Module 3

Sorting Algorithms

▼ Time complexity of selection sort?

quadratic time O(N^2) for its worst and best asymptotic values

▼ How does selection sort work?

- Swap the smallest number with the first element, then move to the second element and swap the next largest with the second element, and so on.
- Start from the first element, which is the zero index and go all the way up to the second last element, whose index is equal to two less than the length of the array (last one doesn’t need a minimum check).

▼ Time complexity of insertion sort?

quadratic time O(N^2) for average runtime

▼ When to use insertion sort?

- It is an algorithm of choice when the data is nearly sorted or when the dataset is small.
- Given those two conditions, it can potentially outperform sorting algorithms that are O(n log(n)) time complexity, such as merge sort.
- Example:
  - When elements are sorted and need changing slightly every now and again. Think of a card game. A player receives a hand of cards and puts them in order. When the player is dealt a new card, it is added to their existing hand. The insertion sort can efficiently re-sort that list of cards.

▼ Insertion sort overview

- The insertion sort algorithm performs in-place sorting and works with any element type that conforms to the comparable protocol.
- It will make N-1 iterations, where i = 1 through N-1.
- The algorithm leverages the fact that elements 0 through i-1 have already been put in sorted order.

▼ How does insertion sort work (example)?

- We represent the array with a capital letter A (1).
- The first thing we do is to save the element at index i into a variable (let's call it current), so that even when we write over the index i, the element at i is saved in another variable, and we are able to access it later (2).
- Then we start accessing the elements of the array, starting from the previous index, which is i minus 1 (3).
- Now we would like to compare the elements of the array one by one with the current element, going to the left to check if the current element is less than these elements (4).
- We do this until either we have reached the left-most end of the array, or the elements at index j are greater than the current element. We can do this in a while-loop, and as long as this condition is true, we keep shifting the element at index j to the right by one (4).
- At line 5, 12 shifts to the right, then we reduce the value of j by one (6), so that we can access the element 8, so 8 is also greater than 7, so we move 8 to the right. Then, again reduce the value of j by one, because this is happening in a while-loop, so now the index j is pointing to element 5, and because 5 is less than 7, hence the while-loop ends, and what we need to do is to put the *current*, which is 7, in the array slot with index j plus one(7), because j, after the while-loop was executed, was pointing to the element 5.

```
# arr = [5, 2, 8, 12, 7]

1. for i = 0 to A.length-1
```

```
  2. current = A[i]
  3. j = i-1
    4. while j>=0 && A[j] > current
    5. A[j+1]=A[j]
    6. j = j-1
    7. A[j+1] = current
```

▼ What is quicksort?

- a divide-and-conquer algorithm
- it is a popular, fast-sorting algorithm that can perform sorting-in-place, so it is space-efficient

▼ What is the time complexity of quicksort?

- Its average running time is O(n log n), mainly due to its tight inner loop.
- It can have a worst case running time of O(n^2), but this can be minimized by ensuring the data is in a random order first. Additionally, ensuring that the correct pivot is selected will dramatically affect the algorithm's performance.

▼ How does quicksort work?

The algorithm works by dividing an initial array into two small subsequences, one with the lower sequences and another with the higher sequences, based on the pivot selected by a partitioning scheme.

▼ What advantage does quicksort have over merge sort?

Quicksort can be done in place, that is, without using any extra memory space, which is proportional to the data size.

▼ What is merge sort?

- a divide-and-conquer algorithm for sorting the elements in an array
- The majority of the sorting work is performed in the merge function, which is responsible for combining the two halves back together.
- Generally:
  - During the divide phase, it recursively splits the array into two halves, referred to as the left half and the right half.

    - That continues until the subarrays have been recursively divided down to their smallest, individual unit.
    - Those subarrays are considered to be sorted because they only have one element, and then from there the merge phase repeatedly merges those subarrays to produce new, larger sorted subarrays and it continues until there is only one subarray remaining, which is the final sorted result.

▼ How does merge sort work?

- **Divide**: If the collection S is zero or one, then return it, since it is already sorted. Otherwise, split the collection into two sequences, S1 and S2, with S1 containing the first N/2 elements of S, and S2 containing the remaining N/2 elements.
- **Conquer**: Recursively sort sublists S1 and S2; if they are small enough, then solve their base case. When the list has a single item or it is empty, it is considered sorted; this is called the base case.
- **Combine**: Merge the sorted S1 and S2 sublists into a sorted sequence and return the elements back.