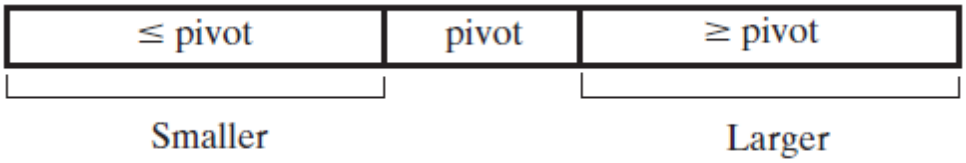
Comp151 Lab07

In Lab07 you will implement methods that utilize faster soring algorithms. Please read the instructions carefully.

I. We have an unsorted array of integers and we want to find the value of an element at index ***k*** that would be there if the array was sorted.

The simplest way to find the ***k^th^*** element is to sort the data and take the value that is at the index ***k***. But sorting does more than necessary to solve this problem. You need to find only the ***k^th^*** smallest entry in the collection for an appropriate value of ***k***. We can use the partitioning strategy of quick sort to find the ***k^th^*** smallest entry in the array:

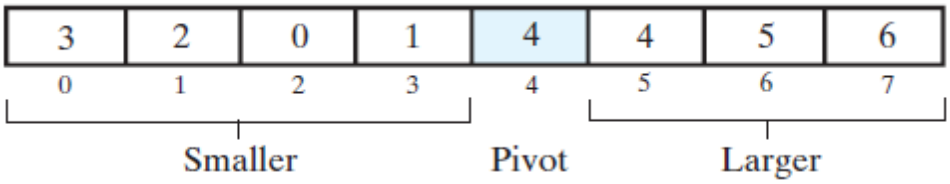
After choosing a pivot and forming the sub-arrays *Smaller* and *Larger*



you can draw **one** of the following **three** conclusions:

1) if ***pivot index*** is the same as ***k***, the ***k^th^*** smallest element is the entry at the ***pivot index***
2) if *Smaller* sub-array contains ***k*** or more entries, it must contain the ***k^th^*** smallest entry
3) otherwise the ***k^th^*** element is in *Larger* sub-array

For example, let us assume that we are looking for the smallest entry that would be at index **3** if the array was sorted. The initial array is: `3 5 0 4 6 1 2 4` . After the first partition it will look as follow:



Index 3 is smaller than the current index of pivot, so we must repeat the process of partitioning on the *Smaller* sub-array.

Based on the above observation:

1. develop a recursive algorithm to find the k^{th} smallest entry. The conclusion *1)* represents the base case, conclusions *2)* and *3)* represent recursive calls.
2. implement the recursive `kthItem` method that is already defined in the `KthElement.java`. Your method should call `partition(a, first, last)` method to get the `pivotIndex`
3. run `main` to test your code

See the sample run:

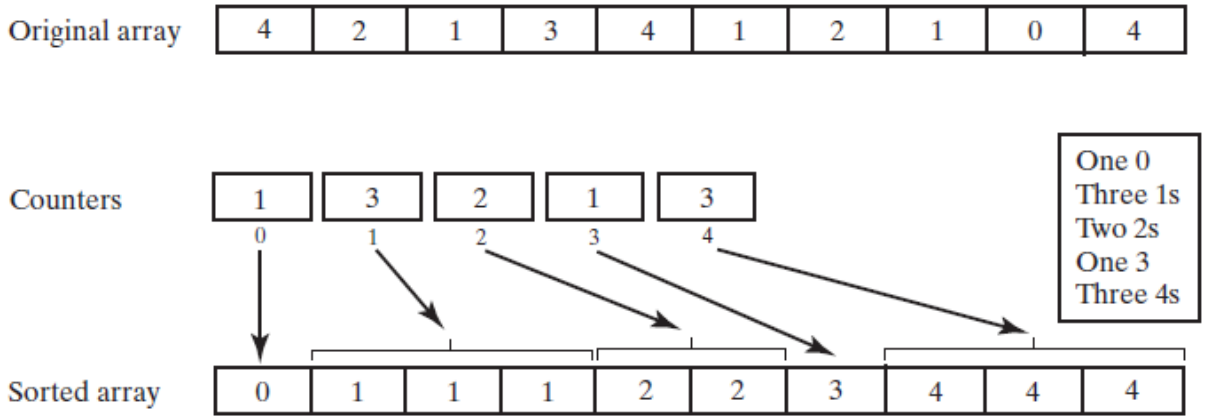
```
What size array should be used?
 It should be an integer value greater than or equal to 1.
21
How many arrays should be used (number of trials)?
 It should be an integer value greater than or equal to 1.
3
What seed should be used?
 It should be an integer value greater than or equal to 1.
11

TRIAL #1
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25]
The original array sorted would be:
[6, 7, 7, 9, 11, 14, 18, 25, 28, 33, 33, 38, 40, 55, 62, 68, 87, 91, 93, 94, 94]
>>> kthItem found median at index 10 with value of 33 <<<
    passes

TRIAL #2
The original array is:
[92, 0, 75, 98, 63, 10, 93, 13, 56, 14, 60, 16, 5, 55, 62, 54, 44, 69, 60, 24, 23]
The original array sorted would be:
[0, 5, 10, 13, 14, 16, 23, 24, 44, 54, 55, 56, 60, 60, 62, 63, 69, 75, 92, 93, 98]
>>> kthItem found median at index 10 with value of 55 <<<
    passes

TRIAL #3
The original array is:
[76, 20, 94, 16, 92, 93, 4, 15, 62, 8, 63, 95, 50, 21, 48, 58, 7, 53, 63, 27, 47]
The original array sorted would be:
[4, 7, 8, 15, 16, 20, 21, 27, 47, 48, 50, 53, 58, 62, 63, 63, 76, 92, 93, 94, 95]
>>> kthItem found median at index 10 with value of 50 <<<
    passes
```

II. A ***counting sort*** is a simple way to sort an array of ***n*** positive integers that lie between ***0*** and ***m***, inclusive. You need ***m+1*** counters. Then, making only one pass through the array, you count the number of times each integer occurs in the array. For example, the picture below shows an array of integers that lie between 0 and 4 and the five counters after the counting sort has made its pass through the array. From the counters, you can see that the array contains one 0, three 1s, two 2s, one 3, and three 4s. These counters enable you to determine that the sorted array should look as follow: `0 1 1 1 2 2 3 4 4 4`



Implement the counting sort as described (please make yourself familiar with the algorithm **before** the lab):

- implement the `countingSort` method that is already defined in `SortArray.java`
- the method has the following signature, where ***a*** represents the array to be sorted, ***maxValue*** is the highest possible ***int*** that can be generated
  `public static void countingSort(int[] a, int maxValue)`
- run `CheckCountingSort.java` to test the method.

See the sample run (please note that seed was not used):

```
What size array should be used?
 It should be an integer value greater than or equal to 1.
21
How many arrays should be used (number of trials)?
 It should be an integer value greater than or equal to 1.
3
What maximum number should be generated?
 It should be an integer value greater than or equal to 1.
9

TRIAL #1
The original array is:
[4, 7, 8, 9, 5, 9, 7, 3, 8, 3, 6, 7, 1, 7, 5, 7, 3, 4, 8, 4, 7]
The original array sorted with countingSort:
```

```
[1, 3, 3, 3, 4, 4, 4, 5, 5, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 9, 9]
  passes

TRIAL #2
The original array is:
[0, 8, 8, 3, 5, 5, 4, 4, 9, 1, 0, 5, 0, 6, 0, 1, 2, 1, 8, 1, 6]
The original array sorted with countingSort:
[0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 4, 4, 5, 5, 5, 6, 6, 8, 8, 8, 9]
  passes

TRIAL #3
The original array is:
[9, 0, 2, 5, 3, 0, 6, 0, 7, 2, 8, 7, 6, 5, 3, 2, 0, 0, 0, 5, 4]
The original array sorted with countingSort:
[0, 0, 0, 0, 0, 0, 2, 2, 2, 3, 3, 4, 5, 5, 5, 6, 6, 7, 7, 8, 9]
  passes
```

III. A **binary radix sort** will sort an array ***a*** of ***n*** integer values based on their binary bits instead of their decimal digits. This sort will need only two buckets. Represent the buckets as a ***2 by n*** array. You should avoid unnecessary work by not copying the contents of the buckets back into the array ***a*** at the end of each pass. Instead just add the values from the second bucket to the end of the first bucket. Implement this algorithm:

- implement the `binaryRadixSort` method that is already defined in `SortArray.java`
  - must make use of methods and constants defined in the `Integer` class like: `Integer.SIZE`, `Integer.numberOfLeadingZeros`, `Integer.rotateLeft`
  - must incorporate the concept of a *bitMask* : http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html
- create a test driver class `CheckBinaryRadixSort.java` to test the method (it should be similar to `CheckCountingSort.java`): in `main` method:
  - prompt the user for three inputs: size of the array, number of trials, and the maximum number to be generated
  - generate an Integer array (use `Random` class to generate random ints between 0 and the maximum value provided by the user to populate the array)
  - print the array content before and after the `binaryRadixSort` method was called
  - create a copy of the generated array and sort the copy with a sort method from `Arrays` class
  - compare the original sorted array with the sorted copy of the array, and print the result: `passes` or `fails`

See sample run (please note that seed was not used):

```
What size array should be used?
 It should be an integer value greater than or equal to 1.
21
How many arrays should be used (number of trials)?
 It should be an integer value greater than or equal to 1.
3
What maximum number should be generated?
 It should be an integer value greater than or equal to 1.
99

TRIAL #1
The original array is:
[43, 23, 27, 42, 43, 98, 85, 95, 85, 20, 16, 30, 40, 45, 85, 33, 45, 85, 39, 57, 51]
The original array sorted with binaryRadixSort:
[16, 20, 23, 27, 30, 33, 39, 40, 42, 43, 43, 45, 45, 51, 57, 85, 85, 85, 85, 95, 98]
  passes

TRIAL #2
The original array is:
[7, 21, 63, 14, 2, 60, 10, 92, 18, 43, 20, 83, 59, 42, 27, 19, 16, 56, 98, 13, 46]
The original array sorted with binaryRadixSort:
[2, 7, 10, 13, 14, 16, 18, 19, 20, 21, 27, 42, 43, 46, 56, 59, 60, 63, 83, 92, 98]
  passes

TRIAL #3
```

```
The original array is:
[4, 84, 24, 84, 6, 48, 24, 18, 53, 83, 50, 53, 20, 24, 6, 49, 92, 66, 99, 23, 38]
The original array sorted with binaryRadixSort:
[4, 6, 6, 18, 20, 23, 24, 24, 24, 38, 48, 49, 50, 53, 53, 66, 83, 84, 84, 92, 99]
  passes
```

IV. The formula: `(last – first + 1)` represents the number of elements the `partition` method is to work with. For arrays that are fewer than 7 (MIN_SIZE) entries, `quickSort` calls `insertionSort` method right away, so `partition` method is not called.

You need to **revise the implementation** of `partition` method in `SortArray.java` as described below:

- *if* we have exactly **7 elements**, choose the current **middle entry** for the pivot
- *else if* we have **between 8 and 40 elements inclusive**, use the ***median-of-three* pivot** selection scheme
- *else* we have more than **40 elements**, the pivot should be the ***median-of-nine*** entries that are about equally spaced, including the **first**, **last**, and **middle** entries. See the median-of-nine pivot selection algorithm below.

  The **median-of-nine** pivot selection algorithm:

  1. first identify the nine indexes
     - must be implemented as a recursive method called `findNineCandidatePivotIndexes`
     - the method's parameter called `indexes` should contain the list of 9 indexes once the method is finished.
  2. sort the calculated `indexes` with `Collections.sort` method. For example, you should see the following content in the `indexes` list:
     a) for array of size **41** the indexes would be: **0**, 5, 10, 15, **20**, 25, 30, 35, **40**
     b) for array of size **50** the indexes would be: **0,** 6, 12, 18, **24**, 30, 36, 42, **49**
     c) for array of size **67** the indexes would be: **0,** 8, 16, 24, **33**, 41, 49, 57, **66**
     d) for array of size **72** the indexes would be: **0**, 8, 17, 26, **35**, 44, 53, 62, **71**
     e) for array of size **100** the indexes would be: **0**, 12, 24, 36, **49**, 61, 74, 86, **99**
  3. after the `indexes` have been sorted:
     - sort **the nine elements** at those indexes using *insertion sort* algorithm (implement and call method `insertionSortNinePivotIndexes`)
     - select the element at the middle index to be the pivot
     - after the pivot selection, the partition algorithm will follow the same flow as with the *median-of-three*

See the sample run from `CheckQuickSort.java` driver:

```
===> TEST array of size 1
The original array is:
[38]
The original array sorted with quickSort:
[38]
  passes

===> TEST array of size 2
The original array is:
[38, 68]
The original array sorted with quickSort:
```

```
[38, 68]
  passes

===> TEST array of size 3
The original array is:
[38, 68, 11]
The original array sorted with quickSort:
[11, 38, 68]
  passes

===> TEST array of size 4
The original array is:
[38, 68, 11, 55]
The original array sorted with quickSort:
[11, 38, 55, 68]
  passes

===> TEST array of size 5
The original array is:
[38, 68, 11, 55, 33]
The original array sorted with quickSort:
[11, 33, 38, 55, 68]
  passes

===> TEST array of size 6
The original array is:
[38, 68, 11, 55, 33, 7]
The original array sorted with quickSort:
[7, 11, 33, 38, 55, 68]
  passes

===> TEST array of size 7
The original array is:
[38, 68, 11, 55, 33, 7, 40]
The original array sorted with quickSort:
[7, 11, 33, 38, 40, 55, 68]
  passes

===> TEST array of size 8
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33]
The original array sorted with quickSort:
[7, 11, 33, 33, 38, 40, 55, 68]
  passes

===> TEST array of size 9
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93]
The original array sorted with quickSort:
[7, 11, 33, 33, 38, 40, 55, 68, 93]
  passes

===> TEST array of size 39
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92]
The original array sorted with quickSort:
[5, 6, 7, 7, 8, 8, 9, 11, 11, 14, 15, 18, 21, 25, 28, 29, 29, 33, 33, 36, 38, 40, 48, 54, 55, 57,
62, 68, 68, 74, 79, 81, 87, 91, 92, 93, 94, 94, 94]
  passes

===> TEST array of size 40
```

```
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32]
The original array sorted with quickSort:
[5, 6, 7, 7, 8, 8, 9, 11, 11, 14, 15, 18, 21, 25, 28, 29, 29, 32, 33, 33, 36, 38, 40, 48, 54, 55,
57, 62, 68, 68, 74, 79, 81, 87, 91, 92, 93, 94, 94, 94]
  passes

===> TEST array of size 41
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32, 42]
The original array sorted with quickSort:
[5, 6, 7, 7, 8, 8, 9, 11, 11, 14, 15, 18, 21, 25, 28, 29, 29, 32, 33, 33, 36, 38, 40, 42, 48, 54,
55, 57, 62, 68, 68, 74, 79, 81, 87, 91, 92, 93, 94, 94, 94]
  passes

===> TEST array of size 50
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32, 42, 23, 14, 67, 96, 0, 44, 94, 29, 38]
The original array sorted with quickSort:
[0, 5, 6, 7, 7, 8, 8, 9, 11, 11, 14, 14, 15, 18, 21, 23, 25, 28, 29, 29, 29, 32, 33, 33, 36, 38,
38, 40, 42, 44, 48, 54, 55, 57, 62, 67, 68, 68, 74, 79, 81, 87, 91, 92, 93, 94, 94, 94, 94, 96]
  passes

===> TEST array of size 67
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32, 42, 23, 14, 67, 96, 0, 44, 94, 29, 38,
85, 16, 30, 4, 11, 19, 9, 36, 69, 5, 73, 77, 65, 72, 73, 74, 46]
The original array sorted with quickSort:
[0, 4, 5, 5, 6, 7, 7, 8, 8, 9, 9, 11, 11, 11, 14, 14, 15, 16, 18, 19, 21, 23, 25, 28, 29, 29, 29,
30, 32, 33, 33, 36, 36, 38, 38, 40, 42, 44, 46, 48, 54, 55, 57, 62, 65, 67, 68, 68, 69, 72, 73,
73, 74, 74, 77, 79, 81, 85, 87, 91, 92, 93, 94, 94, 94, 94, 96]
  passes

===> TEST array of size 72
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32, 42, 23, 14, 67, 96, 0, 44, 94, 29, 38,
85, 16, 30, 4, 11, 19, 9, 36, 69, 5, 73, 77, 65, 72, 73, 74, 46, 41, 7, 37, 43, 72]
The original array sorted with quickSort:
[0, 4, 5, 5, 6, 7, 7, 7, 8, 8, 9, 9, 11, 11, 11, 14, 14, 15, 16, 18, 19, 21, 23, 25, 28, 29, 29,
29, 30, 32, 33, 33, 36, 36, 37, 38, 38, 40, 41, 42, 43, 44, 46, 48, 54, 55, 57, 62, 65, 67, 68,
68, 69, 72, 72, 73, 73, 74, 74, 77, 79, 81, 85, 87, 91, 92, 93, 94, 94, 94, 94, 96]
  passes

===> TEST array of size 100
The original array is:
[38, 68, 11, 55, 33, 7, 40, 33, 93, 7, 14, 94, 87, 9, 62, 18, 94, 91, 28, 6, 25, 5, 81, 8, 57,
48, 21, 94, 54, 29, 8, 11, 36, 15, 79, 68, 74, 29, 92, 32, 42, 23, 14, 67, 96, 0, 44, 94, 29, 38,
85, 16, 30, 4, 11, 19, 9, 36, 69, 5, 73, 77, 65, 72, 73, 74, 46, 41, 7, 37, 43, 72, 79, 92, 47,
18, 58, 50, 94, 1, 54, 4, 81, 41, 67, 78, 69, 12, 5, 76, 32, 55, 13, 66, 73, 41, 66, 27, 63, 66]
The original array sorted with quickSort:
[0, 1, 4, 4, 5, 5, 5, 6, 7, 7, 7, 8, 8, 9, 9, 11, 11, 11, 12, 13, 14, 14, 15, 16, 18, 18, 19, 21,
23, 25, 27, 28, 29, 29, 29, 30, 32, 32, 33, 33, 36, 36, 37, 38, 38, 40, 41, 41, 41, 42, 43, 44,
46, 47, 48, 50, 54, 54, 55, 55, 57, 58, 62, 63, 65, 66, 66, 66, 67, 67, 68, 68, 69, 69, 72, 72,
73, 73, 73, 74, 74, 76, 77, 78, 79, 79, 81, 81, 85, 87, 91, 92, 92, 93, 94, 94, 94, 94, 94, 96]
  passes
```