CS 61B — Sorting
Spring 2026 — Exam-Level Discussion 13: April 20, 2026

1 Summary (Fill in Names of the Sorts)

Sorting. A sorting algorithm rearranges an array into non-decreasing order. This requires a **total order** on the keys: for any `a` and `b`, exactly one of `a < b`, `a = b`, `a > b` holds (trichotomy), and `<` is transitive.

An **inversion** is a pair of indices `i < j` with `a[i] > a[j]`. Sorting is the process of reducing inversions to zero.

Selection sort. Repeatedly find the minimum of the unsorted suffix and swap it into the next sorted position. $\Theta(N^2)$ time on every input; $\Theta(1)$ extra space.

Heapsort.

- **Naive:** add each item to a separate max-heap, then repeatedly delete-max into a new array. $\Theta(N \log N)$ time, $\Theta(N)$ extra space.
- **In-place:** heapify the input array itself using bottom-up sink (linear time, not proven in class), then repeatedly swap the root with the last unsorted slot and sink. $\Theta(N \log N)$ time, $\Theta(1)$ extra space.

Mergesort (top-down). Recursively sort the two halves, then merge them into one sorted array. $\Theta(N \log N)$ worst case, but requires $\Theta(N)$ auxiliary memory for merging.

Insertion sort. For each index i, swap `a[i]` leftward while it is smaller than its left neighbor. Each swap removes exactly one inversion, so the total runtime is $\Theta(N + K)$ where K is the number of inversions. Best case $\Theta(N)$ (already sorted); worst case $\Theta(N^2)$ (reverse-sorted). Excellent on nearly-sorted arrays and on small arrays (roughly $N < 15$), so other sorts often switch to it as a base case.

Quicksort. Pick a pivot, **partition** the array so that every item left of the pivot is $\leq$ pivot and every item right is $\geq$ pivot (the pivot ends up in its final sorted position), then recurse on the left and right sides.

- 3-Scan: scan three times into a new array. $\Theta(N)$ time but $\Theta(N)$ extra space.
- Hoare: walk one pointer in from the left (stopping on items $\geq$ pivot) and another in from the right (stopping on items $\leq$ pivot), swapping whenever both have stopped. In-place, $\Theta(N)$ per partition.

Runtime: best and average $\Theta(N \log N)$; worst case $\Theta(N^2)$, which arises on sorted or reverse-sorted input when you always pick the leftmost item as pivot. Even fairly unbalanced splits (pivot at the 10% mark) still give $\Theta(N \log N)$. Space: $\Theta(\log N)$ on the recursion stack — much less than **mergesort**'s $\Theta(N)$.

Avoiding the $\Theta(N^2)$ worst case. Four philosophies:

- **Randomness:** shuffle the input before sorting, or pick random pivots.
- **Smart pivot selection:** take the median, or an approximation such as median-of-three.
- **Introspection:** track recursion depth and switch to **mergesort** if recursion goes too deep.
- **Preprocessing:** try to detect pathological inputs in advance (limited effectiveness).

Sort	Best	Worst	Extra space
Selection	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$
Heapsort (in-place)	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(1)$
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$
Insertion	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$
Quicksort (Hoare)	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(\log N)$

For interactive demos of every sort on this page, see `https://joshh.ug/61b/sorts/`.

2 All Sorts of Sorts

Show the steps taken by each sort on the following unordered list:

`0, 4, 2, 7, 6, 1, 3, 5`

(a) Insertion sort

Solution:

0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |

(b) Selection sort

Solution:

0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |

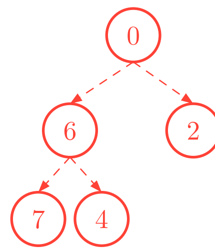
(c) Merge sort

Solution:

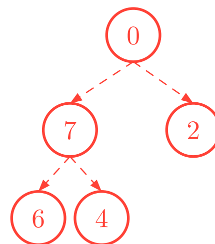
[0 4 2 7 6 1 3 5]
[0 4 2 7] [6 1 3 5]
[0 4] [2 7] [6 1] [3 5]
[0] [4] [2] [7] [6] [1] [3] [5]
[0 4] [2 7] [1 6] [3 5]
[0 2 4 7] [1 3 5 6]
[0 1 2 3 4 5 6 7]

(d) Use heapsort to sort the following array (hint: draw out the heap).
Draw out the array at each step: `0, 6, 2, 7, 4`

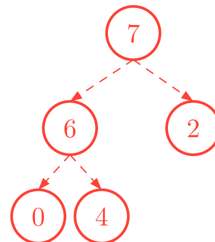
Solution: First, we need to heapify our array. We convert the current array to a max heap:



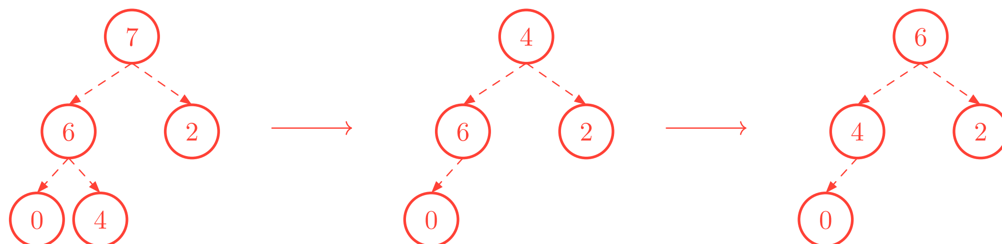
Recall that to heapify our array, we bubble down in reverse level order (bottom to top, right to left). This means we start by bubbling down 4, which in this case gives us the same heap structure. Bubbling down 7, and then 2, leaves the heap unchanged as well. Bubbling down 6 (swapping 6 and 7) then gives us the following:



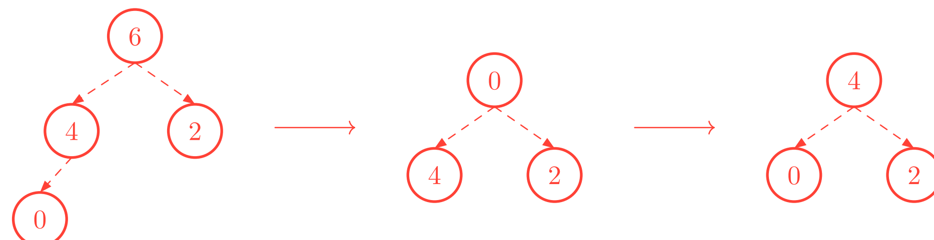
Bubbling down 0 gives us our final heap:



Note that as we heapify, we also modify the underlying array representation as well. This means that our final array looks like [**7, 6, 2, 0, 4**]. We then begin popping off the max value from the heap, placing it at the back of the array. Note that our underlying array representation doesn't consider the popped value as part of the heap any more. We start by popping off 7 and bubbling down:



Our array now looks like this: [**6, 4, 2, 0,** 7], where the bolded section is considered sorted and not part of the heap. We the continue by popping off 6:



and the array looks like [**4, 0, 2, 6, 7**]. We then pop off 4:

and our array looks like [**2, 0, 4, 6, 7**]. In a similar fashion, we pop off 2 and 0 from our heap, resulting in [**0, 2, 4, 6, 7**] and finally our sorted array: [**0, 2, 4, 6, 7**]

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) 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) 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.

(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) 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: 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: 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: 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: 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: 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 $\Theta(N)$.

4 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: $\Theta($ $\quad)$, Worst Case: $\Theta($ $\quad)$

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

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

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

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

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 $\Theta(N)$.