

    cout << "Sorted array: ";
    for (auto elem : arr)
    {
        cout << elem << " ";
    }
    cout << endl;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    cout << "Time taken for sorted array: " << fixed << (double)(end - start) / (double)CLOCKS_PER_SEC << setprecision(7) << endl

    reverse(arr.begin(), arr.end());
    cout << "Reversed array: ";
    for (auto elem : arr)
    {
        cout << elem << " ";
    }
    cout << endl;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    cout << "Time taken for reverse sorted array: " << fixed << (double)(end - start) / (double)CLOCKS_PER_SEC << setprecision(7)
    return 0;
}
```

Outputs:



```
Enter number of elements: 100
70 43 68 3 83 12 69 85 60 24 82 59 61 98 16 56 91 95 11 86 15 8 87 34 13 9 32 65 76 1 62 54 20 38 27 23 77 9
0 97 37 58 50 17 19 52 42 96 64 35 89 92 100 66 39 81 36 75 45 49 84 88 80 40 22 73 28 6 44 33 18 94 79 5 51
 47 46 78 55 57 25 31 4 53 29 71 41 30 99 67 72 14 63 26 21 48 93 2 7 10 74
Shuffled array: 4 22 97 6 93 38 14 90 15 23 16 87 44 51 77 86 62 73 3 82 74 79 84 94 55 13 2 61 57 17 1 12 8
0 70 32 81 41 24 56 98 11 49 60 10 7 100 29 99 40 42 33 19 50 95 36 46 63 67 26 20 64 71 59 76 58 30 28 68 2
5 21 75 85 91 54 89 18 78 37 8 52 9 53 27 5 48 43 69 39 72 35 83 47 45 88 34 66 92 65 96 31
Time taken for random array: 0.000115
Sorted array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 3
5 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 7
1 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Time taken for sorted array: 0.0001980
Reversed array: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 7
0 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 3
4 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Time taken for reverse sorted array: 0.0001170
```


```
Enter number of elements: 10
4 6 5 3 2 1 9 8 7 10
Shuffled array: 5 8 6 9 7 4 2 10 1 3
Time taken for random array: 0.000038
Sorted array: 1 2 3 4 5 6 7 8 9 10
Time taken for sorted array: 0.0000010
Reversed array: 10 9 8 7 6 5 4 3 2 1
Time taken for reverse sorted array: 0.0000020
```

Analysis:
QuickSort generally has an average-case time complexity of O(n log n), while Insertion Sort has an average-case time complexity of O(n^2).

The hybrid algorithm's performance depends on the threshold value chosen. For small arrays, the insertion sort part can contribute to the overall efficiency, but for larger arrays, the quicksort part dominates.

The hybrid algorithm may perform better than traditional QuickSort and Insertion Sort in scenarios where the array contains partially sorted regions. Insertion Sort can efficiently handle nearly sorted parts, and QuickSort can efficiently handle the larger unsorted parts.

The hybrid algorithm's advantage lies in its adaptability to different scenarios, dynamically choosing between QuickSort and Insertion Sort based on the size of the subarrays.

Code 2:
(Files uploaded)

Analysis:
Variations observed in Bubble Sort:

It was observed that bubble sort to minimum time when input was nearly sorted and took the maximum time for when input was large and random. Bubble sort is an algorithm which sorts a given array in 0(n^2) time. It is based on comparisons between two adjacent elements in an array. It is also observed that time taken is relatively higher when the array is reverse sorted. This can be explained due to fact that maximum swaps take place when array is reversely sorted. Thus bubble sort is not an efficient algorithm when array is sorted in reverse order or when the input is very large.

Variations observed in Merge Sort:

It was observed that Merge sort to minimum time when input was nearly sorted and took the maximum time for when input was large and random. Merge sort is an algorithm which sorts a given array in O(nlogn) time. This sorting algorithm inculcates divide and conquer approch , where array is continuously divided into tow halves. Then both the halves are sorted and compared , after which botht the halves formed are merged. It is also observed that when array is randomly generated, time taken is maximum. This can be explained due to fact that when array is randomly organised , many divide and conquer steps are being performed. This increases the no of steps performed. Thus Merge sort is in general an efficient algorithm but fails when array is random or input is large.

Variations observed in Heap Sort:

It was observed that Heap sort to minimum time when input was nearly sorted and took the maximum time for when input was large and random. Heap sort is an algorithm which sorts a given array in 0(nlogn) time. It is a method in which we heapify . It is also observed that time taken is relatively higher when the array is reverse sorted. This can be explained due to fact that maxmimum swaps take place when array is reversely sorted. Thus Heap sort is not an efficient algorithm when array is sorted in reverse order or when the input is very large.