Joseph Barbati
CS323
HW1

HybridSortBarbati Mechanism/Algorithm

Before I get into the results, let me talk about my algorithm for sorting and merging. To sort, I take each 1D array and sort it individually based on a 'best guess' way to sort it. I did this by creating a couple helper methods, one called `findBestSort()`. This method takes the 1D array and gets 3 random elements, each from a different third of the array, and compares them. The logic was if the element in the lowest index was greater than the element at the middle index and the element at the highest index was lower than the middle index, the array is probably random. Another set of logic was if the element in the lowest index was lower than the middle and the middle was greater than the greatest, it's probably random. Then I return a number to the sort method that calls QuickSort on that particular array. The way I find these indices is with pseudorandom number generation in a method called `getRandomIndex()`.

Merging the arrays was relatively easy. 2D arrays are just 1D arrays of arrays; it seems obvious but it's the key to my algorithm. This algorithm is virtually the same as regular merge called in merge sort except it merges arrays instead of indices. Regular merge sort gets its recursive nature from the sorting part, so I had to add a recursion to the merge method to make it work like regular merge sort. My merge method takes an array, a begin index and an end index. It uses the indices to calculate a middle index and then checks the difference between the begin and end indices. If the difference is greater than 1, that means we will try and merge more than 2 arrays, which will not work, so we recurse and split the input array even more by calling merge for begin index and middle index, then from middle index (+1) to end index. When the difference between the parameter variables `beginIndex` and `endIndex` is 1 or 0 (meaning we will merge 2 or 1 arrays, respectively), we merge the arrays. If the difference is 0, the merge is easy (because the 1D array has already been sorted before the merge function is called) so we just return the array. If the difference is 1, we merge the arrays much like the merge step in MergeSort, then return the array. Once the final array is returned, it is merged and sorted.

Joseph Barbati
CS323
HW1

Time Comparison between HybridSortChoi and HybridSortBarbati

As I ran tests, I printed some times to the console. I noted that HybridSortChoi merged the array very quickly (between 0 and 1 ms) and sorted some-what quickly (between 3 and 4 ms). I also noted that HybridSortBarbati sorted much faster than Choi, only taking 1 to 2 ms, but my merge was as slow as his sort, taking 3 to 4 milliseconds to merge (all the above times were reference to sorting and merging each individual array created in `testSpeed()` in the unit test). Based on these numbers I knew each HybridSort would take essentially the same time so I ran 3000 tests to see if one edged the other out.

Hybrid Sort	
Choi	Barbati
Time (ms)	Time (ms)
255.124	253.019

Figure 1a. Times based on an average of 3000 calls to `testSpeed()` in the class `HybridSortTest.java`

The figure above shows that my algorithm edged out Choi's baseline by nanoseconds each call to merge and sort for a total average difference of about 2 ms per call to `testSpeed()`. Since the difference is so small when comparing time in milliseconds, I ran anther 3000 tests using nanoseconds (`System.nanoTime()`).

Hybrid Sort	
Choi	Barbati
Time (ns)	Time (ns)
270020396.188	270375172.6353

Figure 1b. Times based on averaged values over 3000 calls to `testSpeed()` after editing the code to get the time in nanoseconds.

Running tests showed that Choi's merge takes between 1/10 and 2/10 of a millisecond while my merge takes around 1 to 1.2 milliseconds. That difference is made up in the sort, however; Choi's sort takes between 2.5 and 3 milliseconds while my sort takes around 1.8 to 2.2 milliseconds per array in a call to `testSpeed()`. According the next 3000 tests with nanoseconds Choi's algorithm

was an average of 0.37 milliseconds faster each call to `testSpeed()`. This proves exactly how close the two algorithms are when it comes to actual time of performance. I think it's possible to get my algorithm down to be faster the HybridSortChoi, if all the object creation at each call to `merge()` could be mitigated or taken away completely. I could not figure out how to do this, however.

In conclusion, HybridSortChoi and HybridSortBarbati are virtually identical when it comes to performance. Barbati's merge performance is O(NlogN) where N is the number of arrays in the input array and the sort performane is B * complexity of sort, where B is the number of elements in the array.