CS 61B — Sorting

Spring 2021 — Exam Prep Discussion 12: April 12, 2021

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, 192, 1337, 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

mergesort

(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

quicksort

(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

insertion sort

(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

heapsort

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.

N^2

(a) (T/F) Quicksort has a worst case runtime of $\Theta(NlogN)$, where N is the number of elements in the list ~~that we're~~ sorting.

False, quicksort has a worst case runtime of $\theta(N^2)$, if the array is partitioned very unevenly at each iteration.

(b) 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?

The input is small or the array is nearly sorted. Note that insertion sort has a best case runtime of $\theta(N)$, which is when the array is already sorted.

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

5, 4, 3, 2, 1.

Explanation is redundant.

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

False, stability for sorting algorithm means that if two elements in the list are defined to be equal, then they will retain their relative ordering after the sort is complete. Heap operations may mess up the relative ordering of equal items and thus is not stable. 21, 20a, 20b, 12, 11, 8, 7

(e) Give some reasons as to why someone would use mergesort over quicksort.

Mergesort has $\theta(N \log N)$ worst case runtime versus quicksort's $\theta(N^2)$. Mergesort is stable, where quicksort typically isn't. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end. Mergesort is also preferred for sorting a linked list.

(f) 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) $\Theta(N\log N)$

B. MergeSort $O(N\log N) \sim O(N^2)$

C. Selection Sort $O(N^2)$

D. Insertion Sort

E. HeapSort $O($

N. (None of the above)

List all letters that apply. List them in alphabetical order, or if the answer is none of them, use N indicating none of the above. All answers refer to the entire sorting process, not a single step of the sorting process. For each of the problems below, assume that N indicates the number of elements being sorted.

__ABC__ Bounded by Ω(NlogN)lower bound.

__B.E__ Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

__C__ In the worst case, performs Θ(N) pairwise swaps of elements.

__A.B.D__ Never compares the same two elements twice.

__N__ Runs in best case Θ(logN)time for certain inputs

3 Bears and Beds

The hot new Cal startup AirBearsnBeds has hired you to create an algorithm to help them place their customers in the best possible homes to improve their experience. They are currently in their alpha stage so their only customers (for now) are bears. 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:

Given a list of Bears with unique but unknown sizes and a list of Beds with corresponding but also unknown sizes (not necessarily in the same order), return a list of Bears and a list of Beds such that that the ith Bear in your returned list of Bears is the same size as the ith Bed in your returned list of Beds. Bears can only be compared to Beds and we can get feedback on if the Bed is too large, too small, or just right. In addition, Beds can only be compared to Bears and we can get feedback if the Bear is too large for it, too small for it, or just right for it.

The Constraints:

Your algorithm should run in $O(N \log N)$ time on average. It may be helpful to figure out the naive $O(N^2)$ solution first and then work from there.

Our solution will modify quicksort. Let's begin by choosing a pivot from the Bears list. To avoid quicksort's worst case behavior on on sorted array, we will choose a random Bear as the pivot. Next we will partition the Beds into three groups — those less than, equal to, and greater than the pivot Bear. Next, we will select a pivot from the Beds list. This is very important — our pivot Bed will be the Bed that is equal to the pivot Bear. Given that the Beds and Bears have unique sizes, we know that exactly one Bed will be equal to the pivot Bear. Next we will partition the Bears into three groups — those less than, equal to, and greater than the pivot Bed.

Next, we will "match" the pivot Bear with the pivot Bed by adding them to the Bears and Beds lists at the same index, which is as easy as just adding to the end. Finally, in the same fashion as quicksort, we will have two recursive calls. The first recursive call will contain the Beds and Bears that are less than their respective pivots. The second recursive call will contain the Beds and Bears that are greater than their respective pivots.