1. (10 Points) Trace bottom-up merge sort (see class notes for implementation details) with the following input, where merge result with sz=k should contain the result of merging adjacent sub-arrays each with size of k. The last row should contain the final sorted result.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Input | 342 | 773 | 429 | 966 | 19 | 952 | 711 | 133 | 684 | 143 |
| Merge  Result  sz = 1 | 342 | 773 | 429 | 966 | 19 | 952 | 711 | 133 | 684 | 143 |
| Merge  Result  sz = 2 | 342 | 773 | 429 | 966 | 19 | 952 | 133 | 711 | 143 | 684 |
| Merge  Result  sz = 4 | 342 | 429 | 773 | 966 | 19 | 133 | 711 | 952 | 143 | 684 |
| Merge  Result  sz = 8 | 19 | 133 | 342 | 429 | 711 | 773 | 952 | 966 | 143 | 684 |
| Merge Result  sz=10 | 19 | 133 | 143 | 342 | 429 | 684 | 711 | 773 | 952 | 966 |

2. (5 Points) Does the merge method produce the proper output if two input subarrays are not in sorted order? Prove your answer or provide a counter example.

*If two input sub arrays (each of size 4, for example) are not in sorted order, a merge sort will NOT produce the proper output. Merge sorts start at the first element of each array and compare their size to each other, incrementing whichever array had the smaller number in that location and inserting the smaller element into a new array. If the sub arrays (of any size > 1) are not in sorted order, the merge sort will not properly sort the resulting array because the unsorted array cannot become sorted before eventually being inserted into the new array.*

3. (5 Points) Give the sequence of subarray sizes in top-down merge sort for N=10.

10 🡪 5|5 🡪 2|3|3|2 🡪 1|1|1|2|2|1|1|1 🡪 1|1|1|1|1|1|1|1|1|1 🡪

2|2|2|2|2 🡪 4|4|2 🡪 8|2 🡪 10

4. (5 Points) The following is another merge sort top down implementation, what is the running time and space complexity for this implementation in big-O? Briefly explain your answer.

**public** **static** <T **extends** Comparable<T>> **void** sort2(T[] a) {

*sort2*(a, 0, a.length - 1);

}

@SuppressWarnings("unchecked")

**private** **static** <T **extends** Comparable<T>> **void** sort2(T[] a, **int** lo, **int** hi) {

**if** (hi <= lo) **return**;

T[] aux = (T[]) **new** Comparable[a.length];

**int** mid = (lo + hi) / 2;

*sort2*(a, lo, mid); // Sort left half

*sort2*(a, mid + 1, hi); // Sort right half

*merge*(a, lo, mid, hi, aux);

}

*Space complexity for this merge sort method is O[n], and running time is O[n\*log(n)]. For each merge sort, the array size is divided in half for each sub array to be sorted, producing log(n) runtime. However, many of these sub arrays (all which aren't either already sorted nor of size 1) will require two sort function calls. Since we are guaranteed to have some number of arrays which are of size 1, n\*log(n) run time can be assumed even under worst-case analysis. Since each sort function call requires a new array to be declared, space complexity is linear, as for array's size requires n+n\*log(n) sub arrays, which reduces to n, or linear space complexity.*

5. (5 Points) Show the partition result of this input array.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Input | 342 | 773 | 429 | 966 | 19 | 952 | 711 | 133 | 684 | 143 |
| Partition  Result | 19 | 143 | 133 | 342 | 966 | 952 | 711 | 429 | 684 | 773 |

6. (10 Points) Given 10 integers from 1 to 10, provide 5 permutations that each of them results in worst case running time for quick sort, which means each call to partition method can only reduce the array size by 1.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 10 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

7. (5 Points) Suppose that we scan over items with keys equal to the partitioning item’s key instead of stopping the scans when we encounter them. What is the running time for this implementation given input with all equal keys?

*Worst case, as each item must be used as a pivot still when going through the quick sort method and swapped with all other equal keys. This results in n2 run time.*

*This question might also be proposing we swap every element with equal keys altogether, in which case we get best case run time of n\*log(n).*

*I can't tell which situation is being proposed.*

8. (5 Points) QuickSortRandomized uses two methods to do quick sort. The first method shuffles the input array before doing regular quick sort. The second method randomly picks a pivot element during each partition. Run QuickSortRandomizedTest five times and provide the results in the following table. From your experiments, which method runs faster?

|  |  |  |
| --- | --- | --- |
| N | Average Method1 Duration | Average Method 2 Duration |
| 1,000 | *787,159* | *214,893* |
| 10,000 | *2,588,804* | *1,817,298* |
| 100,000 | *47,680,244* | *12,551,829* |
| 1,000,000 | *347,002,096* | *201,163,375* |
| 10,000,000 | *5,449,632,210* | *3,394,964,523* |

9. (10 Points) Implement a non-recursive version of quicksort using a stack (java.util.Stack, https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html). Provide implementation inside QuickSortNonRecursive.sort method.

**Submission Note**

1) For written part of the questions:

1. Write your answers inside a text document (in plain text, MS Word, or PDF format)
2. Name the file as firstname.lastname.assignment2.txt(doc, docx, or pdf) with proper file extension

2) For programming part of the questions

1. Use JDK 1.8 and Junit5
2. Put your full name at the beginning of every source file you created or modified. **2 points will be deducted if your names are not included in the source files.**
3. Do not change the provided package, class, or method name. You can add extra classes or methods if they are needed.
4. **If your code does not compile, you will get zero point**.
5. Use the provided tests to verify your implementation. **Extra tests might be used for grading.**
6. Zip all the source files into firstname.lastname.assignment2.zip

3) Submit both of your files (text document and zip file) via Canvas course web site.

2) Due Sep 30th, 11:59 PM