Problem 4.1 Merge Sort

(a) (4 points) Implement a variant of Merge Sort that does not divide the problem all the way down to subproblems of size 1. Instead, when reaching subsequences of length k it applies Insertion Sort on these n/k subsequences.

For full code: see my_algorithms.cpp

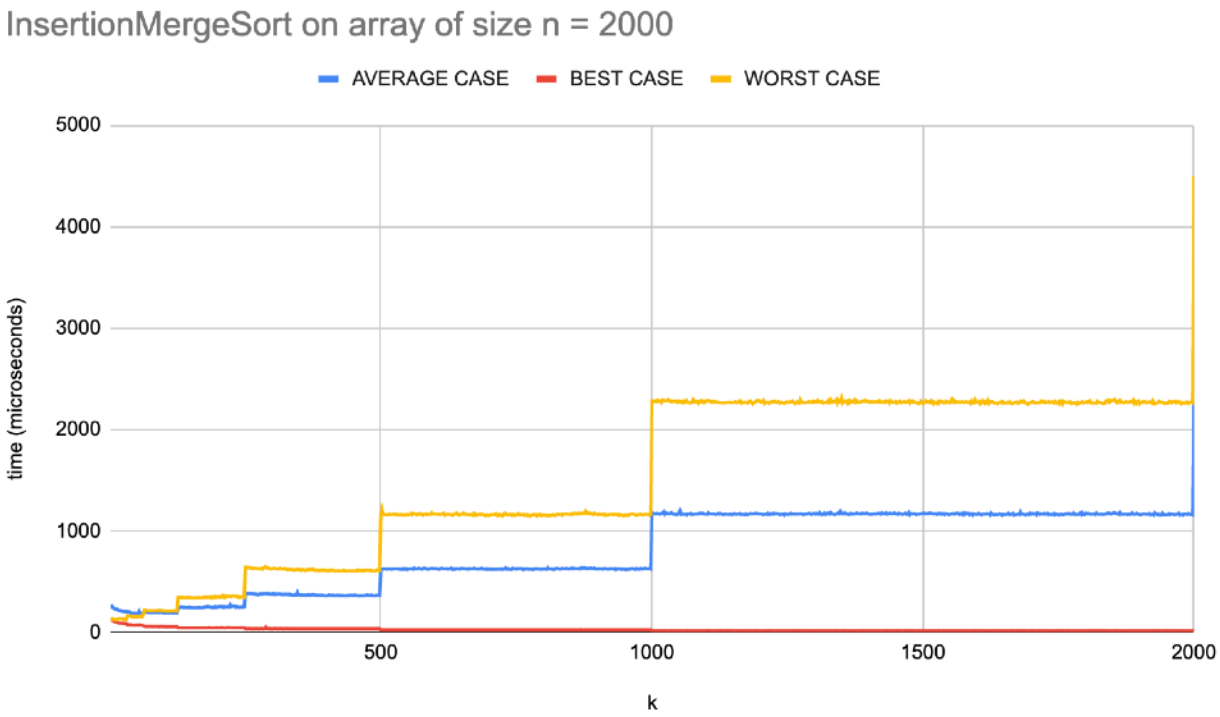
```
template <class T> void MyMergeSort(T* arr, int left, int right, int k){

        if (right - left > k-1){
                int middle = (left+right)/2;
                MyMergeSort<T>(arr, left, middle, k);
                MyMergeSort<T>(arr, middle + 1, right, k);


                merge<T>(arr, left, middle, right);

        }
        else {
                InsertionSort<T>(arr, left, right+1);
        }
}

template<class T> void merge(T* arr, int left, int middle, int right) {
        int i, j, k;
        int arr_left_size = middle - left + 1;
        int arr_right_size = right - middle;
        T arr_l[arr_left_size], arr_r[arr_right_size];
        for (i = 0; i < arr_left_size; i++)
                arr_l[i] = arr[left + i];
        for (j = 0; j < arr_right_size; j++)
                arr_r[j] = arr[middle + 1 + j];
        i = 0;
        j = 0;
        k = left;
        while (i < arr_left_size && j < arr_right_size) {
                if (arr_l[i] < arr_r[j]) {
                        arr[k] = arr_l[i];
                        i++;
                } else {
                        arr[k] = arr_r[j];
                        j++;
                }
                k++;
        }
        while (i < arr_left_size) {
                arr[k++] = arr_l[i++];

        }
        while (j < arr_right_size) {
                arr[k++] = arr_r[j++];
```

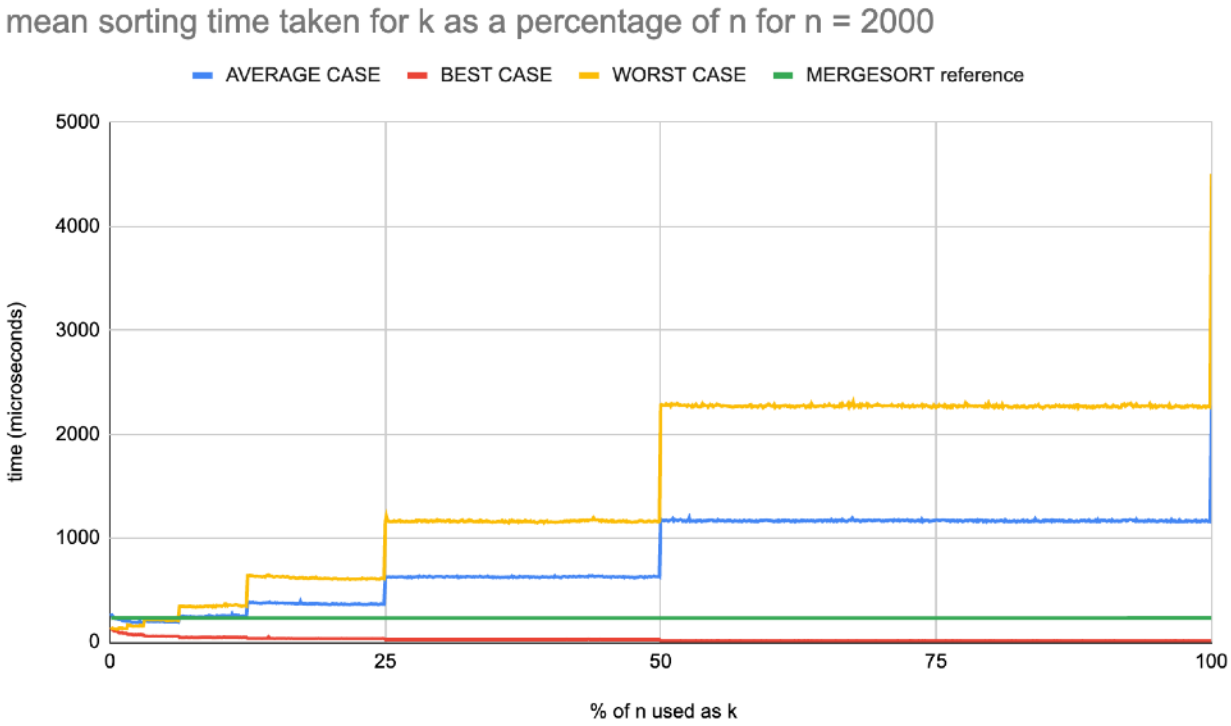
```
        }

}

template <class T> void InsertionSort(T* arr, int start, int end){
        int outerloop, cursor;
        T key;
        for (outerloop = start+1; outerloop < end; outerloop++)
        {
                key = arr[outerloop];
                cursor = outerloop - 1;

                // shift the elements ofthe sub-array if they are greater than key
                while (cursor >= start && arr[cursor] > key)
                {
                        arr[cursor + 1] = arr[cursor];
                        cursor = cursor - 1;
                }
                //insert element where it belongs in sub-array
                arr[cursor + 1] = key;
        }
}
```

(b) (3points) Apply it to the different sequences which satisfy best case, worst case and average case for different values of k. Plot the execution times for different values of k.

Best case: already sorted (increasingly ordered) array
Worst case: decreasingly-ordered array
Average case: random input array

To see the implementation, see testmymergesort.cpp
The following graph used increasing values of 2 for k from 2 to 2,000, with 100 repetitions using the mean computation time.





(c) (3 points) How do the different values of k change the best-, average-, and worst-case asymptotic time complexities for this variant? Explain/prove your answer.

Best case: as seen by the graph above, increasing the size of K decreases the running time. This is because insertion sort has a time complexity of $\Theta(n)$ for an array that is already sorted. Therefore, with a larger k, fewer recursive calls to merge sort are required for insertion sort to sort the sub-arrays, so the computation time is shorter as k increases. Using the best-case sorting time of insertion sort for a sub-array of length k, (which is k), applied to $\frac{n}{k}$ sub-arrays, we

get $k \times \frac{n}{k} = n = \Theta(n)$ time complexity. Then, the merge function is called $\log \frac{n}{k}$ times to merge the sorted sub-arrays, comparing n elements to sort the sub-arrays, resulting in a time complexity of $\theta(n + n \log \frac{n}{k})$, where this becomes $\Theta(n)$ if k = n, or normally just $\Theta(n \log \frac{n}{k})$.

worst case: because of the way the merge sort algorithm works, increasing the value of k only affects the time complexities when it crosses into a new sub-array size boundary, as seen by the second graph, the sorting time of the worst case spikes whenever *k* constitutes a $\frac{1}{2^i}$ sized partition, where $i \in \mathbb{N}^0$. This is because the sub-arrays created by each call of merge sort are half the size of the previous array, so there are several values of k which do not affect the time complexity of the algorithm. However, if increasing the value of k is sufficient to pass it onto the next-sized sub-array, the time complexity increases. This is because the worst case of insertion sort has time complexity $\Theta(n^2)$. In a more intuitive sense, each recursive call to standard merge sort that is avoided by this implementation of merge sort by specifying a sufficiently large *k* effectively doubles the computation time. In more mathematical terms: the worst-case time of insertion sort for a sub-array of length k is k^2, this is applied to $\frac{n}{k}$ sub- arrays, meaning that the worst-case time to sort these sub-lists is: $k^2 \times \frac{n}{k} = nk = \Theta(nk) = \Theta(n)$. Furthermore, the merge function will be called $\log(\frac{n}{k})$ times, resulting in a time complexity of:

$\Theta(nk + n \log \frac{n}{k}) = \Theta(n \log n)$

average case: this will work in much the same way as the worst case, as the average case time complexity of insertion sort on a k-sized sub-array is $\Theta(k^2)$, applied to $\frac{n}{k}$, sub- arrays, the insertion sort has a time complexity of $k^2 \times \frac{n}{k} = nk = \Theta(nk)$. Meanwhile, the merge function will be called $\log \frac{n}{k}$ times, with each merge call having a time complexity of n, resulting in a time complexity of: $\Theta(nk + n \log \frac{n}{k}) = \Theta(n \log \frac{n}{k})$

(d) Bonus(2points) Based on the results from (b) and (c),how would you choose k in practice? Briefly explain.

The full data values can be found inside the folder results

The best value of *k* to choose in practice would be the best sub-array size that would take the least time to sort compared to normal merge sort. From the raw data of the used programs, the value of k with the smallest computation time for the average case was k = 42, with 189.56 microseconds of total computation time. For reference, standard merge sort took 232.17 Microseconds to sort the array.

Problem 4.2 Recurrences

(a) (2points)T(n)=36T(n/6)+2n,

Using the master theorem:

$2n = O(n^{\log_6 36})$

$2n = O(n^{2-\epsilon})$

$2n = O(n^{2-1})$

then, $T(n) = \Theta(n^2)$

(b) (b) (2points)T(n)=5T(n/3)+17$n^{1.2}$

$17n^{1.2} = O(n^{\log_3 5})?$

$17n^{1.2} = O(n^{1.46497-\epsilon})?$, this holds for $\epsilon = 0.06497$, and can be shown using L'hopital's rule:

$$\lim_{n\to\infty} \frac{17n^{1.2}}{n^{1.4}} = \lim_{n\to\infty} \frac{17}{n^{0.2}} = 0$$

Therefore:

$T(n) = \Theta(n^{\log_3 5})$

(c) (2points)T(n)=12T(n/2)+$n^2 lg n$

Using the master theorem:

$n^{\log_2 12} = n^{3.585}$

$f(n) = O(n^{3.585-1})$

Reason: $n^{2.585}$ grows faster than $n^2 \log n$ as $n \to \infty$, easily shown by:

$\lim_{n\to\infty} \frac{\log n}{n^{0.585}}$, using L'Hopital's rule we get:

$$\lim_{n\to\infty} \frac{\log n}{n^{0.585}} \to \lim_{n\to\infty} \frac{n^{0.415}}{n} \to \frac{1}{n^{0.585}} = 0$$

Then,

$T(n) = \Theta(n^{3.585})$

(d) (2 points) T(n) = 3T(n/5) + T(n/2) + 2^n,

intuitively, we can see that 2^n is the largest element of the recursion tree, that it dominates the terms of $2^{n/5}$ and $2^{n/2}$ that are called recursively, so we can say that

$T(n) = \Theta(2^n)$

(e) Bonus(2points)T(n)=T(2n/5)+T(3n/5)+Θ(n).