CS 61B Fall 2023

Sorting

Exam-Level 12: November 6, 2023

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

Solution: Mergesort. One identifying feature of mergesort is that the left and right halves do not interact with each other until the very end. Further, note that the first line has had several steps applied to it, and yet is completely unchanged. This is reflective of how mergesort at first simply partitions the array without sorting anything.

(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

Solution: Quicksort. First item was chosen as pivot, so the first pivot is 1429, meaning the first iteration should break up the array into something like $| < 1429 | = 1429 | > 1429$

(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

Solution: Insertion Sort. Insertion sort starts at the front, and for each item, move to the front as far as possible. These are the first few iterations of insertion sort so the right side is left unchanged

(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

Solution: Heapsort. This one's a bit more tricky. Basically what's happening is that the second line is in the middle of heapifying this list into a maxheap. Then we continually remove the max and place it at the end.

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?

Solution: The array is nearly sorted. Note that the time complexity of insertion sort is $\Theta(N + K)$, where K is the number of inversions. When the number of inversions is small, insertion sort runs fast.

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

Solution: A simple example is: 5 4 3 2 1. Any 5 integer array in descending order would work.

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

Solution: False, stability for sorting algorithms mean 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. As a concrete example, consider the max heap: 21 20a 20b 12 11 8 7

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

Solution: Some possible answers: (1) Mergesort has $\Theta(N \log N)$worst case runtime versus quicksort's $\Theta(N^2)$. (2) Mergesort is stable, whereas quicksort typically isn't. (3) Mergesort is also preferred for sorting a linked list, because it is not necessary to create the auxiliary array to store the intermediate results. One can just modify the pointers of the linked list nodes to "snakeweave" the nodes in order.

(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. MergeSort

C. Selection Sort

D. Insertion Sort

E. HeapSort

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.

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

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

_____ Never compares the same two elements twice.

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

Solution: Bounded by $\Omega(N \log N)$ lower bound: A, B, C. All these sorts take at least $\Omega(N \log N)$. In a sorted list, insertion sort has linear runtime. Similarly, heapsort has linear runtime on a heap of equal items.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime: B, E. Only these two sorts are guaranteed to be better than $\Theta(N^2)$.

Never compares the same two elements twice: A, B, D. Quicksort never compares the same two elements, since after partitioning around a pivot that pivot is never used again for comparison. Mergesort also never compares two elements twice: the comparisons happen in the merge stage and we only compare items from diferent "halves" of the recursive call. Selection sort can compare the same items twice, since it requires finding the maximum on each iteration. Insertion sort never compares the same elements twice; comparisons happen when swapping to the front and once an item is at it's correct location it is never compared or swapped again. Heapsort may require multiple comparisons of the same items: during heapification and during bubbling down after a removal.

Runs in best case $\Theta(\log N)$ time for certain inputs: N. Sorting is theoretically lower-bounded by $\Theta(N)$–any sorting algorithm must examine each element at least once.

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 `Bed` is too large, too small, or just right.
  - `Beds` can only be compared to `Bears` 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.

Solution:
Our solution will modify quicksort. Let's begin by choosing a pivot from the Bears list. To avoid quicksort's worst case behavior on a 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.
Here is a video walkthrough of the solutions as well