1 Identifying Sorts

Below you will find intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm it illustrates from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), and heapsort. When we split an odd length array in half in mergesort, assume the larger half is on the right.

Input list: 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

(a) 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000
1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392
192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001

(b) 1337, 192, 594, 129, 1000, 1429, 3291, 7683, 4242, 9001, 4392
192, 594, 129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392
129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

(c) 1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000
192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000
192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

(d) 1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192
7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001
129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

In all these cases, the final step of the algorithm will be this: 129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

2 Conceptual Sorts

Answer the following questions regarding various sorting algorithms that we've discussed in class. If the question is T/F and the statement is true, provide an explanation. If the statement is false, provide a counterexample.

(a) We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

(b) Give a 5 integer array that elicits the worst case runtime for insertion sort.

(c) (T/F) Heapsort is stable.

(d) Compare mergesort and quicksort in terms of (1) runtime, (2) stability, and (3) memory efficiency for sorting linked lists.

(e) You will be given an answer bank, each item of which may be used multiple times. You may not need to use every answer, and each statement may have more than one answer.

(A) Quicksort (in-place using Hoare partitioning and choose the leftmost item as the pivot)
(B) Merge Sort
(C) Selection Sort
(D) Insertion Sort
(E) Heapsort
(F) None of the above

For each of the statements below, list all letters that apply. Each option may be used multiple times or not at all. Note that all answers refer to the entire sorting process, not a single step of the sorting process, and assume that N indicates the number of elements being sorted.

______________________ Bounded by $\Omega(N \log N)$ lower bound.

______________________ Worst case runtime that is asymptotically better than quicksort's worst case runtime.

______________________ In the worst case, performs $\Theta(N)$ pairwise swaps of elements.

______________________ Never compares the same two elements twice.

______________________ Runs in best case $\Theta(\log N)$ time for certain inputs.

3 Bears and Beds

In this problem, we will see how we can sort "pairs" of things without sorting out each individual entry. The hot new Cal startup AirBearsnBeds has hired you to create an algorithm to help them place their bear customers in the best possible beds to improve their experience. Now, a little known fact about bears is that they are very, very picky about their bed sizes: they do not like their beds too big or too little - they like them just right. Bears are also sensitive creatures who don't like being compared to other bears, but they are perfectly fine with trying out beds.

The Problem:

- **Inputs:**
  - A list of `Bears` with unique but unknown sizes
  - A list of `Beds` with unique but unknown sizes
  - *Note: these two lists are not necessarily in the same order*
- **Output:** a list of `Bears` and a list of `Beds` such that the `i`th `Bear` is the same size as the `i`th `Bed`
- **Constraints:**
  - `Bears` can only be compared to `Beds` and we can get feedback on if the `Bear` is too large, too small, or just right for it.
  - Your algorithm should run in $O(N \log N)$ time on average