Problem 4.1

Nayan Man Singh Pradhan

a.
The c++ file : “4_1a.cpp” deals with problem 4.1a. Here, I implement a mergeSort along with an insertionSort. The user inputs the size of the array and then the value of k. The insertionSort is implemented when (rightPosition-leftPosition+1 <= k).

Screenshot below shows how and when insertionSort and mergeSort is called!

```
void mergeSort (int *arr, int left, int right, int k) {
    if ((right - left + 1) <= k) {
        //std::cout << "insertion sort" << std::endl;
        insertionSort(arr, left, right + 1);
    }
    else if ((right - left + 1) > k) {
        //std::cout << "merge sort" << std::endl;
        int mid;
        if (left < right) {
            mid = left + ((right - left)/2);
            mergeSort (arr, left, mid, k);
            mergeSort (arr, mid + 1, right, k);
            merge (arr, left, right, mid);
        }
    }
}
```

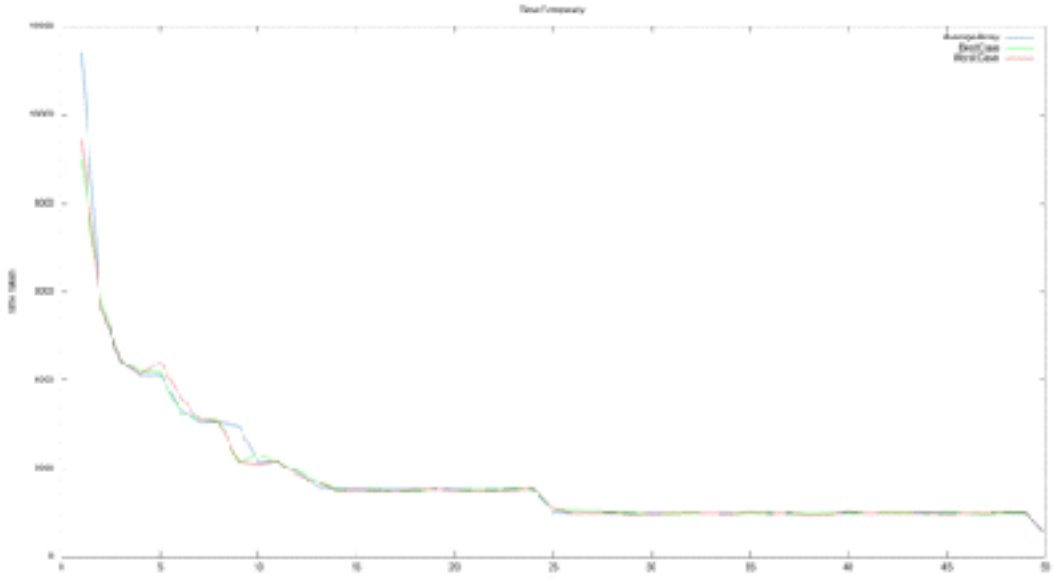
Screenshot below shows how the bestCase, worstCase, and average case array were implemented.

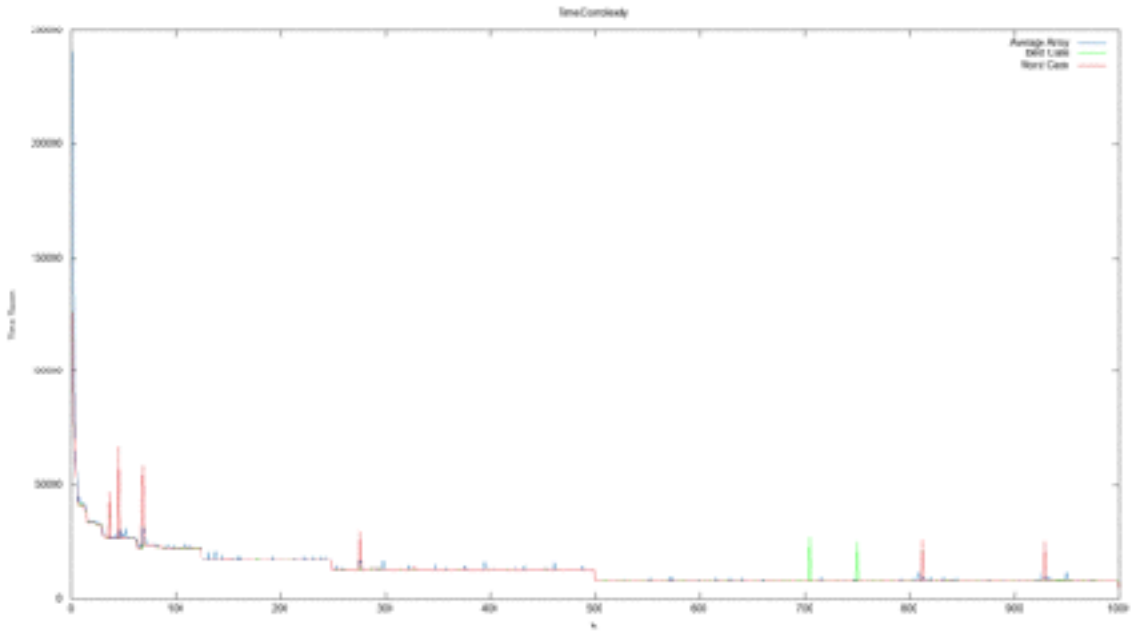
```
int bestCase[size];
for (int i = 0; i < size; i++) {
    bestCase[i] = 1;
}

int worstCase[size];
for (int i = 0; i < size; i++) {
    worstCase[size-1-i] = bestCase[i];
}

for (k = 1; k <= size; k++) {
    output << k << " ";
    long timeSum = 0;
    for(int count = 0; count < avgRepeat; count++) {
        auto startArr = std::chrono::high_resolution_clock::now();
        mergeSort (arr[count], 0, size-1, k);
        auto stopArr = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast
            <std::chrono::nanoseconds>(stopArr-startArr);
        timeSum += duration.count();
    }
    output << timeSum/avgRepeat << " ";
```

b.
The c++ file “4_1b” is similar to c++ file “4_1a”, but instead of printing the sequence in terminal, I am printing the time taken to execute average case (arr), best case (bestCase) and worst case(worstCase) in a different output file “data.txt”. Here, the user enters the value of n (number of elements in array), and k is each value from 1 to k.
The average case for this algorithm is computed by using a random number generator avgRepeat times (in our case avgRepeat = 50), computing the time, and computing the average of the times.
The best case for this algorithm is when the array is already sorted (in ascending order).
The worst case for this algorithm is when the array is in decreasing order. This is because, the worst case of insertionSort is when the array is in decreasing order and the worst case for the mergeSort is the same (as in both best and worst cases n*log(n) computations needs to be done). The execution time for the average, best and worst cases have been plotted (file plot.plt) using gnuplot. I have attached a two screenshots with n = 50, and n = 1000 for reference.





c.
We can clearly see that different values of k effect the time complexity for the best, average, and worst case. We notice that the highest time taken is when k is 1. This is because, when k = 1, everything is done through mergeSort and mergeSort takes longer to compute as it has to divide the array in smaller terms until the last array has only one element. Similarly, we also notice that the minimum time taken is when k = n (ie, the max value of k). This is because when k is maximum, it only implements insertionSort and not mergeSort. Hence, the original array is not divided into various smaller terms until a certain k is reached, and time is saved as insertion sort can be implemented in the array directly. In cases between k = 1 to k = n, the original array is sub-divided into halves until the array contains k elements, then insertion sort is implemented in the k elements. We can notice that the general trend of the graph between k = 1 to k = n is decreasing. Therefore, as k is getting larger (as more terms are sorted using insertion sort), the time taken is decreasing.

d.
Therefore, based on my graph, I would choose k to be the highest, ie k = n as it takes the least amount of time.