CS 61B Spring 2026

Sorting

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 sheet, see `https://joshh.ug/61b/sorts/`.

2 One Step at a Time

Consider the array below. For each part, give the array after the specified step.

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

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

(a) **Selection sort.** Give the array after the first swap.

Solution: `0, 2, 3, 7, 6, 1, 4, 5`. The minimum is `0` at index 6, which selection sort swaps into index 0.

(b) **Heapsort — heapify.** Give the array after bottom-up heapification into a max-heap (using 0-indexed array representation, so index `i`'s children are `2i + 1` and `2i + 2`).

Solution: `7, 6, 3, 5, 4, 1, 0, 2`. Bottom-up heapify sinks each internal node from index $\frac{n}{2} - 1 = 3$ down to index 0:

- Sink index 3 (`7`): only child is index 7 (`5`). `7 > 5`, no swap.
- Sink index 2 (`3`): children `1, 0`. Max child `1 < 3`, no swap.
- Sink index 1 (`2`): children `7, 6`. Swap with `7` → `4, 7, 3, 2, 6, 1, 0, 5`. Then at index 3 (`2`): child `5`, swap → `4, 7, 3, 5, 6, 1, 0, 2`.
- Sink index 0 (`4`): children `7, 3`. Swap with `7` → `7, 4, 3, 5, 6, 1, 0, 2`. Then at index 1 (`4`): children `5, 6`. Swap with `6` → `7, 6, 3, 5, 4, 1, 0, 2`.

(c) **Heapsort — one delete-max.** Starting from the heap you just built, perform one delete-max operation (the first step of heapsort's second phase). Give the resulting array, and mark which slot now holds a "sorted" value outside the heap.

Solution: `6, 5, 3, 2, 4, 1, 0,  7`. Swap the root (`7`) with the last heap slot (index 7, value `2`), shrink the heap to size 7, then sink the new root `2`:

- After swap: `2, 6, 3, 5, 4, 1, 0, 7` — index 7 now holds the extracted max.
- Sink index 0 (`2`): children `6, 3`. Swap with `6` → `6, 2, 3, 5, 4, 1, 0, 7`.
- Sink index 1 (`2`): children `5, 4`. Swap with `5` → `6, 5, 3, 2, 4, 1, 0, 7`.
- Index 3's children (`7, 8`) are outside the shrunken heap, so we stop.

(d) **Mergesort (top-down).** Give the array just before the final merge.

Solution: `2, 3, 4, 7, 0, 1, 5, 6`. The left half `4, 2, 3, 7` sorts to `2, 3, 4, 7`; the right half `6, 1, 0, 5` sorts to `0, 1, 5, 6`. Just before the final merge, the two sorted halves sit side by side.

(e) **Insertion sort.** Give the array after the first swap.

Solution: `2, 4, 3, 7, 6, 1, 0, 5`. At `i = 1`, `a[1] = 2` and `a[0] = 4`; since `2 < 4`, insertion sort swaps them immediately.

(f) **Insertion sort.** Give the array after `4`, `2`, and `3` have finished their turn as "the traveler."

Solution: `2, 3, 4, 7, 6, 1, 0, 5`. After `i = 1`'s round the array is `2, 4, 3, 7, 6, 1, 0, 5`. At `i = 2` the traveler is `3`: it swaps with `4` and then stops (since `3 > 2`), leaving `2, 3, 4, 7, 6, 1, 0, 5`.

(g) **Quicksort — 3-scan partition.** Give the array after one pass of the stable 3-scan partitioning algorithm from class, using the leftmost item as the pivot.

Solution: `2, 3, 1, 0, 4, 7, 6, 5`. Pivot is `4`. Scan 1 copies the keys `< 4` in their original order: `2, 3, 1, 0`. Scan 2 copies the keys `= 4`: `4`. Scan 3 copies the keys `> 4` in their original order: `7, 6, 5`.

(h) **Quicksort — Hoare partition.** Give the array after one pass of Tony-Hoare-style (in-place, two-pointer) partitioning, using the leftmost item as the pivot.

Solution: `1, 2, 3, 0, 4, 6, 7, 5`. With pivot `4`, the left pointer stops on items ≥ 4 and the right pointer stops on items ≤ 4, and they swap whenever both are stopped:

- Left stops at index 3 (`7`), right stops at index 6 (`0`). Swap $\rightarrow$ `4, 2, 3, 0, 6, 1, 7, 5`.
- Left stops at index 4 (`6`), right stops at index 5 (`1`). Swap $\rightarrow$ `4, 2, 3, 0, 1, 6, 7, 5`.
- Pointers cross (left $= 5$, right $= 4$). Finally swap the pivot into position `j = 4`: `1, 2, 3, 0, 4, 6, 7, 5`.

3 Identifying Sorts

The leftmost column (labeled 0) is the original input of strings to be sorted; the rightmost column (labeled 1) contains the strings in sorted order. The other columns show the contents of the array at some intermediate step during one of the sorting algorithms listed below. Match each algorithm to the correct column by writing its number in the blank under that column. Use each number exactly once.

This will require a lot of thinking! It's almost like a Sudoku or Crossword puzzle.

HELP	AMTR	EASE	AMTR	WATC	AMTR	AMTR	AMTR
IFYO	APPE	ETYP	APPE	SWAR	APPE	APPE	APPE
UARE	DINS	AREW	DINS	UARE	AREW	DINS	AREW
READ	EASE	HELP	HELP	SEND	DINS	EASE	DINS
INGT	HELP	EVIL	HISI	SORT	EASE	EVIL	EASE
HISI	HISI	HELP	IDEA	THEP	ETYP	AREW	ETYP
AMTR	IDEA	AMTR	IFYO	RATS	EVIL	ETYP	EVIL
APPE	IFYO	APPE	INGA	READ	HELP	HELP	HELP
DINS	INGA	DINS	INGT	RATS	HELP	HELP	HELP
IDEA	INGT	HELP	READ	RATS	HELP	HELP	HELP
SORT	ITHM	SORT	SORT	MALL	HISI	IFYO	HISI
INGA	LGOR	INGA	UARE	OVER	IDEA	UARE	IDEA
LGOR	OHPL	LGOR	LGOR	LGOR	LGOR	READ	IFYO
ITHM	READ	ITHM	ITHM	ITHM	ITHM	INGT	INGA
OHPL	SORT	OHPL	OHPL	OHPL	OHPL	HISI	INGM
EASE	UARE	IDEA	EASE	ETYP	INGT	IDEA	INGT
SEND	EVIL	SEND	SEND	APPE	SEND	SORT	ITHM
HELP	HELP	HISI	HELP	IFYO	RATS	INGA	LACE
RATS	MALL	RATS	RATS	DINS	RATS	LGOR	LGOR
EVIL	OVER	INGT	EVIL	EVIL	READ	ITHM	MALL
RATS	RATS	RATS	RATS	IDEA	RATS	OHPL	OHPL
SWAR	RATS	SWAR	SWAR	INGT	SWAR	SEND	OVER
MALL	SEND	MALL	MALL	IFYO	MALL	RATS	RATS
OVER	SWAR	OVER	OVER	HISI	OVER	RATS	RATS
THEP	AREW	THEP	THEP	INGA	THEP	SWAR	RATS
LACE	ETYP	LACE	LACE	LACE	LACE	MALL	READ
HELP	HELP	READ	HELP	HELP	INGA	OVER	SEND
RATS	INGM	RATS	RATS	HELP	RATS	THEP	SORT
AREW	LACE	UARE	AREW	AREW	UARE	LACE	SWAR
WATC	RATS	WATC	WATC	AMTR	WATC	RATS	THEP
INGM	THEP	INGM	INGM	INGM	INGM	WATC	UARE
ETYP	WATC	IFYO	ETYP	EASE	SORT	INGM	WATC
0	____	____	____	____	____	____	**1**

(0) Original input
(1) Sorted
(2) Selection sort
(3) Insertion sort
(4) Mergesort
(5) Quicksort (Hoare, no shuffle)
(6) Quicksort (3-scan, no shuffle)
(7) Heapsort

Solution: From left to right, the columns (excluding 0 and 1) correspond to algorithms 4, 5, 3, 7, 2, 6.

- **Column 2 → 4 (top-down mergesort).** The first 16 entries are sorted; entries 17–24 and 25–32 are each sorted but not yet merged — the recursive state where the left half is fully sorted and the right half's two sorted 8-subarrays are waiting to be merged.
- **Column 3 → 5 (standard quicksort).** Partway through recursive partitioning with the first element as pivot.
- **Column 4 → 3 (insertion sort).** The first 12 entries are sorted and match the sorted first 12 of the original; the remaining 20 entries are identical to the tail of the original input (untouched).
- **Column 5 → 7 (heapsort).** The column is a valid max-heap — each parent dominates its children (root `WATC`, children `SWAR` and `UARE`, etc.). This is the state right after heapify, before any delete-max operations.
- **Column 6 → 2 (selection sort).** The first 12 entries are the 12 smallest keys in sorted order; the rest is the leftover permutation after 12 min-select swaps.
- **Column 7 → 6 (3-way quicksort, 3-scan partition).** Using the first element `HELP` as the pivot, the stable 3-scan partition places the 7 keys less than `HELP` (in their original relative order) in positions 1–7, the 3 `HELP`s in positions 8–10, and the 22 keys greater than `HELP` (also in their original relative order) in positions 11–32.

4 Stability

Sometimes we want to sort data by more than one property. For example, suppose we have a roster of students, where each student is an instance of the Java class below:

```
public class Student {
    String name;
    int section;
    ...
}
```

Suppose we want to sort the students so that students in the same section are all together, and within each section they are in alphabetical order. To achieve this, we could do something like:

```
List<Student> studentList = ...;
Collection.sort(studentList, Student.NAME_COMPARATOR);
Collection.sort(studentList, Student.SECTION_COMPARATOR);
```

Throughout this problem you may find this simulator useful: `tinyurl.com/sp26-d13`.

(a) Would we get the correct result if we sorted by section, then name? Optionally: Use the simulator.

Solution: No. The trick with chained stable sorts is that the **last** sort is the **primary** key: its groups dominate the final arrangement, and earlier sorts only decide how ties are broken inside each of those groups.

The code above sorts by name first and by section last, so section is primary and name is the tiebreak — exactly what we want. If we flipped the order and sorted by section first and by name last, then **name** would become the primary key: the final list would be in alphabetical order by name, with ties broken by section — so students would not be grouped by section at all.

The approach above only works correctly if our sorting algorithm is **stable**. A sorting algorithm is stable if items that are equal under our comparison operation keep their original relative order after sorting. For example suppose our list is `(Alice, 3)`, `(Bob, 1)`, `(Carol, 3)`, and suppose we sort it by `section`.

- A **stable** sort produces `(Bob, 1), (Alice, 3), (Carol, 3)` — the two section-`3` students stay in their original roster order.
- An **unstable** sort might produce `(Bob, 1), (Carol, 3), (Alice, 3)`.

Stability matters when there are ties in the sort key that still carry meaningful identity (here, two `Student`s with the same section but different names). To explore this idea, try clicking the "advanced mode" on the simulator, toggle stability, and see how the desired sort fails without stability.

(b) Which of the following sorts (as implemented in class) are stable? Work out by careful thought or through experimentation with the "Even More Advanced" mode of the simulator.

(a) Insertion sort
(b) Quicksort with Tony-Hoare-style in-place partitioning
(c) Mergesort

Solution:

- **Insertion sort — stable.** The traveler only swaps with its left neighbor when it is **strictly** less, so two equal keys never cross each other.
- **Hoare quicksort — not stable.** The two pointers swap items across long distances. For example, on the roster `(Alice, 3), (Bob, 1), (Carol, 3)` with pivot `(Alice, 3)`: the left pointer stops on `(Alice, 3)` (since $\geq$ pivot), the right pointer stops on `(Carol, 3)` (since $\leq$ pivot), they swap, and finally the pivot is swapped into place, giving `(Bob, 1), (Carol, 3), (Alice, 3)` — the two section-`3` students have been reordered.
- **Mergesort — stable.** The merge step uses the tiebreak "if `L[i] <= R[j]`, take `L[i]`", which always picks the earlier (left) element on a tie, so equal keys keep their relative order.

For reference:

- **Selection sort — not stable**, because swapping the min into an early slot can jump a later item over an equal-keyed one (e.g., on `(Alice, 2), (Bob, 2), (Carol, 1)`, the first swap sends `Carol` to the front and `Alice` to the back, giving `(Carol, 1), (Bob, 2), (Alice, 2)`).
- **Heapsort — not stable**, because sink operations move items across unrelated equal keys.
- **3-scan partition — stable at the partition step**, since each scan walks left-to-right and preserves the original order inside each of the three groups.

(c) Java provides two overloads of `Arrays.sort`. On an `int[]` it uses a **quicksort** variant (dual-pivot quicksort). On an `Object[]` it uses a **mergesort** variant (TimSort). Why the difference?

Solution: Equal `int`s are bit-for-bit identical, so the order in which equal keys finish is unobservable and stability is irrelevant. Java therefore picks the option that tends to be fastest and uses the least memory — quicksort, which sorts in place with $\Theta(\log N)$ extra space.

For objects, two elements can compare equal while still being distinguishable (e.g., two `Employee` records with the same salary but different names). Programmers rely on `Arrays.sort` being stable when they sort by multiple keys in succession — for instance, sorting first by last name and then by department leaves employees grouped by department while still ordered by last name inside each department. Java uses a mergesort-based algorithm for object arrays to guarantee this stability.

5 Quicksort is a BST (Extra)

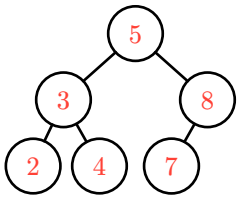
In lecture, we talked about how we can think of heapsort as a better way to use "selection" to sort: rather than repeatedly searching for the smallest item in linear time, we maintain knowledge of the maximum item in logarithmic time. There is a similar and more surprising connection between the idea of using "insertion" to sort and the BSTs.

Consider the array below.

`5, 3, 8, 2, 7, 4`

(a) Insert these numbers (in left-to-right order) into an initially empty BST. Write down every comparison that happens as each item is inserted (e.g., every item except the first will get compared to the root). How many comparisons total?

Solution: 8 comparisons. The resulting BST is:



The comparisons, grouped by insertion:

- insert `3`: `3 vs 5`
- insert `8`: `8 vs 5`
- insert `2`: `2 vs 5`, `2 vs 3`
- insert `7`: `7 vs 5`, `7 vs 8`
- insert `4`: `4 vs 5`, `4 vs 3`

(b) Now run 3-scan (or any partitioning-based) quicksort on the same array, using the leftmost item as the pivot at every level of recursion. Record all of the comparisons that happen. You should observe that the **exact same 8 comparisons** are made. In some sense, partition-based quicksort is just inserting the items into a BST.

Solution:

- Top level on `[5, 3, 8, 2, 7, 4]` with pivot `5`: compare each other item to the pivot → `3 vs 5`, `8 vs 5`, `2 vs 5`, `7 vs 5`, `4 vs 5`. Partitions: `[3, 2, 4] | 5 | [8, 7]`.
- Recurse on `[3, 2, 4]` with pivot `3`: `2 vs 3`, `4 vs 3`. Partitions: `[2] | 3 | [4]`.
- Recurse on `[8, 7]` with pivot `8`: `7 vs 8`. Partitions: `[7] | 8 | [ ]`.

The remaining sub-arrays are size 0 or 1 and need no comparisons. Total: the same 8 comparisons as part (a). Each pivot in quicksort plays the role of the corresponding node in the BST from part (a), and "goes in the left partition / right partition" corresponds to "descends left / descends right" in the BST.

6 Middle-Pivot Attack (Extra)

Suppose we use 3-scan partitioning with the **middle item** (rounded down — i.e. the item at index $\lfloor \frac{N-1}{2} \rfloor$) as the pivot at every level of recursion. Describe an input that gives $\Theta(N^2)$ runtime, assuming we don't shuffle the array first.

Solution: One surprisingly clean answer works for every even N: **odd numbers in descending order, followed by even numbers in ascending order.** For $N = 8$:

```
7, 5, 3, 1, 2, 4, 6, 8
```

The middle item is `1` (the smallest), so the partition is `[] | 1 | [7, 5, 3, 2, 4, 6, 8]`. The middle of that 7-item sub-array is `2`, again the smallest — and the pattern continues: at every level the pivot is the minimum, the partition is fully unbalanced, and the total work is $\Theta(N) + \Theta(N-1) + ... + \Theta(1) = \Theta(N^2)$.

The general construction (which also works for odd N) is: place the smallest value at index $\lfloor \frac{N-1}{2} \rfloor$, then — treating the remaining positions as a list in order — place the next-smallest at **that** list's middle, and so on. For $N = 7$ that yields `6, 4, 2, 1, 3, 5, 7`.