Faysal Khatri
Project 7 Implementation Notes

sort(array)

I call the provided `quicksort()` method to run the sort.

```
	public static <T extends Comparable<T>>
	void sort(T[] data) {
		quickSort(data);
	}
```

cutoff_qsoft(array, min, max, cutoff)

This method will use a bubble sort once the size of the array reaches n.

```
	public static <T extends Comparable<T>>
	void cutoff_qsort(T[] data) {
		cutoff_qsort(data, 0, data.length-1, 6); // default cutoff=6
	}

	private static <T extends Comparable<T>>
	void cutoff_qsort(T[] data, int min, int max, int cutoff) {
		if (min < max && ((max-min) > cutoff))
		{
			// create partitions
			int indexofpartition = partition(data, min, max);

			// sort the left partition (lower values)
			cutoff_qsort(data, min, indexofpartition - 1, cutoff);

			// sort the right partition (higher values)
			cutoff_qsort(data, indexofpartition + 1, max, cutoff);
		}
		else if (min < max)
		{
			// call bubble sort for arrays of length 5 or less
			bubbleSort(data, min, max);
		}
	}
```

find nth(data, n)

This method runs a modified quicksort on the data array until the chosen partition element is found to be the nth element. On each recursive cycle, quicksort will place the partition element in its final place. If the partition element is greater than n, then the method continues searching on the left side, otherwise it continues on the right side. It is done once the partition element is placed at n.

I first implemented this using a modified selection sort, but I think the quicksort algorithm will run a little faster, especially for large n.

```
public static <T extends Comparable<T>>
T find_nth(T[] data, int n) {
	//selectionSortnth (data, 0, data.length-1, n-1);
	return quickSortNth(data, 0, data.length-1, n-1);
}



private static <T extends Comparable<T>>
T quickSortNth(T[] data, int min, int max, int n) {
	if (min <= max)
	{
		// create partitions
		int indexofpartition = partition(data, min, max);

		if (indexofpartition == n) {
			return data[indexofpartition];
		}
		else if (indexofpartition > n) {
			//keep looking in the left side of the array
			return quickSortNth(data, min, indexofpartition - 1, n);
		}
		else {
			//keep looking in the right side of the array
			return quickSortNth(data, indexofpartition + 1, max, n);
		}
	}
	else {
		return null;
	}
}
```

closest_pair(data)

This method uses a modified bubble sort to compare each element to each other to find the closest pair. I had the idea of using a modified merge sort for this after implementing the bubble algorithm, but I'm too lazy to do it again.

```
public static
List<Integer> closest_pair(Integer[] data) {
      ArrayList<Integer> results = new ArrayList<Integer>();

      int position, scan;
      results.add(data[0]);
      results.add(data[1]);
      int smallestDiff = Math.abs(data[1]-data[0]);

      int min = 0;
      int max = data.length-1;

 // position is the "stopping point" for each pass
 for (position = max; position >= min; position--)
 {
     for (scan = 0; scan < position; scan++)
     {
         if (Math.abs(data[scan+1]-data[scan]) < smallestDiff) {
             smallestDiff = Math.abs(data[scan+1]-data[scan]);
             results.clear();
             results.add(data[scan]);
             results.add(data[scan+1]);
         }
     }
 }
 return results;
}
```

shuffle()

This method in the Deck class uses quicksort to sort an array of random integers and performs the same swaps on the deck of cards. My implementation copies the deck from the arraylist into an array and copies them back after the shuffling. I could write a swap() function for the arraylist and avoid the copying.

```
private static <T extends Comparable<T>>
void quickShuffle(PlayingCard[] deck, Integer[] rand, int min, int max) {
	if (min < max)
	{
		// create partitions
		int indexofpartition = partitionShuffle(deck, rand, min, max);

		// sort the left partition (lower values)
		quickShuffle(deck, rand, min, indexofpartition - 1);

		// sort the right partition (higher values)
		quickShuffle(deck, rand, indexofpartition + 1, max);
	}
}
private static <T extends Comparable<T>>
int partitionShuffle(PlayingCard[] deck, Integer[] rand, int min, int max) {
	Integer partitionelement;
	int left, right;
	int middle = min + ((max - min) / 2);

	// use the middle data value as the partition element
	partitionelement = rand[middle];
	// move it out of the way for now
	swap(rand, middle, min);
	swap(deck, middle, min);

	left = min;
	right = max;

	while (left < right)
	{
		// search for an element that is > the partition element
		while (left < right &&
		less_than_equal(rand[left],partitionelement))
			left++;
		// search for an element that is < the partition element
		while (greater_than(rand[right], partitionelement))
			right--;
		// swap the elements
		if (left < right)
			swap(rand, left, right);
			swap(deck, left, right);
	}
	// move the partition element into place
	swap(rand, min, right);
	swap(deck, min, right);
	return right;
}
```

Sample Output

```
Testing sort():
Before sorting: 721 727 248 318 424 914 497 120 986 993 166 954 892 408 443 155 884
258 278 579 681 656 77 698 737 673 309 226 213 330
After sorting: 77 120 155 166 213 226 248 258 278 309 318 330 408 424 443 497 579 656
673 681 698 721 727 737 884 892 914 954 986 993
The array is sorted!

Testing cutoff_sort():
Before sorting: 620 160 255 537 949 170 925 180 414 28 367 827 229 266 438 59 2 907
550 39 889 939 477 525 339 724 939 190 103 904
After sorting: 2 28 39 59 103 160 170 180 190 229 255 266 339 367 414 438 477 525 537
550 620 724 827 889 904 907 925 939 939 949
The array is sorted!

Testing cutoff_sort():
Before sorting: 738 65 414 903 308 497 836 580 145 503 597 800 407 155 540 327 136
395 811 499 306 133 703 879 234 434 110 792 444 942
After sorting: 65 110 133 136 145 155 234 306 308 327 395 407 414 434 444 497 499 503
540 580 597 703 738 792 800 811 836 879 903 942
The array is sorted!

Testing find_nth():
441 49 35 136 175 179 206 218 412 421 409 265 430 475 125 486 515 517 673 556 591 605
640 534 742 760 746 958 974 990
The array is NOT sorted!
1st: 35
4th: 136
30th: 990

Testing closest_pair():
First test: [4, 5]
Second test: [0, 1]
Third test: [10, 11]

Testing shuffle():
Before sorting: 1H 2H 3H 4H 5H 6H 7H 8H 9H 10H JH QH KH 1S 2S 3S 4S 5S 6S 7S 8S 9S
10S JS QS KS 1D 2D 3D 4D 5D 6D 7D 8D 9D 10D JD QD KD 1C 2C 3C 4C 5C 6C 7C 8C 9C 10C
JC QC KC
After sorting:  8C 9C 8S 2H 1C 7D 8D 2S QC 2D 10S 4S JH 10H 1S KS 6S 4C 6H 8H 6C 4H
6D 5D KC 3D JD 9H KD 10C 3H QH 3S 7S 3C JS 10D 7H 1D 5C JC 7C 5H 2C 1H KH 4D 5S QS 9S
QD 9D
```