Reading 8 - Quick Sort

Tasks:

- 7.1: Understand the general principle of how D&C is being applied to QS
- 7.1: Understand the partition procedure
- 7.2: When does the worst-case behavior occur?
- 7.2: When does the best-case occur and how is the runtime recurrence derived?
- 7.2: Understand the argument for the balanced partitioning, especially how come that any constant ratio split produces the best-case runtime.
- 7.2: Understand the idea of good split - bad split. We are going to look into deriving the runtime recurrences in class.
- 7.3: Understand the idea of how the randomized QS is being done and why it is done the way it is being done.

7.1 - Description of Quick Sort

Quick sort has a recursive nature. Because of that it naturally implements the concept of **divide and conquer**.

The **partition procedure** of quick sort goes like this:

1. Select a pivot element from the array to create 2 separate **partitions**. Several methods include:
   - Selecting the middle element
   - Randomly selecting an element
   - Selecting the median of the first, middle, and last elements
2. Swap the pivot element with the last element of the array so we don't mess with it.
3. Swap the elements between the partitions so that all elements less than the pivot are on the left side and all elements greater than the pivot are on the right side.
4. Swapping the pivot back to its correct position (between the two partitions).

Once this partitioning is done, **quick sort** recursively calls itself twice, once for the **left partition** and once for the **right partition**.

The logic of "putting the smaller on the left" and "putting the larger on the right" is what naturally leads to a sorted array.

7.2 - Performance of Quick Sort

Lets imagine this array is like a seesaw:

```
[===================]   <-- the array
          ^             <-- the balance point
          |
----------|----------
```

When the seesaw is **level** (when the partitions are equally sized), quick sort performs at its best. That's because it's split exactly two equal parts, leading to the optimal recursion: $T(n/2)$, which leads to a runtime of $\theta(n \log n)$.

But when the seesaw is **tilted** in either direction (when one partition is much larger than the other, making it **unbalanced**), quick sort performs at its worst. That's because the greater partition has to make more recursive calls, leading to a recursion of $T(n-1)$, which leads to a runtime of $\theta(n^2)$.

However, in practice, we generally say that quick sort still performs good enough if the seesaw is **partially tilted**. That's because as long as the partitions are split in a **constant ratio** (like 70-30 or 80-20), the runtime still remains $\theta(n \log n)$. It's only in extreme cases where the seesaw is **totally tilted** (like 99-1) that quick sort's performance degrades to $\theta(n^2)$.

7.3 - A Randomized Version of Quick Sort

Earlier, I mentioned that one way to select a pivot is to randomly select an element to be that pivot. Since quick sort performs well enough on up to around 80-20 splits, we can say that it will perform well 80% of the time if we randomly select a pivot, with a 20% chance of it performing poorly. *We like those odds!* Plus, it frees us from having to think about how to select a good pivot algorithmically, or falling into the guaranteed worst-case scenario of already sorted data.