Sheetal Parikh
Programming Assignment 2 - Summary

(a) Write pseudocode for median-of-three partitioning.

The median-of-three partitioning pseudocode will be very similar to normal quicksort. We will create a separate function that uses the first, last, and middle elements of the array and places them into another array that gets sorted. The 2nd element of the array of the 3 elements is set as the pivot and added as the leftmost element of the input array.

```
Function median3Pivot(array, left, right)
        first = array(left)
        last = arraylength - 1
        middle = (right + left) / 2

        sortingArray(array[left], array[middle], array[right])
        sort(sortingArray)

        median = sortingArray[1]

        swap median w/ array(left)              //swapping the median with the leftmost
                                                //element to serve as the pivot

Function Partition(array, left, right)
        pivot = array(left)
        i = left +1
        j = right

        while (j > i )
                while (i ≤ j && array[i] ≤  pivot          //searching from left to right
                        i++
                while (i ≤ j && array[j] > pivot           //searching backwards from right to left
                        j--
                if (j > i)
                        swap array[i] with array[j]        //swapping elements in list

        while ( j > left && array[j] ≥ pivot
                j--
```

```
        if (pivot > array [j])
                swap pivot with array[j]
                return j
        else
                return left

Function Quicksort(array, left, right)
        if (left ≥ right)
                return
        if (left < right)
                pivotindex = partition(array, left, right)
                Quicksort (array, left, pivotindex - 1)
                Quicksort (array, pivotindex - 1, right)
```

(b) What is the running time of median-of-three partitioning? Justify your answer.

I believe the running time of median-of-three partitioning quicksort would be very similar to normal quicksort for the average case. The primary difference between median-of-three partitioning and regular quicksort, is that we would find the median of the first, middle, and last number in the array. This median would then be substituted into the partitioning algorithm as the pivot and loop through the algorithm just like for regular quicksort. Therefore, the partitions would be more balanced using median-of-three quicksort compared to regular quicksort. However, in normal quicksort partitioning is still likely to occur in the same way at every level. Partitioning could be balanced as well as unbalanced. Therefore, the average balance of regular quicksort should become similar to median-of-three quicksort.

For median-of-three quicksort, for every partition you would be dividing the number of elements by 2, giving $nlogn$ levels and every level of the tree costs cn or $\Theta(n)$. Breaking up the process, we could say finding the median costs $O(n)$, replacing the median at the beginning of the array costs $O(1)$, calling the partition algorithm costs $O(n)$, and dividing the number of elements by 2 at every level gives us $2T(\frac{n}{2})$. Therefore, we have: $T(n) = 2T(\frac{n}{2}) + 2n$.

Using the master's theorem: $T(n) = 2T(\frac{n}{2}) + n$:

Step 1: a = 2

b = 2
f(n) = n

Step 2: $n^{log_b a} = n^{log_2 2} = 1$

Step 3: n = n
f(n) = n

Step 4: Case 2 = evenly distributed
Because f(n) = $\Theta(n)$, *then* $T(n) = \Theta(n^{log_b a} \, logn)$
$= \Theta \, nlogn$

Therefore, we can see that median-of-three quicksort has the same runtime as regular quicksort, $\Theta \, nlogn$.

(c) What is the running time of Quicksort if you use median-of-three partitioning on an input set that is already sorted? Justify your answer.

If we use median-of-three partitioning on an input that is already sorted, I believe Quicksort would run at a worst case time of $\Theta(nlogn)$. A sorted array causes very unbalanced partitioning for regular quicksort causing a worst-case runtime of $\Theta(n^2)$. For example, if a sorted array were all the same number, the partition for the first level would be 0 for one side and n-1 for the other and the cost for partitioning would be *cn.* At the second level, the partition would be 0 for one side and n-1 for the other and the cost for partitioning would be c(n-1). This pattern would continue giving us a runtime of $\Theta(n^2)$.

However, by finding the midpoint and substituting it as the pivot for every partition, the median-of-three algorithm would make it much less likely to cause unbalanced partitioning like in regular quicksort. Finding the midpoint, forces the algorithm to have balanced partitioning and therefore a runtimes that remains similar regardless of how the input array is sorted. Therefore, the median-of-three algorithm wouldn't have a more efficient runtime than $\Theta(nlogn)$, regardless of the input. Overall, I believe Quicksort would run at a worst case time of $\Theta(nlogn)$.

(d) Implement Quicksort using a normal pivot process and the median-of-three process described above. Test your run time analysis of median-of-three, and then compare the average and worse-case runtimes of Quicksort with the two pivot processes. Note that you must

implement all of these algorithms from scratch. Also remember that CPU time is not a valid measure for testing run time. You must use something such as the number of comparisons.

Based on the analysis above, I assumed that the number of comparisons and swaps would be similar for both the median-of-3 approach (both my version and the book version) and regular quicksort. Although, I chose to have the pivot at the first element of the array array, I do not think it would cause a difference in runtimes. However, because the median-of-three implementations have a more efficient runtime for a worst case scenario,such as a sorted array, I believe the median-of-three implementations would have less comparisons and swaps compared to that of regular quicksort.

I ran 4 tests in all 3 versions of quicksort using unsorted array of sizes: n= 50, 100, 200 and 10,000. I ran a worst case scenario using a sorted array of size: n = 50 and n = 100.

Overall, my results were both similar and different to what I predicted. The book implementations of regular and median-of-three quicksort produced similar results for the unsorted array, regardless of the size. I also expected the median-of-3 quicksort to have less comparisons and swaps than regular quicksort. However, the number of comparisons and swaps for both implementations were basically the same. Using larger test sizes of the unsorted array may have produced the results I expected.

My implementation of the median-of-three quicksort produced results that significantly differed to the book implementations. For tests of unsorted or sorted arrays and all values of n, the number of swaps and comparisons were much smaller than those of the book implementations. I believe this may have been caused by the counting of swaps and comparisons to be very sensitive to any change or difference in code. I tried moving the swap and comparison counters throughout different parts of the program however, it was producing the same result. I don't believe my implementation of median-of-three quicksort was able to handle the calculation of swaps and comparisons for sorted arrays as it produced a swap count of 1.

Summary of Results - swaps and comparisons:

<u>Book Regular Quicksort</u>

- Unsorted
    - n = 50: 189 swaps and 329 comparisons

- n=100: 431 swaps and 611 comparisons
- n=200: 1,112 swaps and 1,525 comparisons
- n=10,000: 91,901 swaps and 160,479 comparisons

- Sorted(ascending order)
  - n = 50: 1,224 swaps and 1,176 comparisons
  - n=100 : 4,949 swaps and 4,851 comparisons

Median of 3 - using Book Quicksort - pivot is the last element of array

- Unsorted
  - n = 50: 223 swaps and 312 comparisons
  - n=100: 509 swaps and 775 comparisons
  - n=200: 1,266 swaps and 1,913 comparisons
  - n=10,000: 85,156 swaps and 165,280 comparisons

- Sorted(ascending order)
  - n = 50: 1,299 swaps and 1,274 comparisons
  - n=100: 5,099 swaps and 5,049 comparisons

Median of 3 - pivot is the first element of the array

- Unsorted
  - n = 50: 65 swaps and 178 comparisons
  - n=100: 129 swaps and 391 comparisons
  - n=200: 302 swaps and 867 comparisons
  - n=10,000: 28,245 swaps and 67,725 comparisons

- Sorted(ascending order)
  - n = 50: 1 swap and 298 comparisons
  - n=100 : 1 swap and 148 comparisons