CS 61B Spring 2019

Sorting

Exam Prep 12: April 15, 2019

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.

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

(a) Mergesort. One identifying feature of mergesort is that the left and right halves do not interactwith each other until the very end.

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

(b) 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 $| < 1429 | = 1429 | > 1429$

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, 9001, 4392, 7683

(c) 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

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

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

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

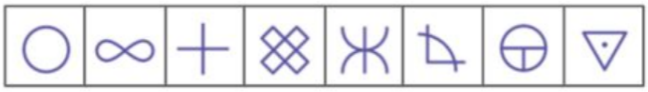
129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

2 Reverse Engineering

Consider the following unsorted array, and the array after an unknown number of iterations of selection sort as discussed in class (where we sort by identifying the minimum item and moving it to the front by swapping). Assume no two elements are equal

Unsorted:

After ? Iterations of Selection Sort:



For each relation below, **write** $<$, $>$, or ? for insufficient information regarding the relation between the two objects

1. $>$, circle is smallest element since it's placed in the front.
2. $>$, on the 5th iteration, the right object (one with the vertical line) is moved down the array before the left object
3. ?, we don't know whether they're in the right places or they haven't gotten to swap yet
4. $<$, infinity is second smallest element after circle

3 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) (T/F) Quicksort has a worst case runtime of Θ(NlogN), whereN is the number of elements in the list that we're sorting.

False, quicksort has a worst case runtime of Θ(N2), 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 Θ(N), which is when the array is already sorted.

(c) Give a 5 integer array such that it elicits the worst case running time for insertion sort.

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

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

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(taken from Stack Overflow), consider the max heap: 21 20a 20b 12 11 8 7

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

Some possible answers: mergesort has Θ(NlogN)worst case runtime versus quicksort's Θ(N2).Mergesort is stable, whereas 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 (nonrandom, inplace 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.

A, B, C 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