CS 61B Fall 2025

Sorting II

Discussion 12: November 17, 2025

1 Sort Identification

Match the sorting algorithms to the sequences, each of which represents several intermediate steps in the sorting of an array of integers. Assume that for quicksort, the pivot is always the first item in the sublist being sorted. Note that these steps are not necessarily the first few intermediate steps and there may be steps which are skipped.

Algorithms: *Quicksort, Merge Sort, Heapsort, Insertion Sort*

0.1
```
12, 7, 8, 4, 10, 2, 5, 34, 14
7, 8, 4, 10, 2, 5, 12, 34, 14
4, 2, 5, 7, 8, 10, 12, 14, 34
```

Solution: Quicksort. A pattern we can see is that most of the elements remain in the same order relative to one another, but the first element keeps moving around. Taking a closer look at this element, we note that everything to the left is less than it, and everything to the right is greater than, indicating quicksort.

0.2
```
23, 45, 12, 4, 65, 34, 20, 43
4, 12, 23, 45, 65, 34, 20, 43
```

Solution: Insertion Sort. A surefire way of identifying insertion sort is the fact it slowly builds a partially sorted array on the left hand side, which is exactly what occurs here.
Another solution is merge sort (with a recursive implementation), where the left half has already been fully merge-sorted

0.3
```
45, 23, 5, 65, 34, 3, 76, 25
23, 45, 5, 65, 3, 34, 25, 76
5, 23, 45, 65, 3, 25, 34, 76
```

Solution: Merge Sort. We notice that the numbers stay relatively close to where they begin, but we also see little sorted runs inside the array, like (23, 45) and (5, 65) for example. These small sorted subarrays and the merging of the subarrays in halves of the array is indicative of Merge Sort.

0.4
```
23, 44, 12, 11, 54, 33, 1, 41
54, 44, 33, 41, 23, 12, 1, 11
44, 41, 33, 11, 23, 12, 1, 54
```

Solution: Heapsort. Heapsort always heapifies the array first, which on the second line we see the maximum element, 54, at the start of the array. Heapifying usually shuffles around the array, which we also see. Placing the maximum element at the back of the array and bubbling up the 44 seals the deal that this is heapsort.

2 Quicksort

1.1 Sort the following unordered list using Quicksort. Assume that we always choose first element as the pivot and that we use the 3-way merge partitioning process described in lecture. Show the steps taken at each partitioning step.

`18, 7, 22, 34, 99, 18, 11, 4`

Solution:

```
-18-, 7, 22, 34, 99, 18, 11, 4
-7-, 11, 4 | 18, 18 | 22, 34, 99
4, 7, 11, 18, 18 | -22-, 34, 99
4, 7, 11, 18, 18, 22 | -34-, 99
4, 7, 11, 18, 18, 22, 34, 99
```

1.2 What is the best and worst case running time of Quicksort with Hoare partitioning on N elements? Given the two lists `[4, 4, 4, 4, 4]` and `[1, 2, 3, 4, 5]`, assuming we pick the first element as the pivot every time, which list would result in better runtime?

Solution: Best: $\Theta(N \log N)$ Running Quicksort on a list that has a pivot splits the partition exactly in half will result in $\Theta(\log N)$ levels, with the same amount work as above (i.e. $\Theta(N)$ at each level). For example, `[3, 1, 2, 5, 4]`. An alternative case is when we have all of the same element in the array (i.e. `[4, 4, 4, 4, 4]`), since the two pointers in Hoare partitioning always end up in the middle.

Worst: $\Theta(N^2)$. In general, the worst case is such that the partioning scheme repeatedly partions an array into one element and the rest.

Running Quicksort on a sorted list will take $\Theta(N^2)$ if the pivot chosen is always the first or last in the subarray: `[1, 2, 3, 4, 5]`. At each level of recursion, you will need to do $\Theta(N)$ work, and there will be $\Theta(N)$ levels of recursion. This sums up to $1 + 2 + ... + N$.

1.3 What are two techniques that can be used to reduce the probability of Quicksort taking the worst case running time?

Solution:

1. Randomly choose pivots.
2. Shuffle the list before running Quicksort.

3 Conceptual Sorts

Here is a video walkthrough of the solutions.

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.

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

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

2.2 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.

2.3 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.

2.4 (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

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

Solution:

(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 more memory efficient 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.

2.6 Describe how you might use a particular sorting algorithm to find the median of a list of N elements in worst case $\Theta(N \log N)$, without fully sorting the list.

Solution: We can use heapsort: the key here is noticing that we don't need to fully sort the list, which we would need to do with Quicksort and merge sort (no guarantees of median location in sorted list until fully sorted). We can heapify our elements and repeatedly pop off the maximum until we reach the middle element; since we know there are N elements, we know that once we've popped off N/2 elements, we've effectively sorted the back half of our list, so the median should be the next element popped off from the max heap. That way, we can save ourselves some work because we won't need to fully sort the rest of the list (though work done would still be $N \log N$).

2.7 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.

A, B, C bounded by $\Omega(N \log N)$ lower bound.

Solution: 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.

B, E Worst case runtime that is asymptotically better than quicksort's worst case runtime.

Solution: B, E. Quicksort has a worst case runtime of $O(N^2)$, while both merge sort and heapsort have a worst-case runtime of $O(N \log N)$.

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

Solution: C. When thinking of pairwise swaps, both selection and insertion sort come to mind. Selection sort does at most $\Theta(N)$ swaps, while it is possible for insertion sort to need $\Theta(N^2)$ swaps (for example, a reverse sorted array).

A, B, D Never compares the same two elements twice.

Solution: A, B, D. Notice for quicksort and merge sort that once we do a comparison between two elements (the pivot for quicksort and elements within a recursive subarray in merge sort), we will never compare those two elements again. For example, we won't compare the pivot against any other element again, and a sorted subarray will never have elements within it compared against one another. Insertion sort is much the same, as we bubble down elements into their respective places, comparing only against elements to the "left" of them.

Selection sort can compare the same items twice, since it requires finding the minimum on each iteration. Heapsort may require multiple comparisons of the same items: during heapification and during bubbling down after a removal.

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

Solution: F. The best case runtime for a sorting algorithm cannot be faster than $\Theta(N)$. This is because at the very least, we need to check if all elements are sorted, and since there are N elements, we can't have an algorithm that sorts faster than $\Theta(N)$.