Your to-dos

1. Lab 5 (partner lab), due Tuesday 10/18 by 10pm.
2. Read before Mon: Bailey, Ch 10.

Sort stability

Suppose we are sorting on just the first letter.

Then \( ab < aa \) and \( ab > aa \).

Note also the positions of these elements in \( A: 0 < 2 \).

This sort is stable, because the relative order of \( ab \) and \( aa \) is the same.
Sort stability

A sort is **stable** if any two equal (or incomparable) objects retain their relative order in a sorted order as in an unsorted order.

Sort stability

More formally,

Let $A$ be an **array**, and $i$ and $j$ indices in that array, s.t. $i \neq j$.

Let $\pi_S(A,i)$ be a function that returns the updated index of $i$ after sorting $A$ with sorting algorithm $S$.


Merge sort

6 5 3 1 8 7 2 4

Merge sort

Invented by John von Neumann in 1948.
Merge sort is a sorting algorithm that uses the divide and conquer technique. It works by recursively partitioning data until no further partitioning is possible, then by merging elements of the partitions back together in sorted order.
Merge sort

Merge sort takes $O(n \times \log_2 n)$ time in the worst case (usually written $O(n \log n)$).

Merge sort takes $O(n \log n)$ time in the best case.

Merge sort takes $O(n)$ auxiliary space because each step makes a copy of the data being sorted.

I.e., merge sort is not an in-place sort. It is out-of-place.

Time complexity proof sketch

Divide takes $O(1)$ because we are just picking a midpoint.

Merge takes $O(n)$ because we have to copy $n/2$ elements into an array of size $n$ twice.

We divide $O(\log n)$ times and merge $O(\log n)$ times.

Therefore, the algorithm is $O(n \log n)$.

Space complexity

At first glance, this looks like $O(n \log n)$ space!

Why isn’t it?

Because after merging, we can discard old arrays (i.e., garbage collect) and reuse that space.
Quicksort is a sorting algorithm that uses the divide and conquer technique. It works by partitioning the data into two arrays around a pivot (a fixed element, like the first element).

It swaps data so that one array contains elements smaller than the pivot and the other array contains elements larger than the pivot. This ensures that, at each step, the pivot is in the correct position in the array.

Performing this procedure recursively on the left and right subarrays until there is nothing left to partition guarantees a sorted array.
Quick sort does not need to combine sub arrays after splitting—the entire array is guaranteed to be sorted upon reaching the base case, and since the sort is done in-place no copying is required.

**Base case** (array of size 1): the pivot is trivially sorted.

**Inductive case**: Assume that the left and right subarrays are sorted. Since the pivot is the middlemost element, then everything to the left is smaller and everything to the right is bigger. Therefore, the entire array is sorted.

Quick sort takes $O(n^2)$ time in the worst case. This case is improbable, and highly improbable as $n \to \infty$.

Quick sort takes $O(n \log n)$ time in the best case.

Quick sort takes $O(n \log n)$ time in the average case.

I.e., quick sort is an in-place sort. Therefore it needs no auxiliary space. As a result, quick sort is almost always chosen over merge sort in any application where all the data can fit into RAM.

In the worst case, we repeatedly choose the worst pivot (either the min or max value in the array). This means that we need to do $n-1$ swaps.

Since there are $n$ worst case choices of pivots, in the worst case, we do $n-1$ swaps $n$ times. $O(n^2)$.

In the best case, we always happen to choose the middlemost value as a pivot. I.e., the two subarrays are the same size. The rest of the proof looks just like the proof for merge sort where we intentionally choose two subarrays of the same size.

If you’re thinking that quick sort’s best case is the same as merge sort’s worst case, remember that quick sort is in-place.
## Sorting Wrapup

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$ - if “optimized”</td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>Worst = Best: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td>Merge</td>
<td>Worst = Best: $O(n \log n)$</td>
<td>$O(n) : 2n + c$</td>
</tr>
<tr>
<td>Quick</td>
<td>Average = Best: $O(n \log n)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Worst: $O(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>

## Recap & Next Class

### Today:
- Sort stability
- Merge sort
- Quick sort

### Next class:
- Linear structures