David Sabzanov

Feb 28, 2017

COMP 496

Project #1

SORTS CLASS:

```
/**
 * David Sabzanov
 * February 28, 2017
 * Project #1
 *
 * Sorts Class
 */

package sorts;

public class Sorts
{
    private static long MSComparisons = 0L;
    private static long ISComparisons = 0L;
    /**
     * @param args the command line arguments
     */

    /*--------------Insertion Sort ----------------------*/
     public static long insertionsort( int[] a)
    {
        ISComparisons = 0;
        int t = 0;
        int n;
        for(int i = 1 ; i < a.length; i++)
        {
            t = a[i];
            n = i;
            while((n >= 1) && (t < a[n - 1]))
            {
                a[n] = a[n - 1];
                n--;
                ISComparisons++;
            }
            a[n] = t;
        }
        return ISComparisons;
    }

    /* --------------------Merge Sort --------------------*/
    //merges sorted slices a[i.. j] and a[j + 1 …  k] for 0<=  i <=j < k < a.length

    public static long merge ( int[] a,  int i, int j , int k)
    {
        int[] temp = new int[a.length];
```

```
    int lowerIndex = i, midIndexPlusOne = j+1, LI = i;

      while(lowerIndex <= j && midIndexPlusOne <= k)
    {
        if(a[lowerIndex] < a[midIndexPlusOne])
        {
            temp[LI] = a[lowerIndex];
            LI++;
            lowerIndex++;
            MSComparisons++;
        }
        else
        {
            temp[LI] = a[midIndexPlusOne];
            LI++;
            midIndexPlusOne++;
            MSComparisons++;
        }
      }
      if(lowerIndex > j)
    {
        while(midIndexPlusOne <= k)
        {       //left sub array exhausted
            temp[LI] = a[midIndexPlusOne];
            LI++;
            midIndexPlusOne++;
        }
      }
    else
    {
        while(lowerIndex <= j)
        {       //right sub array exhausted
            temp[LI] = a[lowerIndex];
            LI++;
            lowerIndex++;
        }
      }

      //write sorted element in array a
      for(LI = i; LI <= k; LI++)
    {
        a[LI] = temp[LI];
      }

    return MSComparisons;
}
```

```
//sorts  a[ i .. k]  for 0<=i <= k < a.length
private static long mergesort(int[] a, int i , int k)
{
   int mid = (i + k)/2;
   if (i < k)
   {
      mergesort(a, i, mid);
      mergesort(a, mid + 1, k);
      merge(a,i, mid, k);
   }
   return MSComparisons;
}

//Sorts the array using mergesort
public static long mergesort(int[] a)
{
   int i = 0;
   int k = a.length-1;
   mergesort(a, i, k);
   return MSComparisons;
}

public static boolean isSorted(int[] a)
{
   for (int i = 0; i < a.length-1; i++)
   {
      if (a.length == 1)
      {
         return true;
      }
      if (a[i] > a[i+1])
      {
         return false;
      }
   }
   return true;
}

public void setMSComparison(long c) {
   MSComparisons = c;
}

public long getMSComparison() {
   return MSComparisons;
```

```
    }

    public void setISComparison(long c) {
        ISComparisons = c;
    }

    public long getISComparison() {
        return ISComparisons;
    }
}
```

DRIVER CLASS:

```
/**
 * David Sabzanov
 * February 28, 2017
 * Project #1
 *
 * Driver Class
 */
package sorts;

/**
 *
 * @author davidsabzanov
 */
public class Driver {

    private static int[] arraySize = {10,100,1000,10000,100000,200000,300000};
    private static Sorts sort = new Sorts();

    public static void main(String[] args)
    {
        // TODO code application logic here
        //Sorts sort = new Sorts();
        experiment1(); //mergesort
        experiment2(); //insertionsort
    }



    //Driver Methods

    public static void experiment1()
    {
        System.out.println("Merge Sort:");
        System.out.printf("| %8s | %15s | %11s | %14s | %11s |",
                "n", "C(n)", "C(n)/nlogn", "T(n)", "T(n)/nlogn");
        System.out.println();
        for (int i=0; i<arraySize.length; i++)
        {
            long averageTime = 0L;
            long finalTime = 0L;
            long averageComparisons = 0L;
            long sumComparisons = 0L;
            for (int j=0; j<5; j++)
            {
```

```
            int[] n = new int[arraySize[i]];
            for (int k = 0; k<n.length; k++)
            {
               n[k] = (int)(Math.random() * 1000000);
            }
            sort.setMSComparison(0L);
            //MSComparisons = 0L;


            long timeBefore = System.nanoTime();
            Sorts.mergesort(n);
            long timeAfter = System.nanoTime();
            long endTime = timeAfter - timeBefore;
            finalTime = finalTime + endTime;

            sumComparisons = sumComparisons + sort.getMSComparison();
         }
         double nlogn = arraySize[i]*(Math.log(arraySize[i])/Math.log(2));
         averageComparisons = sumComparisons/5;
         averageTime = finalTime/5;
         double avgCompOverN2 = averageComparisons/nlogn;
         double avgTimeOverN2 = averageTime/nlogn;

         System.out.printf("| %8d | %15d | %11.4f | %14d | %11.4f |",
               arraySize[i], averageComparisons, avgCompOverN2, averageTime,
avgTimeOverN2);
         System.out.println();
      }
   }
```

```
   public static void experiment2()
   {
      System.out.println("Insertion Sort:");
      System.out.printf("| %8s | %15s | %11s | %14s | %11s |",
            "n", "C(n)", "C(n)/n^2", "T(n)", "T(n)/n^2");
      System.out.println();
      for (int i=0; i<arraySize.length; i++)
      {
         long averageTime = 0L;
         long finalTime = 0L;
         long averageComparisons = 0L;
```

```
        long sumComparisons = 0L;
        for (int j=0; j<5; j++)
        {
           int[] n = new int[arraySize[i]];
           for (int k = 0; k<n.length; k++)
           {
              n[k] = (int)(Math.random() * 1000000);
           }
           sort.setISComparison(0L);
           //ISComparisons = 0L;


           long timeBefore = System.nanoTime();
           Sorts.insertionsort(n);
           long timeAfter = System.nanoTime();
           long endTime = timeAfter - timeBefore;
           finalTime = finalTime + endTime;

           sumComparisons = sumComparisons + sort.getISComparison();
        }
        double nSquared = arraySize[i]*arraySize[i];
        averageComparisons = sumComparisons/5;
        averageTime = finalTime/5;
        double avgCompOverN2 = averageComparisons/nSquared;
        double avgTimeOverN2 = averageTime/nSquared;
        System.out.printf("| %8d | %15d | %11.4f | %14d | %11.4f |",
              arraySize[i], averageComparisons, avgCompOverN2, averageTime,
avgTimeOverN2);
        System.out.println();
     }
  }
}
```

This project consists of two experiments that compares the efficiency of two sorting algorithms, merge sort and insertion sort, where in each experiment you create arrays of sizes 10, 100, 1000, 10000, 100000, 200000, 300000 that have random integers from 1 to 100000, five times. Each time an array of random integers are created, it is sent out to sort. For the first experiment, the arrays are tested out with the merge sort algorithm and the algorithm is timed from start to finish and we need to return the number of comparisons made to sort the array of each size for five times. For every array size that we go through five times, we average out the number of comparisons, C(n), and the time it took for the merge sort algorithm to complete in nanoseconds, T(n). Once we find the the average number of comparisons and the average time of completion for the merge sort algorithm, we divide those values by the size of the array times the logarithm base 2 of the size of the array, C(n)/nLog(n) and T(n)/nLog(n) respectively. So if the size of the array was 10, we would divide the average number of comparisons and the average time of completion by 10log(10), assuming that the logarithm is base 2. The reason why we divide by nlogn, where n is the size of the array, is because the average case for the merge sort algorithm is nlogn. The calculations of dividing the average number of comparisons made and the average time of completion by nlogn result in the rate each comparison was made and the rate at which the completion time was calculated for each array size. The same thing is done in the second experiment only the algorithm used is insertion sort and instead of nlogn, n^2 is used and for the same reason. After running both experiments, the output presents two tables of data; one for each experiment.

Output:

Merge Sort:

n	C(n)	C(n)/nlogn	T(n)	T(n)/nlogn
10	23	0.6924	12604	379.4182
100	540	0.8128	181297	272.8792
1000	8696	0.8726	3696172	370.8862
10000	120458	0.9065	149891600	1128.0467
100000	1536316	0.9250	8489579240	5111.2360
200000	3272724	0.9292	34168042839	9701.5163
300000	5084659	0.9315	76680905915	14048.2878

Insertion Sort:

n	C(n)	C(n)/n^2	T(n)	T(n)/n^2
10	22	0.2200	6112	61.1200
100	2459	0.2459	201864	20.1864
1000	247983	0.2480	2321801	2.3218
10000	25070428	0.2507	26450559	0.2645
100000	2495080596	1.7695	2857909828	2.0268
200000	10019488534	7.4478	10880402009	8.0877
300000	22484338384	-115.7118	24213177794	-124.6090

By looking at the merge sort data of experiment 1 versus the insertion sort data of experiment 2, we can see how the average comparisons starts at about the same number yet it increases faster, somewhat exponentially, during the insertion sort experiment than the merge sort experiment as “n” increases. From this output we can see how at certain n’s the time T(n) is faster in merge sort than in the insertion sort algorithm and vice-versa. For example in T(10), insertion sort is faster than merge sort. But in T(100), merge sort is faster than insertion sort. We can see this change again when n = 1000 where T(1000) for merge sort is 3,696,172 which is greater than T(1000) for insertion sort which is 2,321,801. After that, from n = 10000 to 300000, it seems like merge sort is much slower than insertion sort.

Looking at the values for C(n)/nLog(n) in the merge sort chart, we can see that the values are in a stable increase. Whereas looking at the values for $C(n)/n^2$ in the insertion sort chart, we can see a sharper increase in values. This shows the rate of which the number of comparisons are made for each sorting algorithm at each “n” size. This is also evident when looking through the C(n) column.

Similarly, when looking at the values for T(n)/nLog(n) in the merge sort chart, we don’t really see a pattern. It’s different from the C(n)/nLog(n) values. The first value at n = 10 (379.4182) is greater than at n = 100 (272.8792). But n = 100 (272.8792) happens to be less than n = 1000 (370.8862). So we can see this small dip in values that may mean that rate of completion of the merge sort algorithm at n = 100 was quick than when n = 10 or when n = 1000. In the case of the insertion sort algorithm, the dip occurs when n = 10000.

There may be several reasons as to why this might be happening. We know that normally the merge sort algorithm is Big-Oh of nLog(n) which is better than the insertion sort algorithm which is Big-Oh of n^2. But in reality there may be other factors that may affect this change. Since this is run on a personal computer, there may be other processes that run in the background at different points of time while the sorting algorithm is running and therefore changes the speed of time of the completion of the sorting algorithm. There may simply be an issue with inefficient memory allocation. There may also be several values in the array that are already somewhat sorted and therefore require less time for the algorithm to sort through.