CS 61B Fall 2025

Sorting II

Exam-Level Discussion 12: November 17, 2025

1 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.

0.1 (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.

0.2 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.

0.3 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.

0.4 (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

0.5 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.

0.6 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 $N \log N$).

0.7 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 $\Theta(N)$.

2 Sorted Runtimes

We want to sort an array of N **unique** numbers in ascending order. Determine the best case and worst case runtimes of the following sorts:

(a) Once the runs in merge sort are of size <= $\frac{N}{100}$, we perform insertion sort on them.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:

Best Case: $\Theta(N)$, **Worst Case:** $\Theta(N^2)$

Once we have 100 runs of size $\frac{N}{100}$, insertion sort will take best case $\Theta(N)$ and worst case $\Theta(N^2)$ time. Note that the number of merging operations is actually constant (in particular, it takes about 7 splits and merges to get to an array of size $\frac{N}{2^7} = \frac{N}{128}$).

(b) We use a linear time median finding algorithm to select the pivot in quicksort.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:

Best Case: Θ(Nlog(N)), **Worst Case:** Θ(Nlog(N))

Doing an extra N work each iteration of quicksort doesn't asymptotically change the best case runtime, since we have to do N work to partition the array. However, it improves the worst case runtime, since we avoid the "bad" case where the pivot is on the extreme end(s) of the partition.

(c) We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:

Best Case: Θ(Nlog(N)), **Worst Case:** Θ(Nlog(N))

While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list until the list is sorted in **descending order**. Once the list is in descending order, it can be sorted in ascending order with a simple linear time pass.

(d) We use any algorithm to sort the array knowing that:

- There are at most N inversions.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

  Best Case: $\Theta(N)$, **Worst Case:** $\Theta(N)$

  Recall that insertion sort takes $\Theta(N + K)$ time, where K is the number of inversions. Thus, the optimal sorting algorithm would be insertion sort. If $K < N$, then insertion sort has the best and worst case runtime of $\Theta(N)$.

- There is exactly 1 inversion.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Best Case: $\Theta(1)$, **Worst Case:** $\Theta(N)$

First, we notice that if there is only 1 inversion, it could only involve 2 adjacent elements. Intuitively, if two elements that are apart form an inversion, than some element between these two would also form an inversion with one of the elements.

(Optional) A formal argument is as follows: suppose the only inversion involves 2 elements that are not adjacent. Let's call their indices i and j, where $i < j$, and $a[i] > a[j]$ (definition of inversion). Because they are not adjacent, there exist some index k, such that $i < k < j$. In case 1, assume $a[k] > a[i]$. Then it follows that $a[k] > a[i] > a[j]$. Because $k < j$ but $a[k] > a[j]$, (k, j) forms an inversion, so we have a contradiction (we assumed only 1 inversion). In case 2, assume $a[k] < a[i]$, but then we have $k > i$, so (k, i) also form an inversion, which is also a contradiction.

Using this, we can just compare neighboring elements to find that exact inversion, and swap the 2 elements. If the inversion involves the first two elements, constant time is needed. If the inversion involves elements at the end, N time is needed.

- There are exactly $\frac{N(N-1)}{2}$ inversions.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

  Best Case: $\Theta(N)$, **Worst Case:** $\Theta(N)$

  If a list has $\frac{N(N-1)}{2}$ inversions, it means it is sorted in descending order. This is because every possible pair is an inversion (The total number of unordered pairs from N elements is $\frac{N(N-1)}{2}$, which is the number of inversions in a reverse-sorted list. So, it can be sorted in ascending order with a simple linear time pass. We know that reversing any array is a linear time operation, so the optimal runtime of any sorting algorithm is $\Theta(N)$.

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

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.