Analysis

(a) The median-of-three partitioning method presented here is implemented using three functions - one to calculate the median of three numbers, one to swap the median with the last value in the array if necessary, and one to actually partition the array. These functions are presented below.

```
/* Method to calculate median */
Median(int i, int j, int k){

   int temp;

   if (i > j){
      temp = i;
      i = j;
      j = temp;
   }
   if (i > j){
      temp = j;
      j = k;
      k = temp;
   }
   if (i > j){
      temp = i;
      i = j;
      j = temp;
   }

   return j;

}

/* Method to swap median with element at last index */
MO3-Swap(int[] A, int p, int r){

   x = Median(A[p], A[r], A[floor((p+r)/2)]);

   if (A[p] == x){
      exchange(A[p], A[r]);
   }

   if (A[floor((p+r)/2)] == x){
      exchange(A[floor((p+r)/2)], A[r]);
   }

}

/* Method to partition array */
Partition(int[] A, int p, int r){

   MO3-Swap(A,p,r);
   x = A[r];
   i = p - 1;
```

```
    for (int j = p; j < r; j++){
        if (A[j] <= x){
            i++;
            exchange(A[i], A[j]);
        }
    }

    exchange (A[i+1], A[r]);

    return i + 1;

}
```

(b) The worst-case performance remains the same - that is, O(n^2). This is described well on page 175 of CLRS. The difference is that with median-of-three partitioning, the case where an array consists of all duplicate entries is the is the only case where the performance is minimized, and it seems to me that this would be a very rare case.

Performance is still O(n^2) if A[p], A[r], and A[$\lfloor \frac{p+r}{2} \rfloor$] are *always* the three lowest or three highest values in the array. In this case, the partition will produce one subproblem with one element and one subproblem with $n - 2$ elements. The sum of the costs incurred at each level can be modeled as a summation of consecutive odd integers from 1 to N - this is given by $\lceil \frac{n}{2} \rceil^2$, and it is clear that this evaluates to O(n^2).

The benefit of median-of-three partitioning is that it improves the likelihood of obtaining a good partitioning and reduces the probability of achieving worst-case performance. It *does not* improve the best case performance of quicksort, and so the time complexity of quicksort with median-of-three partitioning is still Θ(n lg n).

(c) Using quicksort with median-of-three partitioning on a pre-sorted dataset results in the best case performance, which is Θ(n lg n). The value found at A[$\lfloor \frac{p+r}{2} \rfloor$] will always be the median, and will always provide the optimal partition of the array. The recurrence relation for this optimal split is given by the following:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n). \qquad (1)$$

Figure 1 shows how sorted, reverse-sorted, and random ordering all follow n lg n when utilizing the median-of-three partitioning method.

(d) All code is provided. The performance of quicksort using both the median-of-three partitioning method and the partitioning method included in CLRS is evaluated. In addition, an additional important change was made. When testing the performance for high values of n (n ¿= 10000), the standard recursive quicksort method would produce StackOverflowError. To prevent this error, a partial recursive method with implemented that would recurse only on the smaller partition. This change allowed proper functioning of quicksort up to the highest value of n tested (n = 100000). All in all, performance testing was done on all combinations of quicksort (default vs median-of-three partitioning, full vs partial recursion) and ordered, reverse-ordered, random, all duplicate, and 10% duplicate datasets with four different values of n for each (n=100, 1000, 10000, and 100000).

A few notes. First, the random, sorted, and reverse-sorted datasets do not contain any duplicate values. In addition, the 10% duplicate datasets are repeating - for example, for the n=100 dataset, the order is 1-2-3-4-5-6-7-8-9-10-1-2-3-4-5-6-7-8-9-10-... . Finally, the performance evaluation for the random dataset was averaged over five runs. In this, an important observation is that the coefficient of variation for performance of random datasets was less than 7% for all n.

Performance was evaluated by tracking the number of calls of the exchange() method and by tracking the number of calls of the quicksort() method. Figures 1-4 provide an overview of the performance.

From this, it is clear that the median-of-three partitioning method provides a significant improvement for quicksort versus simply choosing the last element as the partition. However, it is also important to implement the partial recursion method over the full recursion method in order to avoid recursive stack overflow. There is a slight hit to performance; however, I believe this would be a worthwhile modification if it prevents error.

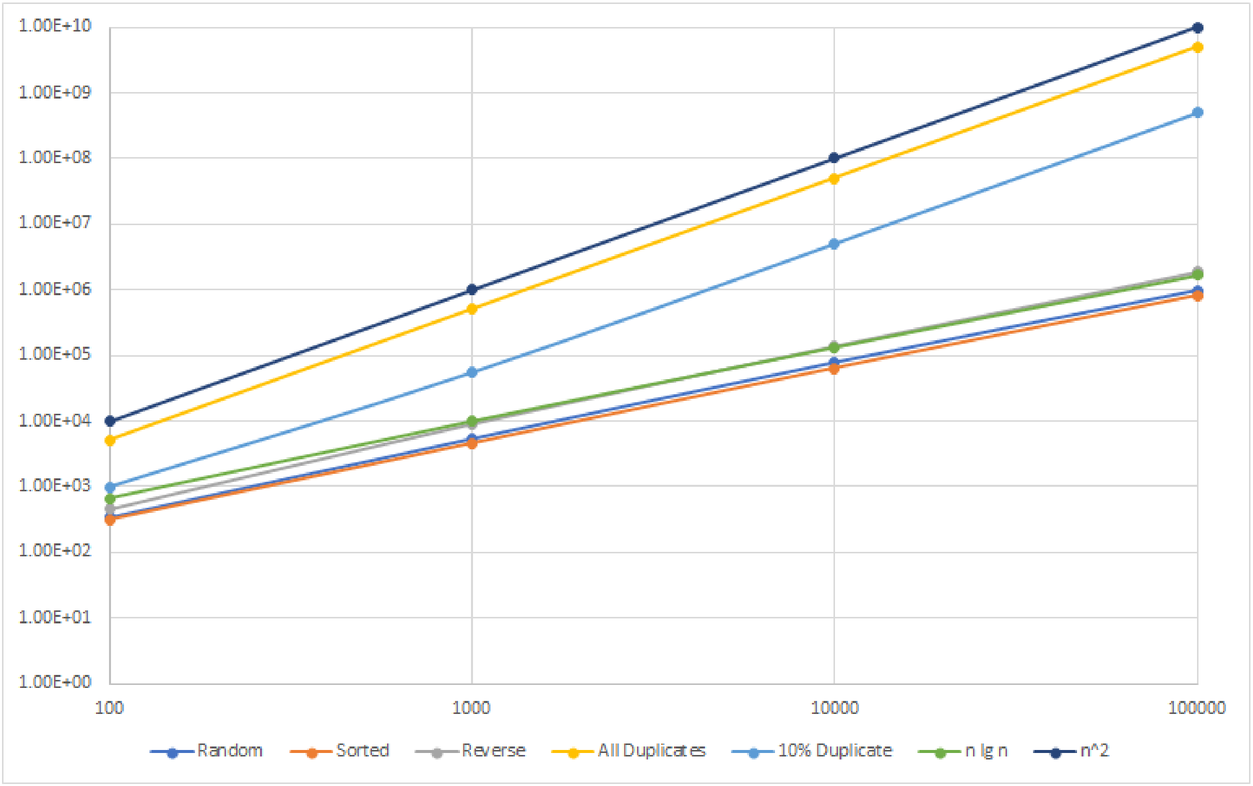


Figure 1: Performance of quicksort using the median-of-three partitioning method and partial recursion.

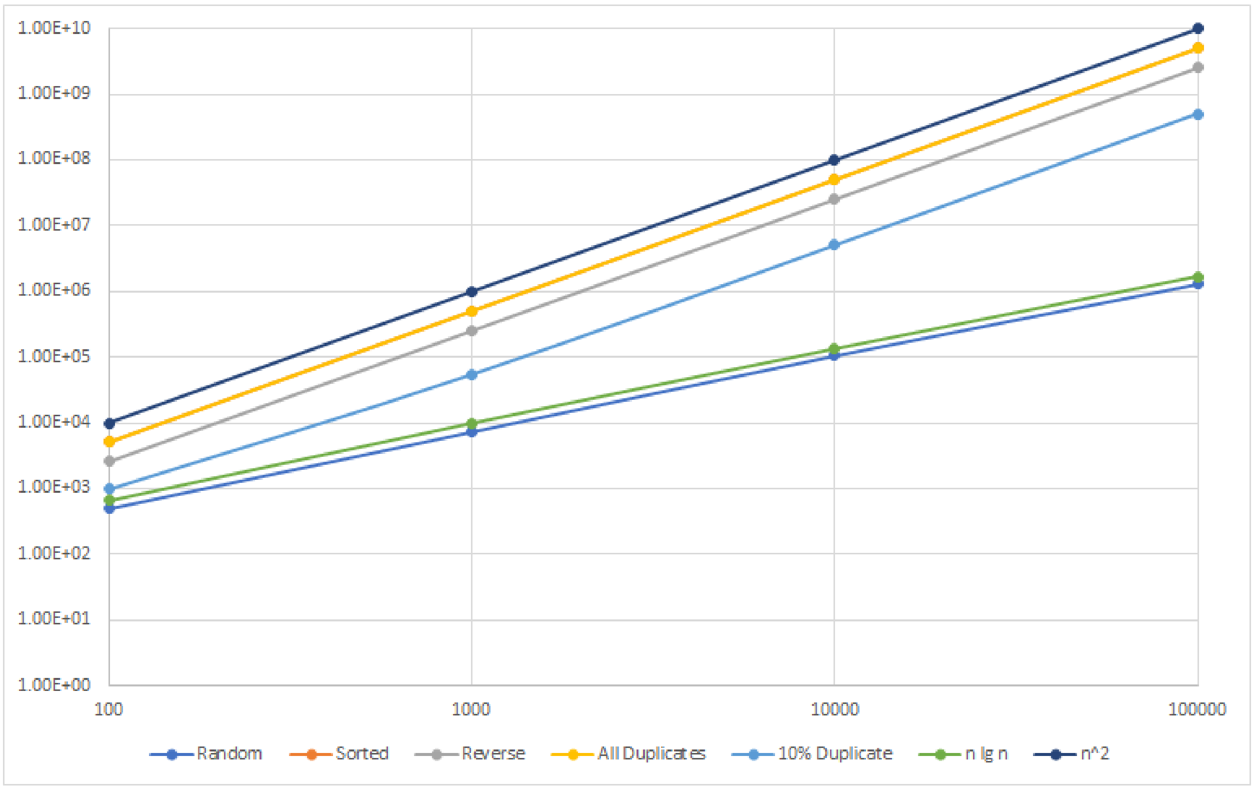
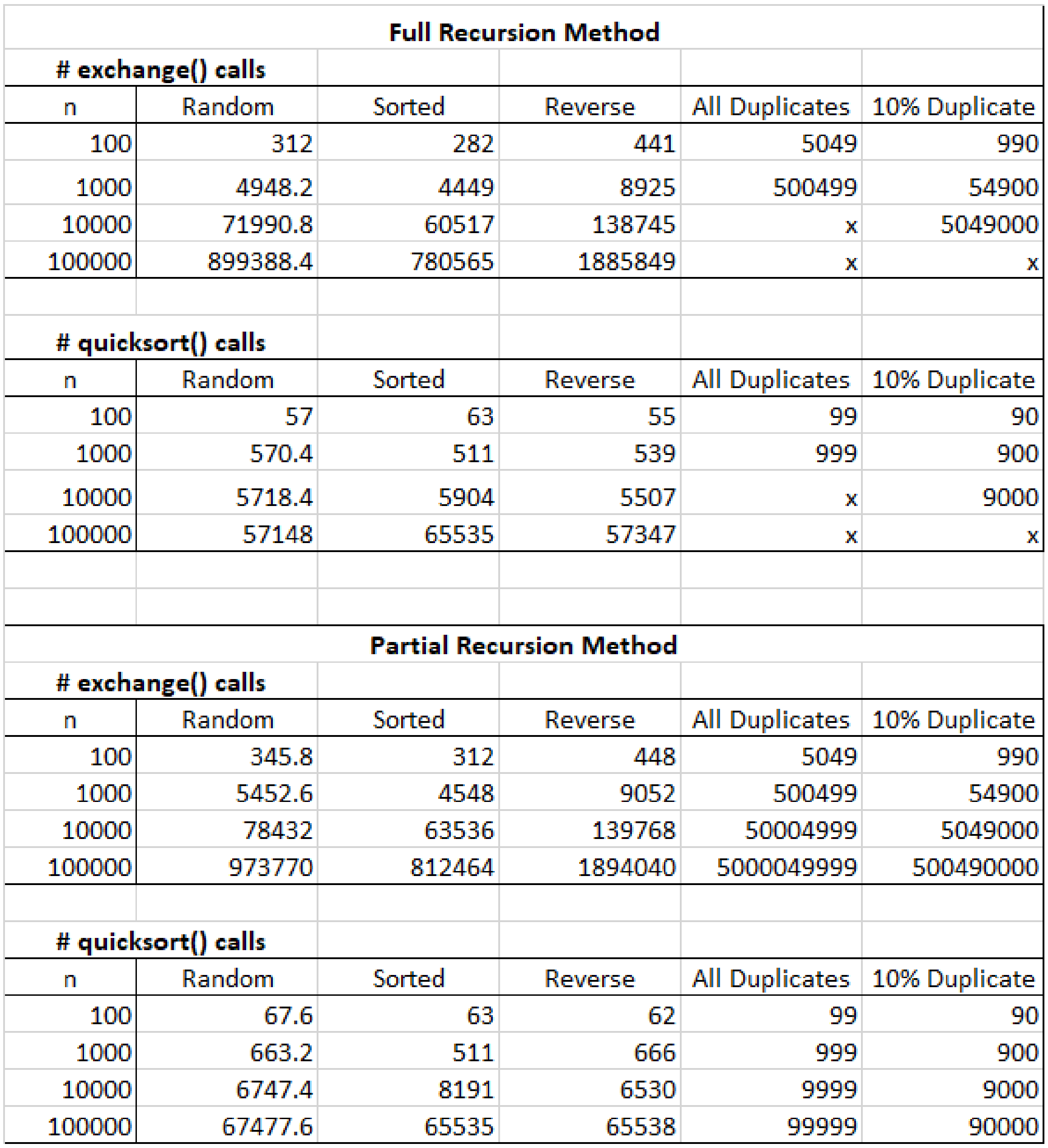


Figure 2: Performance of quicksort using the regular partitioning method and partial recursion.



Full Recursion Method

# exchange() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	312	282	441	5049	990
1000	4948.2	4449	8925	500499	54900
10000	71990.8	60517	138745	x	5049000
100000	899388.4	780565	1885849	x	x

# quicksort() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	57	63	55	99	90
1000	570.4	511	539	999	900
10000	5718.4	5904	5507	x	9000
100000	57148	65535	57347	x	x

Partial Recursion Method

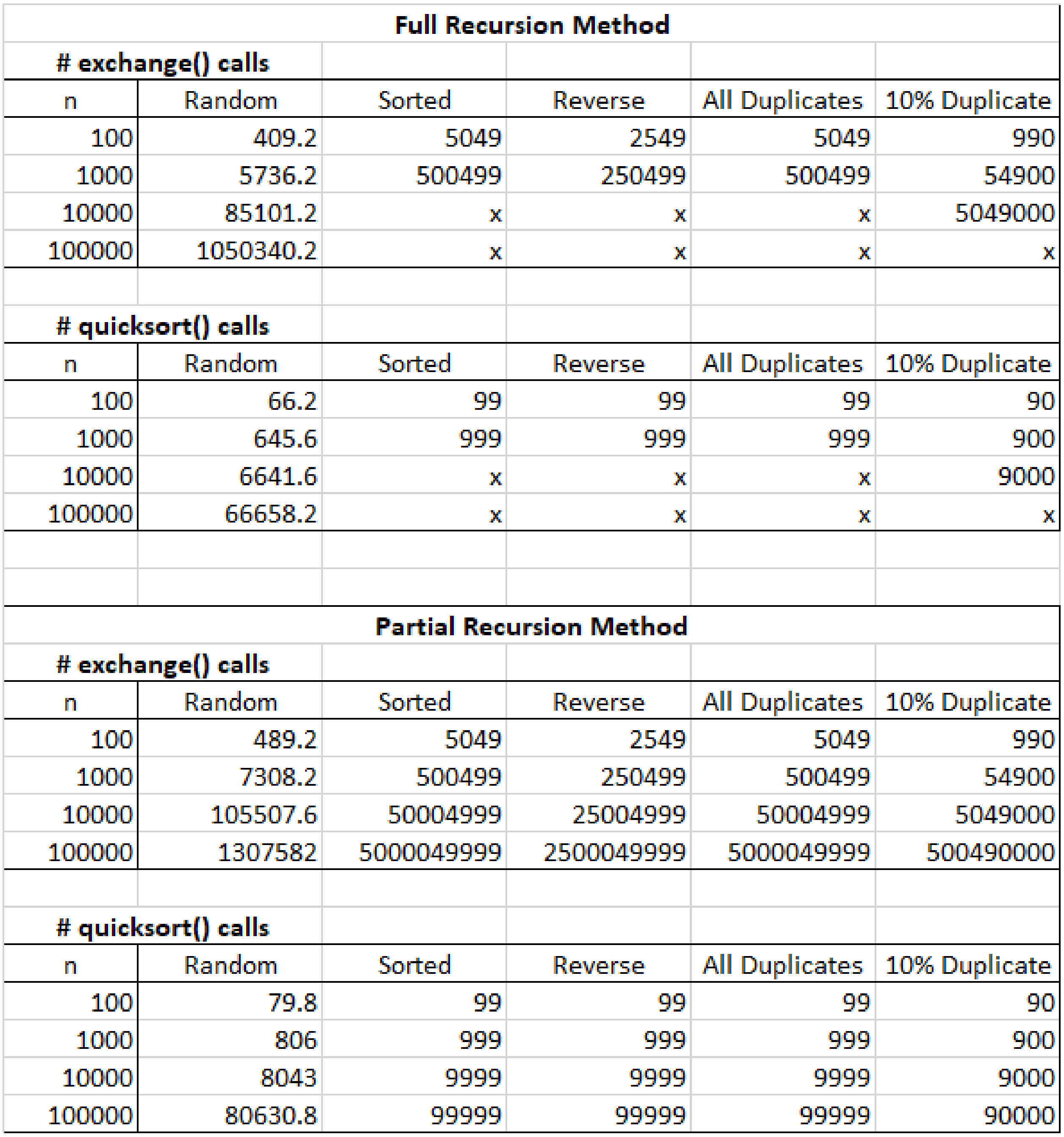
# exchange() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	345.8	312	448	5049	990
1000	5452.6	4548	9052	500499	54900
10000	78432	63536	139768	50004999	5049000
100000	973770	812464	1894040	5000049999	500490000

# quicksort() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	67.6	63	62	99	90
1000	663.2	511	666	999	900
10000	6747.4	8191	6530	9999	9000
100000	67477.6	65535	65538	99999	90000

Figure 3: Performance overview of quicksort using the median-of-three partitioning method.



Full Recursion Method

# exchange() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	409.2	5049	2549	5049	990
1000	5736.2	500499	250499	500499	54900
10000	85101.2	x	x	x	5049000
100000	1050340.2	x	x	x	x

# quicksort() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	66.2	99	99	99	90
1000	645.6	999	999	999	900
10000	6641.6	x	x	x	9000
100000	66658.2	x	x	x	x

Partial Recursion Method

# exchange() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	489.2	5049	2549	5049	990
1000	7308.2	500499	250499	500499	54900
10000	105507.6	50004999	25004999	50004999	5049000
100000	1307582	5000049999	2500049999	5000049999	500490000

# quicksort() calls

n	Random	Sorted	Reverse	All Duplicates	10% Duplicate
100	79.8	99	99	99	90
1000	806	999	999	999	900
10000	8043	9999	9999	9999	9000
100000	80630.8	99999	99999	99999	90000

Figure 4: Performance of quicksort using the regular partitioning method.

1 Appendix

Here follows the "full recursive" quicksort method.

```
if (p < r) {
   count++;
   long[] data = part.partition(array, p, r, ops);
   int q = Math.toIntExact(data[0]);
   ops = data[1];
   quicksort(array, p, q - 1);
   quicksort(array, q + 1, r);
}
```

Here follows the "partial recursive" quicksort method.

```
while(p<r){
   count++;
   long[] data = part.partition(array, p, r, ops);
   int q = Math.toIntExact(data[0]);
   ops = data[1];

   if (q - p <= r - (q + 1)){
      quicksort(array, p, q);
      p = q + 1;
   }
   else{
      quicksort(array, q + 1, r);
      r = q - 1;
   }
}
```