Homework 02 *COSC 311: Algorithms*, Fall 2022

Due: Friday, 09/30/2022 at 11:59 pm

Exercise 1. In class we examined MergeSort and QuickSort as two divide-and-conquer sorting algorithms. We showed that the worst-case running time of MergeSort is $O(n \log n)$, while the worst-case running time of QuickSort is $O(n^2)$. Nonetheless, we argued that if the pivots in the QuickSort algorithm are chosen randomly, then the average running time is $O(n \log n)$. For this exercise, you should investigate the empirical running times of these two sorting methods on large arrays. You may use either provided implementations (Sorting.java, SortTester.java) or you may implement your own versions of these methods in your language of choice.

1. For a range of large array sizes (say, between $10,000$ and $1,000,000$) plot the running times of MergeSort and *QuickSort* of random arrrays of integers to compare the efficiency of the two methods. Which tends to be faster? Do you find your results surprising? How can you explain this difference in running time?

2. As we saw in the first assignment, for sorting small inputs, iterative algorithms such as InsertionSort might be significantly faster than recursive procedures such as MergeSort. We can exploit this behavior to speed up recursive procedures like MergeSort and QuickSort by modifying the base case: when sorting a sufficiently small (portion of an) array, rather than making a recursive call, we can invoke the procedure that is faster for small arrays. Modify your MergeSort and QuickSort implementations in this way to try to make them as fast as possible for large imputs (say, size $1,000,000$). How much of a performance improvement do you see for each algorithm? What parameters (i.e., base case size) give you the best performance? (Note, you will have to modify your "base-case" algorithm so that it only sorts an array in a given range of indices, rather than the entire array.)

Exercise 2. In class and in the last assignment, we have considered five sorting algorithms: SelectionSort, BubbleSort, InsertionSort, MergeSort, and QuickSort. Suppose your task is to sort an array a of size n that is already *almost* sorted in the sense that every element in a is stored at an index that is close to its index after a is sorted. Quantitatively, there is a small number k (e.g., $k = 5$) such that the following holds: if a is the original array and s is the array after sorting, then for every value $v = a[i]$, we have $v = s[j]$ with $\mid i - j \mid \leq k$. That is, sorting a does not move any element more than k indices away from its original position. In this scenario, what sorting algorithm would you expect to be most efficient, and why?

Challenge. For your chosen algorithm, derive a bound on the number operations it performs as a function of both n and k.

Exercise 3. In class we proved that any sorting algorithm that relies on comparisons of the form compare(a, i, j) to sort requires $\Omega(n \log n)$ comparisons to sort arrays of size n. On the other hand, we saw that when an array's values are represented by B bits each, the algorithm RadixSort sorts an array of size n using $O(Bn)$ elementary operations. In languages like Java and C/C++, ‘int‘ values are represented by $B = 32$ bits. Thus, RadixSort uses $O(32n) = O(n)$ elementary operations. Why does the $O(n)$ running time of

RadixSort in this scenario *not* contradict the lower bound of $\Omega(n \log n)$ that we proved for all sorting algorithms?

Hint. Consider the simpler case of sorting values represented by $B = 1$ bits. That is, all values are either 0 or 1. Does the $\Omega(n \log n)$ lower bound apply to this case? Why or why not?

Exercise 4. Suppose a is a sorted array of n distinct integer values. That is, $a = [a_1, a_2, \ldots, a_n]$ with $a_1 < a_2 < \ldots < a_n$. (Note that some values may be negative.) We say that a value a_i is a *fixed point* if $a_i = i$.

1. Devise an algorithm that finds a fixed point of a if one exists, or (correctly) reports that so fixed point exists that runs in time $O(\log n)$ time. (*Hint.* Since the a_i are sorted distinct integers, we have that for all indices i, j with $j \geq i$, $a_j - a_i \geq j - i$.)
2. Argue that your algorithm does indeed have running time $O(\log n)$.
3. Suppose the elements of a are *not* assumed to be distinct (but are still sorted integer values). Does your algorithm still work in this case? Why or why not? Is it possible to divise an algorithm with running time $O(\log n)$ in this case?