Set 28: Sorts and Searches

Skill 28.01: Implement and interpret a selection sort

Skill 28.01 Concepts

The Selection Sort uses an incremental approach to sorting. During the first pass the smallest value is selected from the entire array and swapped with the first element. On the second pass the smallest value is selected from the array beginning with the 2nd element and swapped with the second element, etc.

The video below illustrates the selection sort concept,

https://youtu.be/jZBWRZZT8AI?si=bpfmHMu_MvveYhC2

Skill 28.01 Exercise 1

Skill 28.02: Interpret an insertion sort

Skill 28.02 Concepts

An insertion sort also uses an incremental approach. With an insertion sort, the array can be thought of as consisting of two parts, a sorted list followed by an unsorted list. The idea behind an insertion sort is to move elements from the unsorted list to the sorted list one at a time. As each item is moved, it is inserted into its correct position in the sorted list. To place the new item, the array elements need to be shifted to create a slot.

The video below illustrates the insertion sort concept:

https://www.youtube.com/watch?v=YgHlTV-c530

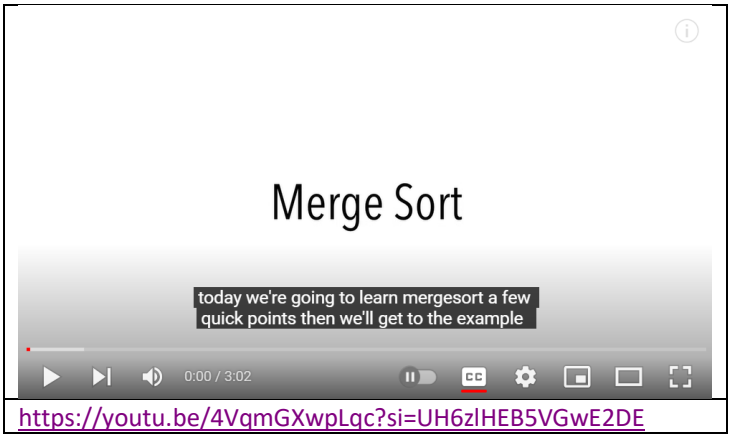
Skill 28.02 Exercise 1

Skill 28.03: Interpret a merge sort

Skill 28.03 Concepts

Selection and insertion sorts are inefficient for large n, requiring approximately *n* passes through a list of *n* elements. More efficient algorithms can be devised using a "divide-and-conquer" approach, which is used in the merge sort algorithm.

The Merge Sort uses the divide-and-conquer approach. It begins by placing each element into its own individual list. Then each pair of adjacent lists is combined into one sorted list. This continues until there is one big, final, sorted list. The process is illustrated in the video below.



https://youtu.be/4VqmGXwpLqc?si=UH6zlHEB5VGwE2DE

Skill 28.03 Exercise 1

Skill 28.04: Interpret a quick sort

Skill 28.04 Concepts

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. There are several different ways to choose the pivot value,

- pick first element as pivot
- pick last element as pivot
- pick a random element as pivot
- pick median as pivot

The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

This concept is further illustrated in the video below.

https://youtu.be/Hoixgm4-P4M?si=xQo2nMJLXNwHM_g8

Skill 28.04 Exercise 1

Skill 28.05: Implement a linear (sequential) search

Skill 28.05 Concepts

The sequential search, also known as the linear search, is the simplest search algorithm. The basic strategy is that every element in the data set is examined in the order presented until the value being searched for is found. If the value being searched for doesn't exist, a flag value is returned (such as -1 for an array).

If the data being searched are not sorted, then the sequential sort is a relatively efficient search. However, if the data being searched are sorted, we can do much better.

The video below illustrates the linear (sequential) search:

https://www.youtube.com/watch?v=aqFTmGbKajI

Skill 28.05 Exercise 1

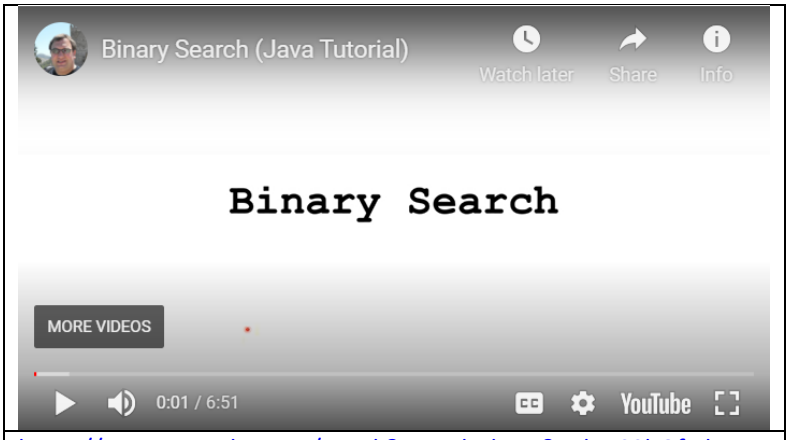
Skill 28.06: Interpret a binary search

Skill 28.06 Concepts

In a sequential search, after we compare against the first item, there are at most n−1 more items to look through if the first item is not what we are looking for. A key feature of a binary search is that the list being search **must** be sorted. A binary search takes advantage of a sorted list of elements, by first examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.

The video below illustrates the binary search:



https://www.youtube.com/watch?app=desktop&v=hKI93hOfeIk

The following code illustrates the implementation of a binary search,

```
function binarySearch(data, target) {

    var start = 0;
    var end = data.length - 1;

    while (start <= end)   {
         var mid = Math.floor((start + end) / 2);     /* Calculate midpoint */

         if (target < data[mid])     {
              end = mid - 1;
         }     else if (target > data[mid])     {
              start = mid + 1;
         }     else     {
              return mid;
         }
    }
    return -1;
}
```

Skill 28.06 Exercises 1 & 2