QuickSort vs. Selection Sort

1. The Quicksort has been mathematically proven to be the fastest sorting algorthm. The algorthm is written below, as is the Selection Sort.

```
public class quicksorter
{
  private final int MAX = 100000;
   private int[] ar = new int[MAX];

   public void fill()                              // fill with random numbers
     {for (int x=0;x < ar.length; x++)
        ar[x] = (int)(Math.random() * 100000) + 1;
      }

      public void fill2()                        // fill with sorted numbers
     {for (int x=0;x < ar.length; x++)
        ar[x] = x;
      }

   public void quicksort()
     {quicksorthelper(0,ar.length-1);
      }

    public void selectionSort()
     {for (int x=0; x < ar.length; x++)
        {int place = x;
         for (int y=x+1; y < ar.length;y++)
           if (ar[x] < ar[place])
             place = x;

           int temp = ar[x];
           ar[x] = ar[place];
           ar[place] = temp;
        }}

    public void quicksorthelper(int low,int high)
     {int pivotplace;
       if (low >= high)      return;

      int pivotvalue = ar[low];                    //  will switch this to middle later
      int frontofbig = low+1;                      // point to front of numbers > pivot value
      for (int x=frontofbig; x <= high;x++)
       if (ar[x] < pivotvalue)
          {int temp =ar[x];
```

```
            ar[x] = ar[frontofbig];             // keep all small to left of frontofbig
            ar[frontofbig] = temp;
            frontofbig++;
          }

          ar[low] = ar[frontofbig-1];                    // when done move pivot value to pivot place
          ar[frontofbig-1] = pivotvalue;
           pivotplace = frontofbig-1;

                    quicksorthelper(low, pivotplace-1);          // now sort 2 partitions
                     quicksorthelper(pivotplace+1, high);
      }
   }
```

The hardest part of writing the Quicksort is writing the partition algorithm that divides the array into two parts separated by the pivot value.

So if the array was 12 4 18 8 23 9 45 2 and we picked the first number 12 as the pivot value then the array would become 4 8 9 2 12 18 23 45 with the numbers separated around the pivot value. We would then call the quicksort function to sort the two smaller sections, and then the smaller sections, until all sections were down to a size of 1.

2. Below is the call, with a timer to the Quicksort and Selection sorts

```
import java.io.*;

public class SortObjects2
{
  public static void main(String[] args)
  {
quicksorter q = new quicksorter();

q.fill();
long timeA = System.currentTimeMillis();
q.quicksort();
long timeB = System.currentTimeMillis();
System.out.println("Elapsed time with Quicksort Sort: " + (timeB - timeA) + " milliseconds");

q.fill();
timeA = System.currentTimeMillis();
q.selectionSort();
timeB = System.currentTimeMillis();
System.out.println("Elapsed time with Selection Sort: " + (timeB - timeA) + " milliseconds");
  }
}
```

The output was :
Elapsed time with Quicksort Sort: 18 milliseconds
Elapsed time with Selection Sort: 19758 milliseconds

The Quicksort was over 1000 times faster for 100,000 integers.

Run again:
Elapsed time with Quicksort Sort: 22 milliseconds
Elapsed time with Selection Sort: 19494 milliseconds

Over 800 times faster. The times will change because some sets of data will sort quicker than others.

Run a third time we get:
Elapsed time with Quicksort Sort: 23 milliseconds
Elapsed time with Selection Sort: 23374 milliseconds

3. There is a problem with the Quicksort though. If the array is sorted, then the algorithm becomes really slow, becoming almost as slow as the Selection Sort. This is because you aren't separating the data into smaller and smaller partitions. So if we sort 4000 sorted integers (call fill2, instead of fill) you get:

Elapsed time with Quicksort Sort: 27 milliseconds
Elapsed time with Selection Sort: 38 milliseconds

the reason we only sorted 4000 elements is because with more than that we get a stack overflow and the program crashes because the program has so many recursive calls it has to remember. So the Quicksort won't even work for large groups of data.

4. Quicksort fix. To fix the Quicksort bug, we can use something other than the first value as the pivot value. If we pick the middle value of the segment as the middle value it will avoid this problem of working with sorted arrays.

We can keep the partition algorithm by just switching the first value in the segment we are sorting with the middle value in that segment.

```
if (low >= high) return;
   int mid = (low + high) /2;
   int pivotvalue = ar[mid];
   ar[mid] = ar[low];
   ar[low] = pivotvalue;
```

Then the Quicksort functions as expected, even for sorted values.

Elapsed time with Quicksort Sort: 19 milliseconds
Elapsed time with Selection Sort: 19398 milliseconds