Sorting

Overview

We spend a lot of time sorting things

- it makes searching easier
- many problems in computer science are functionally just searching problems

In this module, we will learn about how to sort elements stored inside an array

Example:

- Sort the cats stored inside an array by their name alphabetically

Learning Objectives

- To be able to use **insertion sort** to sort elements inside an array
- To be able to use **selection sort** to sort elements inside an array
- To be able to use Java methods to sort an array or a list

MergeSort

We'll study two sorting algorithms today: Insertion Sort and Selection Sort.

In recitation, we'll reference **MergeSort**, which is a recursive sorting algorithm that usually runs faster than Insertion or Selection Sort. It is **not** considered testable material for this course, and is only included for your reference.

Insertion Sort: High Level View

- Maintain a sorted sub-section of the array starting at the beginning
    - This section doesn't account for elements outside of the section
    - This section starts as just the first element
- Continually add the next element of overall array to the sorted sub-section, shifting the elements in the subsection to maintain order
- After transfering the last unsorted element to the sorted subsection and adjusting it, sorting is done!

Insertion Sort

- Insertion sort compares the first two elements and put them in order
- Insertion sort then takes the third element and put it into the right position with respect to the first two
- Insertion sort then takes the fourth element and put it into the right position with respect to the first three
- And so on, until the entire array is sorted

Insertion Sort

20	10	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 1

20	10	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We add 10 to the sorted sub-section. To do so, we swap 10 and 20 since 10 < 20

Since there are no elements smaller than 10 in the sub-section, we are done processing 10 and can move on to the next element.

10	20	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 2

10	20	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 15 to 20 and swap them since 15 < 20

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 15 to the value at index 0

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 15 to the value at index 0

Since 10 < 15, we don't swap.

We can now consider 15 as "integrated" into the sorted subsection

10	**15**	**20**	**54**	**55**	**11**	**78**	**14**
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 3

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 54 to the value at index 2

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 54 to the value at index 2

Since 20 < 54, we don’t swap.

We can now consider 54 as "integrated" into the sorted subsection

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 4

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 55 to the value at index 3

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 55 to the value at index 3

Since 54 < 55, we don't swap.

We can now consider 55 as "integrated" into the sorted subsection

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 5

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 4

10	15	20	54	55	11	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 4

Since 11 < 55, we swap

10	15	20	54	11	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 3

Since 11 < 54, we swap

10	15	20	11	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 2

Since 11 < 20, we swap

10	15	11	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 1

Since 11 < 15, we swap

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 11 to the value at index 0

Since 10 < 11, we stop and consider the sub-section sorted

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 6

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 78 to the value at index 5

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 78 to the value at index 5

Since 55 < 78, we don’t swap.

We can now consider 78 as "integrated" into the sorted subsection

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We start to process index 7

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 6

10	11	15	20	54	55	78	14
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 6

Since 14 < 78, we swap the values

10	11	15	20	54	55	14	78
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 5

Since 14 < 55, we swap the values

10	11	15	20	54	14	55	78
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 4

Since 14 < 54, we swap the values

10	11	15	20	14	54	55	78
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 3

Since 14 < 20, we swap the values

10	11	15	14	20	54	55	78
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 2

Since 14 < 15, we swap the values

10	11	14	15	20	54	55	78
0	1	2	3	4	5	6	7

Insertion Sort

We compare 14 to the value at index 1

Since 11 < 14, we stop

Since 14 was the last value, the array is now sorted

10	11	14	15	20	54	55	78
0	1	2	3	4	5	6	7

Insertion Sort: Summary

For each unsorted element, swap current element with its predecessor, if out of order.

Repeat until the array is sorted.

Insertion Sort

```
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

- Why does the outer loop start at 1?
- Why does the inner loop start at `i`?
- Why does the inner loop have the continuation condition `(j > 0) && (array[j].compareTo(array[j-1]) < 0)`?

Insertion Sort

```
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

- **The first element is always sorted with respect to itself.**
- **We want to compare the first unsorted element to sorted elements only.**
- **We can stop as soon as we find an element that is smaller than the unsorted element.** All other elements to the left will also be smaller.

Poll:

Poll:

If we are doing Insertion Sort an array of size 5, what is the least number of comparisons that could be done during the sort?

```
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

What would the input array look like to cause this case?

Poll:

Poll:

If we are doing Insertion Sort an array of size 5, what is the least number of comparisons that could be done during the sort? **Four.**

```java
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

What would the input array look like to cause this case? **A sorted array!**

Poll:

If we are doing Insertion Sort an array of size 5, what is the **MOST** number of comparisons that could be done during the sort?

```
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

What would the input array look like to cause this case?

Poll:

If we are doing Insertion Sort an array of size 5, what is the **MOST** number of comparisons that could be done during the sort? **Ten.**

```
public static void insertionSort(Comparable[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j].compareTo(array[j-1]) < 0); j--) {
            Comparable temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
        }
    }
}
```

What would the input array look like to cause this case? **An array that is in descending order.**

Selection Sort

Find the `i`th smallest value, and put it at index `i`.

- Selection sort finds the smallest element in the array and place it at position 0
- then finds the smallest element in the array starting at index 1, and places it at position 1
- then finds the smallest element in the array starting at index 2, and places it at position 2

And so on, until the entire array is sorted

Selection Sort

Start by trying to find the index of the smallest value

20	10	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Index of smallest value: 1

Destination Index: 0

Selection Sort

Once the index of smallest is found. Swap it with index 0

10	20	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Selection Sort

Find the next smallest value of the array

10	20	15	54	55	11	78	14
0	1	2	3	4	5	6	7

Index of smallest value: 5

Destination Index: 1

Selection Sort

Once the next smallest is found, swap it with index 1

- Repeat until the (length-1)th smallest value is found and swapped
- (We've only sorted two elements—six more to go!)

10	11	15	54	55	20	78	14
0	1	2	3	4	5	6	7

Selection Sort

Selection Sort

- We initialize the position of the smallest element
- We update indexOfSmallest if we found a smaller element
- We place the smallest element at the right position

Selection Sort Code

```
public static void selectionSort(String[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int indexOfSmallest = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[j].compareTo(array[indexOfSmallest]) < 0) {
                indexOfSmallest = j;
            }
        }
        String temp = array[indexOfSmallest];
        array[indexOfSmallest] = array[i];
        array[i] = temp;
    }
}
```

Why do we stop the outer loop at `array.length - 1`?

Why do we start the inner loop at `i + 1`?

Selection Sort Code

```
public static void selectionSort(String[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int indexOfSmallest = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[j].compareTo(array[indexOfSmallest]) < 0) {
                indexOfSmallest = j;
            }
        }
        String temp = array[indexOfSmallest];
        array[indexOfSmallest] = array[i];
        array[i] = temp;
    }
}
```

There's no need to "sort" the last element—it'll just be the biggest.

The sorted portion of the array can always be found up to position `i`, and we start `indexOfSmallest` at position `i`.

Poll:

If we are doing Selection Sort an array of size 5, what is the least number of comparisons that could be done during the sort? What would the input array look like?

```
public static void selectionSort(String[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int indexOfSmallest = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[j].compareTo(array[indexOfSmallest]) < 0) {
                indexOfSmallest = j;
            }
        }
        String temp = array[indexOfSmallest];
        array[indexOfSmallest] = array[i];
        array[i] = temp;
    }
}
```

Sorting an Array: The Easy Way

These sorting algorithms are actually rather slow in practice. Java's built-in `Arrays.sort()` and `Collections.sort()` (for Lists) are much faster.

- refer to previous slide deck for examples of how to use
- these methods use Timsort, which is like a hybrid of Insertion and Merge sort.

```java
public class Party implements Comparable<Party> {
    private int partyHats;
    private String theme;
    private int numGuests;

    public Party(int partyHats, String theme, int numGuests) {
        this.partyHats = partyHats;
        this.theme = theme;
        this.numGuests = numGuests;
    }

    public int getPartyHats() {
        return partyHats;
    }

    public String getTheme() {
        return theme;
    }

    public int getNumGuests() {
        return numGuests;
    }

    public int compareTo(Party other) {
        if (numGuests == other.getNumGuests) {
            if (numGuests > 20) {
                if (partyHats == other.getPartyHats()) {
                    return theme.compareTo(other.getTheme());
                } else {
                    return partyHats - other.getPartyHats();
                }
            } else {
                if (theme.equals(other.getTheme())) {
                    return partyHats - other.getPartyHats();
                } else {
                    return theme.compareTo(other.getTheme());
                }
            }
        }
        return numGuests - other.getNumGuests();
    }
}
```