***If turn in your project using email as a time stamp, please put down its date and time:***

_____

Name: _____

*(Use this page as the cover sheet when you turn in your project report)*

# Project #2
# CS331 Design and Analysis of Algorithms

| | | |
|---|---|---|
| 1. (40 points) | Data Sets, Test Strategies and Explanation of Results | |
| 2. (15 points) | Select 2 Vs Select 3 | |
| 3. (15 points) | Select 4 Vs Select 1 | |
| 4. (15 points) | Strength and Constraints of Your Work | |
| 5. (15 points) | Program Correctness | |
| 6. (-10 points per day) | Late Penalty | |
| **(100 points)** | **Total** | |

1: 40 points
2: 15 points
3: 15 points
4: 15 points
5: 15 points

Total: 100 points

_____

100

**Data Sets:**

The data values inside the array of size N make up anywhere from 1 to (N * N) which are randomly generated. In the main class, I call generate() method and pass in a size as an argument. The method generates an array of size passed in the argument and fills the array with randomly generated values. To note, all the array values are unsorted and distinct. To verify uniqueness in values, I call the method contains() to test whether the randomly generated value already exists in the array.

**Test Strategies:**

I create a size N array where N doubles every iteration in an endless loop which begins at 10. Now, I am computing the four different methods of same size five times and am getting the average of the computation time for all four approaches. This process is repeated by doubling the array size by every iteration so long until the computer crashes. The data values inside the array make up anywhere from 0 to N*N which are randomly generated. The result would a print out the average calculated run times for Select-kth 1, Select-kth 2, Select-kth 3, and Select-kth 4 in addition to displaying the array and showing the kth smallest element for the five k values specified in the project (1, N/4, N/2, 3N/4, and N) for each array size.

**Results:**

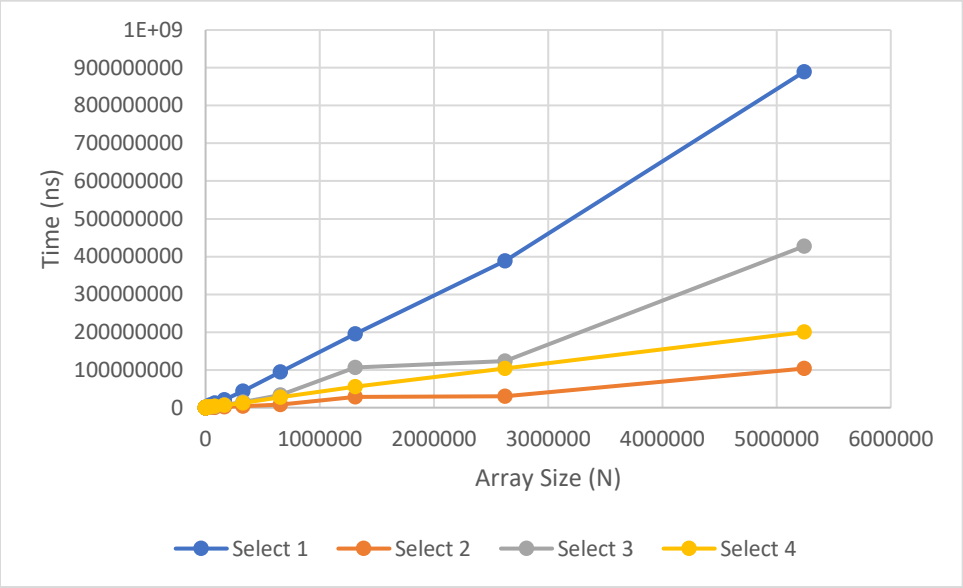
| Size (N) | Select-kth 1 (ns) | Select-kth 2 (ns) | Select-kth 3 (ns) | Select-kth 4 (ns) |
|---|---|---|---|---|
| 10 | 9102 | 4323 | 5802 | 54158 |
| 20 | 22755 | 14631 | 13675 | 36090 |
| 40 | 21618 | 6339 | 6148 | 33045 |
| 80 | 16497 | 5477 | 9421 | 67025 |
| 160 | 33564 | 28060 | 43526 | 113529 |
| 320 | 84992 | 59201 | 134201 | 169023 |
| 640 | 154214 | 48932 | 116838 | 109125 |
| 1280 | 300951 | 87610 | 236814 | 166322 |
| 2560 | 598814 | 273408 | 609538 | 418060 |
| 5120 | 1093131 | 842478 | 308503 | 358498 |
| 10240 | 1707749 | 283183 | 368104 | 628414 |
| 20480 | 3232642 | 265532 | 575608 | 1004729 |
| 40960 | 7488211 | 486372 | 1299434 | 2096483 |
| 81920 | 12086823 | 904527 | 3360103 | 3580497 |
| 163840 | 21178404 | 1899063 | 7168502 | 6289047 |
| 327680 | 43526549 | 3515982 | 14780285 | 12743786 |
| 655360 | 94425918 | 7750248 | 33363614 | 27232498 |
| 1310720 | 195290751 | 28152196 | 106404719 | 55902833 |
| 2621440 | 388444234 | 29766453 | 123739168 | 103671178 |
| 5242880 | 889767264 | 103761305 | 427699683 | 200126127 |

First of all, all four are graphed and completed accurately. The kth values found are accurate, and there is a clear pattern between the four approaches. Out of all four approaches, Select 1 is by far the slowest, and the fastest comes out to be Select 2 with Select 4 only a little slower than it, and

Select 3 being a little more slower than Select 4. It's understandable as to why Select 1 is the slowest, but a little bit of explanation is needed for why Select 2 is the fastest. It's extremely unlikely that Select 2 was in its best case, so the remaining explanation is that Select 4 will beat out Select 2 in even larger array sizes such as in high millions. And, as observed in the results, Select 3 will be slower than Select 2 as expected because recursion uses stack which takes place in memory and it takes time.

Find out when does Select-kth 4 become faster than Select-kth 1? At array size 640.

Is Select-kth 2 always faster than Select-kth 3? Almost most of the time. 16 out of 19 times.




**Select 2 Vs Select 3:**

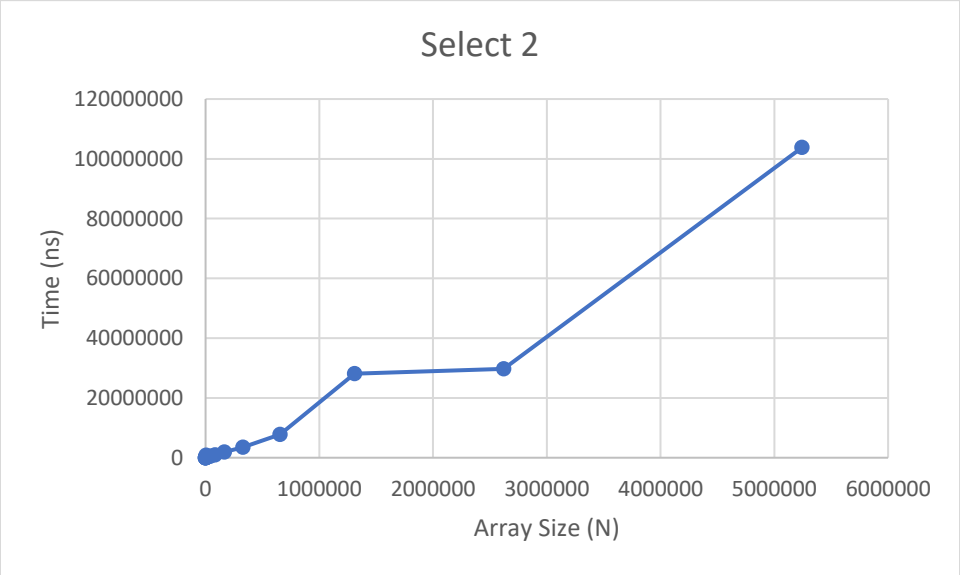
The algorithm for Select 2 and Select 3 is to apply the procedure Partition used in Quicksort. The procedure partitions an array so that all elements smaller than some pivot item come before it in the array and all elements larger than that pivot item come after it. The slot at which the pivot item is located is called the pivotposition. I solve the Selection Problem by partitioning until the pivot item is at the kth slot. Here's a bit of explanation in technical words:

For each iteration:

If pivot position i = k ⇒ Done! Terminate the program.
If pivot position i < k ⇒ Select (k-i)th element in the 2nd sub-list.
If pivot position i > k ⇒ Select kth element in the 1st sub-list.

In Select 2, the above-mentioned algorithm is implemented using the partition procedure of Quicksort iteratively using a loop. In Select 3, the above-mentioned algorithm is implemented recursively by partitioning the left subarray if k is less than pivotposition, and by recursively partitioning the right subarray if k is greater than pivotposition. When k = pivotposition, we're done and return. Both approaches are very similar in the manner that objective for both of them is the same. The only difference between is that Select 2 is done iteratively with a loop, while Select 3 is done recursively.

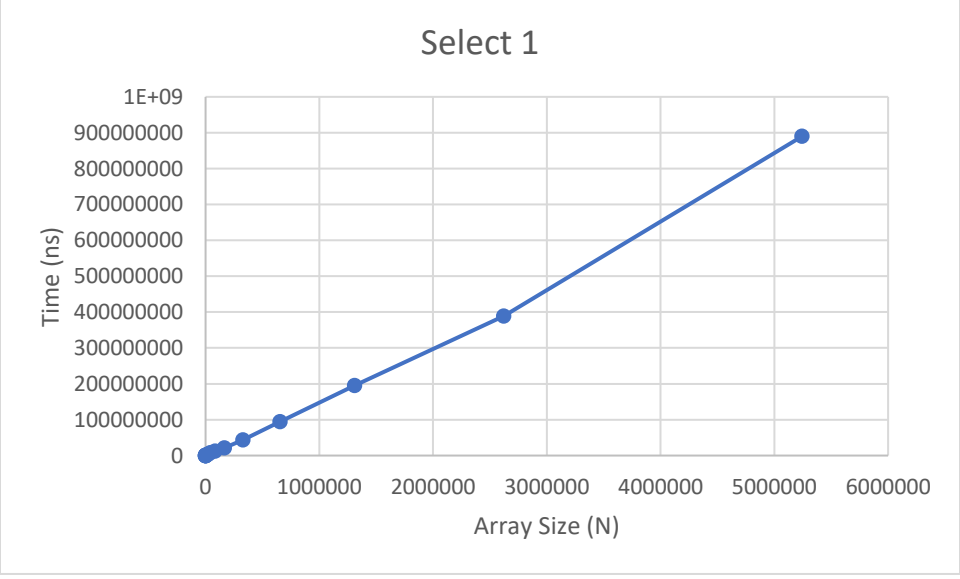
The best-case complexity of the aforementioned algorithm is O(n) — when pivot is the kth smallest one — while the worst case is $O(n^2)$ — call partition O(n) times, each time takes O(n). Comparing to the experimental data, we could clearly see that Select 2 is faster than Select 3. Although both have the same time complexity, the reason for the difference in time could be correlated to how Select 3 is implemented. The method Select 3 is slower simply due to the fact that recursion uses stack and memory is in use. That, in itself, increases the time for the method when compared to iterative Select 2. Furthermore, the reason for bumpiness in the graph (i.e., the graph not having a consistent slope) could be because the time complexity for each distinct size could be anywhere from O(n) to $O(n^2)$. A sudden drop at 2.621 million could be due to the randomly generated array being close to the best case. Thus, because of a range in time complexity, the slope is not consistent for different size results.

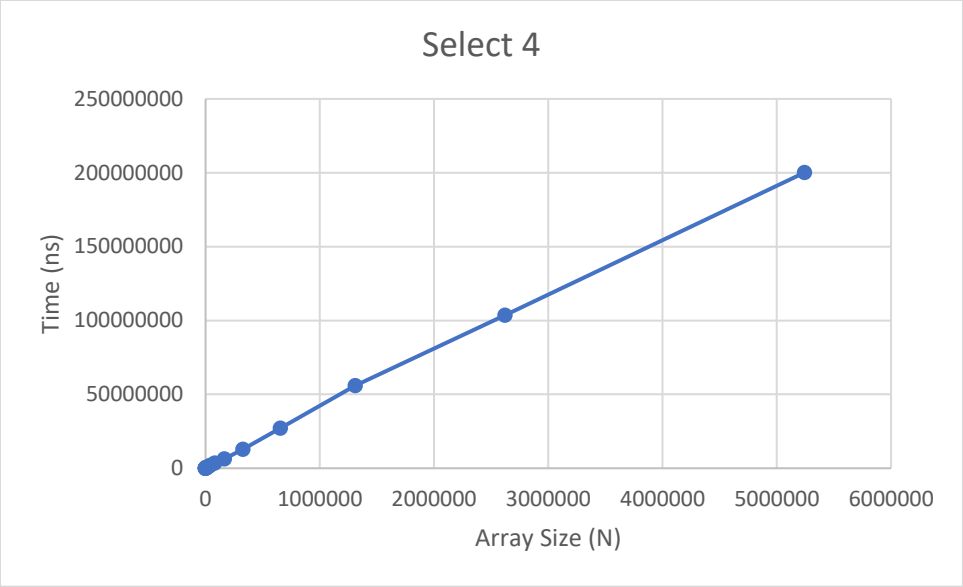



**Select 4 Vs Select 1:**

Select 1 is the most straightforward one out of all the approaches. This algorithm sorts the array using merge sort and then returns the kth smallest element. This sorting algorithm keeps on dividing the array until it can no longer be divided (i.e.: we achieve atomic values), and then combine while comparing and sorting in the same manner they were broken down. It takes O(n*log(n)) amount time in the best, average and worst case. Since the time complexity is the same for best, average, and worst, we should expect a fairly consistent slope and that's exactly what we see in the experimental result. The obtained result makes logical sense.

Select 4 is similar to algorithm for Select 2 and Select 3, except that it always finds an element called median of medians and uses that as a pivot. The way I find that element is as such: I divide the array by five, then find the medians in all sub-arrays and store all the medians in another array. Subsequently, I find the median of medians of the array that holds the medians of all the sub-arrays. Finally, I set the median of medians as the pivot item and use it to compare other elements of the array against the pivot. If an element is less than the pivot value, the element is placed to the left of the pivot, and if the element has a value greater than the pivot, it is placed to the right. The algorithm recurses on the array, honing in on the value (kth-smallest) it is looking for as a result. The worst-case time complexity is O(n). When connecting to the obtained results for Select 4, I could make a connection to Select 1, which was expected. Similar to Select 1, the algorithm for Select 4 yields a fairly consistent slope and it makes logical sense as to why. Because the time complexity is O(n) in best and worst, it is expected to have consistent slope similar to Select 1. Such is exactly what's depicted on the graph.






**Strengths and Constraints:**

My computer information:


The project involves implementing and analyzing four different algorithms for the Selection Problem. I used the Java language and coded and tested the project in the latest Eclipse software as of the project due date.

Overall, the program is running accurately and accomplishes all the aspects of the project well without any errors. Based on the data, I can depict the pattern of the four different methods and relate it back to the theoretical component of them respectively. It does not match perfectly because to be able to correlate one to one the experiment to theoretical, I would need more data from larger array sizes that are in large millions and higher to clearly see the correlation pattern. At that time, I predict that Select 4 should be the fastest.

The best part of the project is the actual implementation of the four approaches. The implementation is very clear, easy to understand, and efficient. If I were to do the project again, I would implement in a slightly more efficient manner even though current code perfectly follows the requirements. For example, I realized that Select 1 could be implemented in a slightly more efficient manner. If I had more time, I would re-run my program on a desktop computer with perhaps a better way of implementing the supplemental methods and making them quicker. For

example, the method for finding the medians. However, I would aspire and obtain the test results of all the four different approaches on arrays of very large sizes. I would need a computer that is extremely fast in terms of computation power and one where there are no concerns with memory usage. Generally speaking, the faster the computer, the results would be better suited to what's expected. The reason that I do not have arrays of sizes in large millions or higher listed is simply because it takes a lot of processing power and memory to calculate them. Obtaining the highest array size listed here of roughly five and a quarter million already takes nearly four hours. Thus, I manually stop the program as per instruction mentioned in the class because it takes a very long time to compute.

Although the data does not appear to perfectly mirror the time complexity of each approach, it is accurate representation of the method and the reason for slight deviation from expected is due to the nature of array size. Better expected results would come with larger size that's not being reflected in the results.

**Program Correctness:**

In addition to my code attached for correctness, below is my sample output for the beginning array size of ten. The following below being correct means that the four approaches are accurately accomplished. In here, array contains {19, 20, 10, 8, 2, 11, 15, 12, 4, 13}. As per the instruction, I find the 1st, 2nd, 5th, 7th, and 10th k position values which are {2, 4, 11, 13, 20} respectfully. Such is exactly the same in the screenshot provided.

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jdk-10.0.2\bin\javaw.exe (May 1, 2019, 6:13:13 PM)
The array contains:
19, 20, 10, 8, 2, 11, 15, 12, 4, 13
The 1 smallest value in the array found using Select-kth 1 is: 2
The 1 smallest value in the array found using Select-kth 2 is: 2
The 1 smallest value in the array found using Select-kth 3 is: 2
The 1 smallest value in the array found using Select-kth 4 is: 2
The 2 smallest value in the array found using Select-kth 1 is: 4
The 2 smallest value in the array found using Select-kth 2 is: 4
The 2 smallest value in the array found using Select-kth 3 is: 4
The 2 smallest value in the array found using Select-kth 4 is: 4
The 5 smallest value in the array found using Select-kth 1 is: 11
The 5 smallest value in the array found using Select-kth 2 is: 11
The 5 smallest value in the array found using Select-kth 3 is: 11
The 5 smallest value in the array found using Select-kth 4 is: 11
The 7 smallest value in the array found using Select-kth 1 is: 13
The 7 smallest value in the array found using Select-kth 2 is: 13
The 7 smallest value in the array found using Select-kth 3 is: 13
The 7 smallest value in the array found using Select-kth 4 is: 13
The 10 smallest value in the array found using Select-kth 1 is: 20
The 10 smallest value in the array found using Select-kth 2 is: 20
The 10 smallest value in the array found using Select-kth 3 is: 20
The 10 smallest value in the array found using Select-kth 4 is: 20
For array size = 10:
        Select-kth 1 time: 31175 nanoseconds.
        Select-kth 2 time: 15815 nanoseconds.
        Select-kth 3 time: 26623 nanoseconds.
        Select-kth 4 time: 209578 nanoseconds.
```

```
import java.util.Arrays;
import java.util.Random;
/*
 * A class that presents four methods of finding the
 * kth-smallest value in the array.
 */
public class Select
{
	/*
	 * Find the kth-smallest value using mergersort. Given an array,
	 * beginning index, ending index, and the kth index inside,
	 * perfrom sorting and then return the kth index since it
	 * will automatically be the kth-smallest value in the array.
	 */
	public int select_1(int inputArray[], int beginningIndex, int endIndex, int k)
	{
	    if (k > 0 && k <= endIndex - beginningIndex + 1)
	    {
	      inputArray = mergeSort(inputArray);
	      return inputArray[k - 1];
	    } else
	        return -1;
	}

	/*
	 * Given an array, sort it using Mergersort.
	 */
	private int[] mergeSort(int[] inputArray)
	{
	  int inputArraySize = inputArray.length;
	  if (inputArraySize <= 1)
	      return inputArray;

	  int[] firstHalfOfInputArray = new int[inputArraySize / 2];
	  int[] secondHalfOfInputArray = new int[inputArraySize - (inputArraySize / 2)];

	  for (int i = 0; i < firstHalfOfInputArray.length; i++)
	      firstHalfOfInputArray[i] = inputArray[i];
	  for (int i = 0; i < secondHalfOfInputArray.length; i++)
	      secondHalfOfInputArray[i] = inputArray[i + (inputArraySize / 2)];

	  return merge(mergeSort(firstHalfOfInputArray), mergeSort(secondHalfOfInputArray));
	}

	/*
	 * Helper method for Mergesort. Combines two arrays accordingly.
	 */
	private int[] merge(int[] firstArray, int[] secondArray)
	{
	  int[] combinedArray = new int[firstArray.length + secondArray.length];
	  int firstArrayCounter = 0, secondArrayCounter = 0;
	  for (int k = 0; k < combinedArray.length; k++)
	  {
	      if (firstArrayCounter >= firstArray.length)
	            combinedArray[k] = secondArray[secondArrayCounter++];
	      else if (secondArrayCounter >= secondArray.length)
	            combinedArray[k] = firstArray[firstArrayCounter++];
	      else if (firstArray[firstArrayCounter] <= secondArray[secondArrayCounter])
	            combinedArray[k] = firstArray[firstArrayCounter++];
	      else
	            combinedArray[k] = secondArray[secondArrayCounter++];
	  }
	  return combinedArray;
	}
```

```
/*
 * Find the kth-smallest value using the partition procedure of
 * Quicksort iteratively. Given an array, beginning index, ending index,
 * and the kth index inside, return the kth-smallest value.
 */
public int select_2(int inputArray[], int beginningIndex, int endIndex, int k)
{
      if (k >= 0 && k <= endIndex - beginningIndex + 1)
      {
            int pivot;
            while(true)
            {
                  if (beginningIndex == endIndex)
                        return inputArray[beginningIndex];

                  pivot = inputArray[endIndex];
                  pivot = partition(inputArray, beginningIndex, endIndex, pivot);

                  if (k == pivot)
                        return inputArray[k];
                  else if (k < pivot)
                        endIndex = pivot - 1;
                  else
                        beginningIndex = pivot + 1;
            }
      } else
            return -1;
}



/*
 * Find the kth-smallest value using the partition procedure of
 * Quicksort recursively. Given an array, beginning index, ending index,
 * and the kth index inside, return the kth-smallest value.
 */
public int select_3(int inputArray[], int beginningIndex, int endIndex, int k)
{
      if (k >= 0 && k <= endIndex - beginningIndex + 1)
      {
            if (beginningIndex == endIndex)
                  return inputArray[beginningIndex];

            int pivot = inputArray[endIndex];
            pivot = partition(inputArray, beginningIndex, endIndex, pivot);

            if (k == pivot)
                  return inputArray[k];
            else if (k < pivot)
                  return select_3(inputArray, beginningIndex, pivot - 1, k);
            else
                  return select_3(inputArray, pivot + 1, endIndex, k);
      } else
            return -1;
}




/*
 * Find the kth-smallest value using the median of medians
 * technique recursively. Given an array, beginning index,
 * ending index, and the kth index inside, return the kth-smallest value.
 */
```

```
    public int select_4(int inputArray[], int beginningIndex, int endIndex, int k)
    {
      if (k > 0 && k <= endIndex - beginningIndex + 1)
      {
          int inputArraySize = endIndex - beginningIndex + 1;
          int tempLocator, median[] = new int[(inputArraySize + 4) / 5];

          for (tempLocator = 0; tempLocator < median.length - 1; tempLocator++)
          {
                median[tempLocator] = findMedian(Arrays.copyOfRange(inputArray, 5 *
                tempLocator + beginningIndex, * tempLocator + beginningIndex + 4), 5);
          }

          if (inputArraySize % 5 == 0)
          {
                median[tempLocator] = findMedian(Arrays.copyOfRange(inputArray, 5 *
                tempLocator + beginningIndex, 5 * tempLocator + beginningIndex + 4), 5);

                tempLocator++;
          } else
          {
                median[tempLocator] = findMedian(Arrays.copyOfRange(inputArray, 5 *
                tempLocator + beginningIndex, 5 * tempLocator + beginningIndex +
                (inputArraySize % 5)), inputArraySize % 5);

                tempLocator++;
          }

          int medianOfMedians;
          if (tempLocator == 1)
                medianOfMedians = median[tempLocator - 1];
          else
                medianOfMedians = select_4(median, 0, tempLocator - 1, tempLocator / 2);


int partitionLocator = partition(inputArray, beginningIndex, endIndex, medianOfMedians);

          if (partitionLocator - beginningIndex == k - 1)
                return inputArray[partitionLocator];
          if (partitionLocator - beginningIndex < k - 1)
                return select_4(inputArray, partitionLocator + 1, endIndex, k -
                (partitionLocator + 1) + beginningIndex);

          return select_4(inputArray, beginningIndex, partitionLocator - 1, k);
      } else
          return -1;
    }

    /*
    * Returns the median element of the array. Given the array,
    * and an index, return a median value of the array.
     */
    private int findMedian(int inputArray[], int inputArraySize)
    {
      Arrays.sort(inputArray);
      return inputArray[inputArraySize / 2];
    }




    /*
    * Given array, index i, index j, swap i and j.
     */
```

```
private int[] swap(int[] inputArray, int i, int j)
{
  int temp = inputArray[i];
  inputArray[i] = inputArray[j];
  inputArray[j] = temp;
  return inputArray;
}

/*
 * Perform partition given the array, beginning index, end index, and arbitrary pivot.
 */
private int partition(int[] inputArray, int beginningIndex, int endIndex, int pivot)
{
  for (int i = 0; i < inputArray.length; i++)
  {
      if (inputArray[i] == pivot)
      {
          swap(inputArray, i, endIndex);
          break;
      }
  }

  int tempCounter = beginningIndex;
  int index = beginningIndex - 1;

  while (tempCounter < endIndex)
  {
      if (inputArray[tempCounter] < pivot)
      {
          index++;
          swap(inputArray, tempCounter, index);
      }
      tempCounter++;
  }
  index++;
  swap(inputArray, index, endIndex);

  return index;
}

/*
* Given an array, print the contents of the array
 */
public void display(int[] inputArray)
{
  System.out.println("The array contains: ");

  for(int i = 0; i < inputArray.length; i++)
  {
      System.out.print(inputArray[i]);

      if(i == inputArray.length - 1)
            continue;
      System.out.print(", ");
  }
  System.out.println("");
}
```

```
	/*
	 * Given a size n, generate unsorted array with distinct values
	 */
	public int[] generate(int arraySize)
	{
	    int arr[] = new int[arraySize];
	    Random randomInteger = new Random();

	    for(int i = 0; i < arraySize; i++)
	    {
	      boolean included = true;

	      while(included)
	      {
	            int temp = randomInteger.nextInt(Math.abs(2*arraySize)) + 1;
	            boolean contains = contains(arr, temp);
	            if (!contains)
	            {
	                  included = false;
	                  arr[i] = temp;
	            }
	      }
	    }

	    return arr;
	}

	/*
	 * Given an array and a value, search for the value
	 * in the array and return
	 */
	private boolean contains(int[] inputArray, int targetValue)
	{
	      for (int temp : inputArray)
	      {
	            if (temp == targetValue)
	                  return true;
	      }

	      return false;
	}



}
```

```
/*
 * Driver class to test the Select-kth algorithms
 */
public class Test
{
	/*
	 * Main method to test out the average time for each algorithm for same problem.
	 */
	public static void main(String[] args)
	{
		Select Select_kth = new Select();
		final int REPEAT_TIMES = 5;
		long totalTimeSelect1 = 0, totalTimeSelect2 = 0, totalTimeSelect3 = 0,
		totalTimeSelect4 = 0;
		for (int i = 10; i > 0; i*=2)
		{
		  int[] nums = Select_kth.generate(i);
int[] kthSmallest = {1, nums.length/4, nums.length/2, (3*nums.length)/4, nums.length};
		  int n = nums.length;
		  Select_kth.display(nums);
		  long startTime, endTime;
		  for (int j = 0; j < REPEAT_TIMES; j++)
		  {
		      startTime = System.nanoTime();
		      int mergeSortResult = Select_kth.select_1(nums, 0, n - 1, kthSmallest[j]);
		      endTime = System.nanoTime();
		      System.out.println("The " + kthSmallest[j] + " smallest value in the array
		      found using Select-kth 1 is: " + mergeSortResult);
		      totalTimeSelect1 += endTime - startTime;

		      startTime = System.nanoTime();
int partitionQuicksortIterativeResult = Select_kth.select_2(nums, 0, n-1, kthSmallest[j]-1);
		      endTime = System.nanoTime();
		      System.out.println("The " + kthSmallest[j] + " smallest value in the array
		      found using Select-kth 2 is: " + partitionQuicksortIterativeResult);
		      totalTimeSelect2 += endTime - startTime;

		      startTime = System.nanoTime();
int partitionQuicksortRecursiveResult = Select_kth.select_3(nums, 0, n-1, kthSmallest[j]-1);
		      endTime = System.nanoTime();
		      System.out.println("The " + kthSmallest[j] + " smallest value in the array
		      found using Select-kth 3 is: " + partitionQuicksortRecursiveResult);
		      totalTimeSelect3 += endTime - startTime;

		      startTime = System.nanoTime();
int medianOfMediansResult = Select_kth.select_4(nums, 0, n - 1, kthSmallest[j]);
		      endTime = System.nanoTime();
		      System.out.println("The " + kthSmallest[j] + " smallest value in the array
		      found using Select-kth 4 is: " + medianOfMediansResult);
		      totalTimeSelect4 += endTime - startTime;
		  }
		  totalTimeSelect1 /= REPEAT_TIMES;
		  totalTimeSelect2 /= REPEAT_TIMES;
		  totalTimeSelect3 /= REPEAT_TIMES;
		  totalTimeSelect4 /= REPEAT_TIMES;
		  System.out.println("For array size = " + n + ": " +
		      "\n\tSelect-kth 1 time: " + totalTimeSelect1 + " nanoseconds." +
		      "\n\tSelect-kth 2 time: " + totalTimeSelect2 + " nanoseconds." +
		      "\n\tSelect-kth 3 time: " + totalTimeSelect3 + " nanoseconds." +
		      "\n\tSelect-kth 4 time: " + totalTimeSelect4 + " nanoseconds.\n");
		  System.out.println("\n \n \n");
		  }
	}
}
```