CS430-01 Summer 2022
Final Project Report
Group Members:

Venkata Siva Rupesh Akurati	CWID A20501754
Santosh Reddy Edulapalle	CWID A20501739
Jack Harrison Mohr	CWID A20503445

Summary of Functions

- Main.java
  - Main
    - Runs CLI menu
- Median.java
  - Median357
    - Prompts for 3, 5, and 7 integer values and finds their respective medians.
  - genericMedian
    - Prompts for an array length and a corresponding number of integer values and a requested order statistic, and returns that order statistic from the array in O(n) time complexity.
  - quicksortRandomized
    - Prompts for an input array and then times the execution of a quicksort function that uses a randomized pivot value.
  - quicksortMedianAsPivot
    - Prompts for an input array and then times the execution of a quicksort function that uses the median value of each subarray as the pivot value.
  - medianOrderStat
    - Returns the median of value of the input array.
  - randomizedSelect
    - chooses random pivot value of subarray
  - partition
    - partitions the subarray
  - quicksortRandomPivot
    - implements quicksort with a random pivot value.
  - randomizedPartition
    - used in implementing quicksortRandomPivot
  - subArrQuicksort
    - chooses median as pivot for quicksort
  - normalSort
    - assists subArrQuicksort
  - subMedian
    - picks pivot as the median of 3 elements chosen (first and last as boundaries)
  - split
    - splits the subarray around the pivot value
  - swapping
    - swaps two values in an array
- TestQuicksort.java
  - Main
    - Initiates the tests and defines parameters for the test arrays.
  - generateRandomIntArray
    - generates an array of random integers to be used as input for the quicksort tests.
  - testQuicksortRandomized
    - tests randomized pivot quicksort. Returns execution time.
  - testQuicksortMedianPivot

- tests median pivot quicksort. Returns execution time.

Partition

```
58      //method for partitioning the array
59      public int partition(int[] a, int start, int end)
60      {
61          int x = a[end], i = start;
62          for (int j = start; j <= end - 1; j++) {
63              if (a[j] < x) {
64                  // Swapping a[i] and a[j]
65                  int temp = a[i];
66                  a[i] = a[j];
67                  a[j] = temp;
68
69                  i++;
70              }
71          }
```

Both the order statistic algorithm and quicksort rely on some version of partition.
Partition runs in O(n) time complexity. This is clear since every line is some constant cost and the loop on line 62 runs n times (that is, it runs the end – start times, the number of elements in the input sub-array).

The same goes for the randomized partition version.

Analysis of ith order statistic ('randomizedSelect' function)

```
38      //randomized method to choose pivot
39      public int randomizedSelect(int[] a, int p, int r, int i)
40      {
41          // k smaller then no. of elements
42          if (i > 0 && i <= r - p + 1) {
43              //performing partition
44              int q = partition(a, p, r);
45              //pivot is same as answer
46              if (q - p == i - 1)
47                  return a[q];
48              //iterating left subarray
49              if (q - p > i - 1)
50                  return randomizedSelect(a, p, q - 1, i);
51              //recurring right subarray
52              return randomizedSelect(a, q + 1, r, i - q + p - 1);
53          }
54          //k greater than no. of elements in the array
55          return Integer.MAX_VALUE;
56      }
```

The worst case running time of randomizedSelect is $\Theta(n^2)$. As in quicksort (see below), that can only occur if the partitions are maximally inefficient, that is if each time the pivot value is the largest element so each O(n) call to partition produces an n-1 size sub-problem.

However, when the pivot is randomized, randomizedSelect has an expected time complexity of $\Theta(n)$.

Analysis of Quicksort

Comparing Run-Time of Quicksort with randomized pivots vs Quicksort that finds the median order statistic for pivots.

Test data:

We ran ten tests. For each test, we generated a new array of one million random integers. Both quicksort algorithms received the same randomly generated array as input for each test. We timed the execution time for each algorithm. The output looked like this:

Test Number	Randomized Quicksort (ns)	Median Order Statistic Quicksort (ns)
0	1749773008	34050098
1	1777374581	18491947
2	1784502033	23162293
3	1784502033	23557598
4	1789256061	23157396
5	1788730423	23280477
6	1772594758	23013775
7	1772226291	23119687
8	1780479923	23881109
9	1783400064	23294507
Mean (arithmetic)	1778283917.5	23900888.7

Randomized quicksort mean runtime: 1.778 seconds

Median order statistic mean runtime: .0238 seconds

Performance ratio (randomizedQuicksort/medianQuicksort) = 1.778/.0238 = 7.476

Analysis:

In our test, quicksort executed on average approximately 7 times faster when using the median order statistic as a pivot value rather than a randomized pivot value. Why?

Why is quicksort more efficient when using the median order statistic as its pivot value rather than a random pivot value?

"The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort." – *Introduction to Algorithms*, chapter 7.2

Intuitively, the algorithm's efficiency depends on the number of recursive calls it has to make, which in turn depends on the pivot value. The partitioning of an n-element sub-array costs $\Theta(n)$ time, since the algorithm has to sequentially compare each element with the pivot value. The worst-case is when the pivot is either the largest or the smallest element in the subarray. In that case, partition will produce two sub-problems: one of size n-1 and the other of size 0, since basically, all that has been determined is that all the values in the sub-array are either bigger or smaller than the pivot value.

Since the partitioning costs $\Theta(n)$, this worst-case behavior or partition will create a recurrence:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

Clearly, that is going to produce n sub-problems of size $\Theta(n)$, for a total runtime cost of $\Theta(n^2)$.

In contrast, choosing the median value as the pivot in the subproblem is more efficient because the partition algorithm will still cost $\Theta(n)$ but will produce two subproblems, each of size n/2, which ignoring the floor/ceiling operations, the recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

By the master theorem for recurrences, this yields a runtime of $\Theta(nlgn)$.

Now, a randomized version will usually produce sub-optimal sub-problems because it will usually not choose the median value, thus creating more sub-problems than necessary. It follows that a randomized version will be less efficient.