Simultaneous Minimum and Maximum

1. The Simultaneous Minimum and Maximum selection algorithm, does not actually modify the existing array therefore the variable “numSwaps” in this case just references the number of times a new minimum or maximum was stored in their respective variables.
2. The Simultaneous Minimum and Maximum Algorithm had better results on randomized arrays with respect to the number of comparisons and swaps compared to almost sorted arrays.
3. These results make sense due to the nature of the input parameters (almost sorted vs randomized). While the minimum element is resolved as fast as expected on almost sorted data since it’s the first element, the maximum element will nearly always be at the end of the array.
4. Consequently, in order to find the simultaneous maximum the algorithm must search until the very end of the array. This is inefficient and explains why randomized input will on average be faster since the variables will not be updated as often.
5. The Simultaneous Minimum and Maximum is optimal in regard to the number of comparisons and swaps for more randomized data sets.

Kth Smallest Item

1. The Randomized Select Algorithm is used to find the kth smallest element in an array. It behaves similar to a Quicksort however instead of recursing throughout both partitions of the array it will only need to recurse one side of the array in order to find the Kth smallest element.
2. For instance, if i = 3, we want to find the 3rd smallest element in the array. Your output would look something like this:

   i = 3
   i'th smallest element is 43
   [3, 28, 43, 87, 469, 311, 483, 304, 442, 298, 202, 170, 391, 127, 387, 392]

3. Notice that the array is sorted up to the 3rd element, and the rest of the array remains unchanged. The nature of the Quicksort algorithm allows us to stop sorting the array once the pivot index reaches the desired value.
4. The Randomized Select Algorithm reached its worse case with respect to the number of swaps on almost sorted arrays however these results are inconsistent for several reasons.
5. The value of k can influence the results and depending on the randomized partition. Therefore depending on the input the results could sway in favor of either set of input parameters.
6. This algorithm should not be judged alone by the amount of comparisons and swaps, and due to the randomized aspects of the implementation, misleading and unpredictable results can occur.

Median of Medians:

1. The Median of Medians algorithm attempts to find the *approximate* median of an array by dividing the entire array into subarrays of length five and then by resolving the medians of all the subarrays into another array which is then sorted to find the median.
2. The implementation of this algorithm can be done in a variety of different ways and I chose to additionally make my implementation work for arrays that are not divisible by 5. This is accomplished by the subroutine *median2*, with the cost of additional comparisons and swaps to the final result.
3. The Median of Medians performed worse with respect to number of comparisons and swaps on almost sorted arrays.
4. The problem with evaluating performance based off swaps and comparisons on this selection algorithm is that it will provide misleading results because the process of finding the Median of Medians is going to always generate additional swaps and comparisons even a traditional quicksort implementation.

Notes:
The process of finding the median of medians is meant to serve as a way to find a more optimal balance of partitioning. If you count the number of swaps and comparisons of this process then you are going to be led to believe that it is inferior to a randomized quicksort or a median of three quicksort if your evaluation only takes the number of swaps and comparisons into account.