PROJECT #1

COMP 482

Hrag Ayvazian

9/26/17

Sorts.java

```java
/**
 * Hrag Ayvazian
 * 9/26/17
 * Project #1
 */
public class Sorts {

    //Returns true only if a is sorted from smallest to largest values

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

    /* --------------------MergeSort ----------------------------------------*/

    //Merges sorted slices a[i.. j] and a[j + 1 ... k] for 0<= i <=j < k < a.length
    public static void merge (int A[], int i, int m, int j) {
        int left = m - i + 1;

        int right = j - m;

        int leftarr[] = new int[left];

        int rightarr[] = new int[right];

        for(int b = 0; b < left; b++) {
            leftarr[b] = A[i + b];
        }

        for(int z = 0; z < right; z++) {
            rightarr[z] = A[m + 1 + z];
        }

        int x = 0;
        int y = 0;
        int c = i;
        while(x < left && y < right) {
            if(leftarr[x] <= rightarr[y]) {
                A[c] = leftarr[x];
                x++;
            }
            else {
                A[c] = rightarr[y];
                y++;
            }
            c++;
        }

        while(x < left) {
            A[c] = leftarr[x];
            x++;
            c++;
        }

        while(y < right) {
            A[c] = rightarr[y];
            y++;
            c++;
        }
    }

    //Sorts a[ i .. k] for 0<=i <= k < a.length
    public static void mergeSort (int A[], int i, int j) {
        if(i<j) {
            int m = (i+j)/2;
            mergeSort(A, i, m);
            mergeSort(A,m+1, j);
            merge(A, i, m, j);
        }
    }

    //Sorts the array a using mergesort
    public static void mergeSort (int A[]) {
        int i=0;
        int j = A.length-1;
        mergeSort(A,0,j);
    }

    /* ---------- QuickSort ---------------------------------------------- */

    //Implements in-place partition from text. Partitions subarray s[a..b] into
s[a..l-1] and s[l+1..b]
    // so that all elements of s[a..l-1] are less than each element in s[l+1..b]
    public static int partition (int A[], int i, int j) {
        int par = A[j];
        int l = i;
        int r = j-1;
        int temp;
        while(l <= r) {

            while(l <= r && A[l] <= par) {
                l++;
            }

            while(r >= l && A[r] >= par) {
                r--;
            }

            if(l < r) {
                temp = A[l];
                A[l] = A[r];
                A[r] = temp;

            }
        }
        temp = A[l];
        A[l] = A[j];
        A[j] = temp;
        return l;
    }

    //Quick sorts subarray a[i..j]
    public static void quickSort (int A[], int i, int j) {
        if(i < j) {
            int s = partition(A, i, j);
            quickSort(A, i, s-1);
            quickSort(A, s+1, j);
        }
    }

    //Quick sorts array a
    public static void quickSort( int[] A) {
        int i=0;
        int j = A.length-1;
        quickSort(A,0,j);
    }
}
```

SortTester.java

```
/**
 * Hrag Ayvazian
 * 9/26/17
 * Project #1
 */
public class SortTester {
    public static void main(String[] args) {
        int arra[] = {34,67,23,19,122,300,2,5,17,18,5,4,3,19,-40,23};
        int arra2[] = {34,67,23,19,122,300,2,5,17,18,5,4,3,19,-40,23};
        System.out.print("Unsorted Array: ");
        for(int x =0; x < arra.length; x++) // Print's Unsorted Array
        {
            System.out.print(arra[x]);
            if(x < arra.length-1) {
                System.out.print(",");
            }
        }

        Sorts tests = new Sorts();
        System.out.println("\nIs the array Sorted (true/false)? " +
tests.isSorted(arra)); // Checks to see if array is already sorted


/////////////////////////////////////////////////////////////////////////////////////
        //Merge Sort
        System.out.println("\nMerge Sort");
        long star = System.nanoTime(); // Start timer
        //tests.mergeSort(arra,0,arra.length-1); // Run(Call) Merge Sort
        tests.mergeSort(arra); // Run(Call) Merge Sort
        long en = System.nanoTime(); // End timer
        System.out.print("Sorted array: ");
        for(int x = 0; x < arra.length; x++) // Print's Sorted Array
        {
            System.out.print(arra[x]);
            if(x < arra.length-1) {
                System.out.print(",");
            }
        }
        long tota = en - star; // Total time taken to sort
        System.out.println("\nTime: " + tota);


/////////////////////////////////////////////////////////////////////////////////////
        //Quick Sort
        System.out.println("\nQuick Sort");
        long star2 = System.nanoTime(); // Start timer
        //tests.quickSort(arra2,0, arra2.length-1); // Run(Call) Quick Sort
        tests.quickSort(arra2); // Run(Call) Quick Sort
        long en2 = System.nanoTime(); // End timer
        System.out.print("Sorted array: ");
        for(int x =0; x < arra2.length; x++) // Print's Sorted Array
        {
            System.out.print(arra2[x]);
            if(x < arra.length-1) {
                System.out.print(",");
            }
        }
        long tota2 = en2 - star2; // Total time taken to sort
        System.out.println("\nTime: " + tota2);



////////////////////////////////////////////////////////////////////////////////////
        //Table Format
        //Experiment #1 & Experiment #2

        System.out.println("\nInfo for Table");

        int mergewon = 0;
        int quickwon = 0;
        int n = 10;
        long mergemean = 0;
        long quickmean = 0;

        for(int i = 0; i < 7; i++) {
            int[] array = new int[n+1];
            int[] array2 = new int[n+1];
            for (int ntrial = 0; ntrial < 20; ntrial++) { //Runs 20 different trials
                for (int a = 0; a < n; a++) { //Makes array n length
                    array[a] = (int) (Math.random() * 1000000 + 1);
                    array2[a] = array[a];
                }

                Sorts test = new Sorts();
                //Merge Sort
                long start = System.nanoTime(); // Start timer
                test.mergeSort(array); // Run(Call) Merge Sort
                long end = System.nanoTime(); // Run(Call) Merge Sort
                long total = end - start; // Total time taken to sort
                mergemean = mergemean + total; // Adds all the total time for all 20
trials to calculate merge


                //Quick Sort
                long start2 = System.nanoTime(); // Start timer
                test.quickSort(array2,0, array2.length-1); // Run(Call) Quick Sort
                long end2 = System.nanoTime(); // Run(Call) Merge Sort
                long total2 = end2 - start2; // Total time taken to sort
                quickmean = quickmean + total2;

                if(total <= total2) { //If Merge Sort < Quick Sort. Then Merge Sort is
faster
                    mergewon++;
                }

                else if(total > total2) { //If Merge Sort > Quick Sort. Then Quick
Sort is faster
                    quickwon++;
                }
            }

            System.out.println(n + "    " + "20    Merge Won:" + mergewon + "    Quick
Won:" + quickwon);

            mergemean = mergemean / 20;
            quickmean = quickmean / 20;

            System.out.println("Merge mean " + mergemean);
            System.out.println("Mean/nLog2(n) " + mergemean / (n * (Math.log(n) /
Math.log(2))));
            System.out.println("Quick mean " + quickmean);
            System.out.println("Mean/nLog2(n) " + quickmean/ (n * (Math.log(n) /
Math.log(2))) + "\n");

            mergemean = 0;
            quickmean = 0;
            mergewon = 0;
            quickwon = 0;

            if(i < 5) {
                n = n * 10;
            }

            else {
                n = n * 2;
            }
        }
    }
}
```

Output of array {34,67,23,19,122,300,2,5,17,18,5,4,3,19,-40,23}

```
Unsorted Array: 34,67,23,19,122,300,2,5,17,18,5,4,3,19,-40,23
Is the array Sorted (true/false)? false

Merge Sort
Sorted array: -40,2,3,4,5,5,17,18,19,19,23,23,34,67,122,300
Time: 38690

Quick Sort
Sorted array: -40,2,3,4,5,5,17,18,19,19,23,23,34,67,122,300
Time: 32040
```

Output Information for Report In A Table Format

Info for Table

```
10    20    Merge Won:3    Quick Won:17

Merge mean 65485

Mean/nLog2(n) 1971.2949266055807

Quick mean 8716

Mean/nLog2(n) 262.377744220726


100    20    Merge Won:5    Quick Won:15

Merge mean 69169

Mean/nLog2(n) 104.10971885040956

Quick mean 46989

Mean/nLog2(n) 70.72549233127405


1000    20    Merge Won:0    Quick Won:20

Merge mean 676290

Mean/nLog2(n) 67.86119192253128

Quick mean 137557

Mean/nLog2(n) 13.802927704516753


10000    20    Merge Won:0    Quick Won:20

Merge mean 4596795
```

```
Mean/nLog2(n) 34.59432947295526

Quick mean 1671150

Mean/nLog2(n) 12.576656931346553


100000    20    Merge Won:0    Quick Won:20

Merge mean 27540748

Mean/nLog2(n) 16.581182502045596

Quick mean 15747597

Mean/nLog2(n) 9.480998113256247


1000000    20    Merge Won:0    Quick Won:20

Merge mean 252542529

Mean/nLog2(n) 12.670479401640142

Quick mean 149597526

Mean/nLog2(n) 7.505557100520386


2000000    20    Merge Won:0    Quick Won:20

Merge mean 447351147

Mean/nLog2(n) 10.686039737499803

Quick mean 294703729

Mean/nLog2(n) 7.039695281888643
```

Report

For this experiment/project we are comparing and analyze the differences between the mergesort and quicksort algorithms. We know that mergesort and quicksort should have a runtime of O(nlog(n)) for the average case. However, before we can do this we have to create the mergesort and quicksort algorithm and while each algorithm requires their own different methods to work properly to sort an unsorted array. Both quicksort and mergesort have their own helper function to send additional information like the start point and array length to another mergesort and quicksort algorithm. Both helper functions of each sorting algorithm are about the same with the exception of each calling their own sorting method. Furthermore, now we can understand how both the mergesort and quicksort algorithms work. Mergesort works by dividing an unsorted array into n sublists with each containing one element and it repeatedly merges the sublists to produce a new sorted sublist until there is only one sublists remaining. While quicksort divides the array into two smaller sub-arrays by creating the low elements and the high elements. Then you have to pick a pivot from the array and partition the array. By picking a pivot and partitioning the array you are essentially reordering the array so that numbers smaller than the pivot come before the pivot and number larger than the pivot go after. Lastly, for quicksort you have to recursively apply it to the sub-array of elements with small numbers and a sub-array of elements with big numbers.

n	nTrials	# mergeSort Wins	#quickSort Wins
10	20	3	17
100	20	5	15
1000	20	0	20
10000	20	0	20
100000	20	0	20
1000000	20	0	20
2000000	20	0	20

Taking a look at the table above we notice a n is the size of the array and as the table progresses downwards the size of the array gets bigger. Also, that we are running each size of the array 20 times and timing it to see who has the faster sorting time. To see how many, time mergesort wins and how many times quicksort wins. Taking a closer look at how many times mergesort and quicksort wins we can notice that quicksort has the most wins on all sizes of arrays. However, out of the 20 trials we can see that mergesort also has a decent amount of wins at n=100 but quicksort still wins because it has the most amount of wins out of the 20 trial runs. I calculated the amount of times each sorting algorithm won by starting a time before calling each sorting algorithms and stopping the timer after it completed sorting it. Then by comparing each time you can conclude which sorting algorithms is faster and keep a counter to keep track of which algorithm won.

n	mergeSort: mean runtime(nanosecs)	mergeSort: mean runtime/ (n*log$_2$(n))	quickSort: mean runtime (nanosecs)	quickSort: mean runtime/ (n*log$_2$(n)
10	65485	1971.2949266055807	8716	262.377744220726
100	69169	104.10971885040956	46989	70.72549233127405
1000	676290	67.86119192253128	137557	13.802927704516753
10000	4596795	34.59432947295526	1671150	12.576656931346553
100000	27540748	16.581182502045596	15747597	9.480998113256247
1000000	252542529	12.670479401640142	149597526	7.505557100520386
2000000	447351147	10.686039737499803	294703729	7.039695281888643

Looking at the table above we calculated the mean runtime for both quicksort and mergesort and also the mean runtime divided by (n*log2(n)). To calculate the mean runtime for both mergesort and quicksort I added up all the total time for all 20 trials into a variable called mergemean and quickmean to add up all of the times and then to divide it by 20 because it's running through each size of the array 20 times. Also, looking at the mean runtime in nano seconds we can notice that both quicksort and mergesort increase in time as the size of the array gets bigger. However, comparing both mergesort and quicksort we can see that mergesort has a faster mean runtime then quicksort for the small size arrays. But when the array size gets bigger the mean runtime for mergesort slows down.

Furthermore, now looking at the mean runtime divided by (n*log2(n)) for both quicksort and mergesort. We notice that both of their times are large when it is a smaller array but when the array size gets bigger the time decreases. This is because when you divide a value of nlog(n) by nlog(n) you get the constant time and in our case as the array size gets bigger the constant decreases which gets it close to 0.