Activity: Simple Sorts

Learning Objectives:

- Trace through bubble, insertion, and selection sort algorithms Analyze
- and compare the performance of sorting algorithms

Time to Complete: 45 minutes

To Receive Credit: Submit individual PDF to Canvas **Instructions:**

> For these activities, follow the steps closely. Your goal is to complete all parts of the activity within the time provided.
>
> If a section is hidden behind a "Continue" button, you should make sure that you have completed the previous sections and answered any questions before moving on. Do not skip ahead or reveal any solution until you have completed the previous steps.

> ✓ **Collaboration Required**
>
> You will need a partner for this activity. One person will be the "Sorter" and the other will be the "Counter". You will switch roles throughout the activity.

Your Name(s):

Part 1: Bubble Sort (Warmup)

(10 minutes)

1. Designate one person as the "Sorter" and the other as the "Counter".
    - The **Sorter** will be the one in charge of manipulating the array.
        - Keep track of both indices `i` and `j` as you go.
            -

You can use your left hand for `i` and your right hand for `j`, etc.

- Say the following phrases out loud as you go:
  - Every time you make a *comparison*, say "comparison".
  - Every time you make a *swap*, say "swap".

The
- **Counter** will keep track of the number of comparisons and swaps made.
  - You should use a piece of paper to tally these.
  - You will also be responsible for answering questions about the process in this assignment.
    - You are also responsible for checking the Sorter's work and making sure they are making the correct moves!
- Both of you will be responsible for tracing through the code step-by-step, **exactly as written** -- you are the computer and must follow the instructions!

> **Don't have physical objects?**
>
> You can use a virtual deck of cards online: **Deck of Cards**

2. Using cards, pieces of paper, dice, or other objects, arrange the following numbers in an array like so:

Index	0	1	2	3
Value	4	2	5	3

- Note that the array has a length of 4, even if *physically* you may have room for more elements.
  - When tracing through the code, treat `arr.length` as **4**!

3. Take a look at the code for the Bubble Sort algorithm:

```
public static void bubbleSort(int[] arr) {      for
(int i = 0; i < arr.length - 1; i++) {          for
(int j = 1; j < arr.length - i; j++) { //
```

```
Note the (length - i)
            if (arr[j - 1] > arr[j]) { // This is a
"comparison"
                swap(arr, j - 1, j);
            }
        }
    }
}
```

The `swap()` method is not shown, but it should swap the elements at the given indices.

- For example, `swap(arr, 0, 3)` would swap the elements at indices 0 and 3 in the array `arr .`

4. Trace through `bubbleSort` and physically sort the array. Count the number of **comparisons** and **swaps** (separately) you make:

   - Again, treat `arr.length` as **4** when tracing through the code, meaning you should only change values between indices 0 through 3
   - What is the final order? (Just list the numbers)

   - How many comparisons did you make?

   - How many swaps did you make?

   Solution

   - The final order of the numbers is: `2 3 4 5` (sorted!)
   - You should have made **6 comparisons** and **3 swaps**.

5. Now, think about the following question:

- What basic operation should we count, comparisons or swaps? Justify your choice.

Solution

It depends, but we like to count **comparisons** since they are the most common basic operation in virtually all sorting algorithms.

The number of **swaps** can vary based on the values being sorted, meaning we would need to consider di ! erent cases. You may also notice that the number of swaps is always less than or equal to the number of comparisons.

Sometimes, it is helpful to analyze **both**!

Takeaway

The main idea of Bubble Sort is to repeatedly swap adjacent elements if they are in the wrong order. This causes the largest unsorted element to "bubble up" to its correct position.

Part 2: Insertion Sort

(15 minutes)

1. Arrange the following numbers in an array like so:

Index				
Value	4	2	5	

2. Take a look at the code for the Insertion Sort algorithm:

```
public static void insertionSort(int[] arr) {
for (int i = 1; i < arr.length; i++) {
for (int j = i; j > 0; j--) {
            if (arr[j - 1] > arr[j]) { // This is a
"comparison"
                swap(arr, j - 1, j);
            } else {
                break; // Exit the inner loop early
            }
        }
    }
}
```

3. Switch roles, then trace through the code and sort the elements. Count the number of **comparisons** and **swaps** you make:
   - How many comparisons did you make?
   - How many swaps did you make?

Solution

You should have made **5 comparisons** and **3 swaps**.

4. Now, arrange the elements like so:

Index	0	1	2	3
Value	5	4	3	2

5. Repeat the process of sorting and counting.

-

- How many swaps did you make?

Solution

You should have made **6 comparisons** and **6 swaps**.

How many comparisons did you make?

6. Let's add another element. Arrange the array like so:

Index	0	1	2	3	
Value	5	5	4	3	

- Note that there are **duplicates**! Make sure the duplicate numbers are unique somehow so that you can tell them apart (e.g., di!erent suits or colors).
- List the order of the two 5's in the array:
    - For example, "the green 5 is before the red 5".

7. Repeat the process of sorting and counting. Make sure you are still tracing through the code exactly as described.

- Now, your `arr.length` is **5**.
- How many comparisons did you make?
- How many swaps did you make?

Solution

You should have made **10 comparisons** and **9 swaps**.

That's much more than when n=4... is the growth linear? Or worse?

8. Before we move on, take a look at the final order of your numbers. Remember that we had multiple elements with the same number but unique in some way.
   - Take a look at the original order of the 5's. Let's look and see if this is still the case in the final order...
     -
     the same number? Describe.

     Solution

     Yes, the final order preserves the original order of the elements with the same number.

Stable Sorting

Insertion Sort is a ***"stable"*** sorting algorithm. This means that it preserves the relative order of equal elements in the sorted array.

Does your final order preserve the original order of the elements with

9. Now, think about worst- and best-case scenarios for Insertion Sort:
   - Think about the **worst-case** scenario for Insertion Sort. What would it look like?
     - For n=5, give an example of input that would result in the worst-case:

Solution

The worst-case scenario would be an array in **reverse order**.

This would require the maximum number of comparisons and swaps.

- Now, think about the **best-case** scenario for Insertion Sort. What would it look like?
  - For n=5, give an example of input that would result in the best-case:

Solution

The best-case scenario would be an array that is **already sorted**.

This would require the minimum number of comparisons and swaps.

10. Now, answer the following questions:

- Look at the code and think about what happens in the worst- and best-case scenarios.
- Consider **comparisons** as your basic operation to count.
  - Sort?
  - Comparisons: What is the best-case Big- θ time complexity of Insertion Sort?

Comparisons: What is the worst-case Big-θ time complexity of Insertion

Solution

- The worst-case time complexity of Insertion Sort is Θ(n^2).
- The best-case time complexity of Insertion Sort is Θ(n).

- Now consider **swaps** as your basic operation.
  -
  - Swaps: What is the best-case Big- θ time complexity of Insertion Sort?

Solution

- The worst-case time complexity of Insertion Sort is Θ(n^2) when counting swaps.
- The best-case time complexity of Insertion Sort is Θ(1) when counting swaps.

Takeaway

How Insertion Sort works is that it maintains a sorted "subarray" on the left. We then go through each element in the right and move it to its correct position in the sorted subarray. This is repeated until the entire array is sorted.

It is **stable** and has a best-case time complexity of Θ(n) when considering the number of comparisons. However, in the worst-case, it has a time complexity of Θ(n^2).

Swaps: What is the worst-case Big-θ time complexity of Insertion Sort?

Part 3: Selection Sort

(15 minutes)

1. Arrange the following numbers in an array like so:

Index	0	1	2	3
Value	4	2	5	3

2. Take a look at the code for the Selection Sort algorithm:

```
public static void selectionSort(int[] arr) {
for (int i = 0; i < arr.length - 1; i++) {
        // Find the largest item
        int bigindex = 0; // Index of the largest item
seen so far
        for (int j = 1; j < arr.length - i; j++) {
if (arr[j] > arr[bigindex]) { // This is a
"comparison"
                bigindex = j;
            }
}

        swap(arr, bigindex, (arr.length - 1) - i);
    }
}
```

3. Switch roles, then trace through the code and sort the elements. Count the number of **comparisons** and **swaps** you make:
   - How many comparisons did you make?
   - How many swaps did you make?

Solution

You should have made **6 comparisons** and **3 swaps**.

4. Now, arrange the elements like so:

Index	0	1	2	3
Value	5	4	3	2

5. Repeat the process of sorting and counting.
   -
   - How many swaps did you make?
   - How does this compare with the previous run? Anything to note?

Solution

You should have made **6 comparisons** and **3 swaps**.

How many comparisons did you make?

6. Now, arrange the following elements like so:

Index	0	1	2	3	
Value	5	5	4	3	

   - Again, make sure the duplicate numbers are unique.
   - List the order of the two 5's in the array:
     - For example, "the green 5 is before the red 5".

7. Repeat the process of sorting and counting.
   - How many comparisons did you make?

- How many swaps did you make?

Solution

You should have made **10 comparisons** and **4 swaps**.

8. Now, take a look at your final element order and compare with the initial order. Focus on the order of the duplicate elements.

Solution

No, Selection Sort is **not stable**. For example, the order of the 5's should have been reversed in the final order.

Is Selection Sort a stable sorting algorithm? Justify your answer.

9. Now, think about the worst- and best-case scenarios for Selection Sort:

- Think about the **worst-case** scenario for Selection Sort. What would it look like?
  - For n=4, give an example of input that would result in the worst-case:

Solution

Trick question! Selection Sort has the same number of comparisons and swaps regardless of the input. It always makes the same number of comparisons and swaps.

10. Let's try it. Arrange the elements like so:

Index	0	1	2	3	
Value	2	3	4	5	

11. One more time, repeat the process of sorting and counting.
    -
    - How many swaps did you make?
    - Is this the same or di ! erent from the previous run?

    Solution

    You should have made **10 comparisons** and **4 swaps**. This is the same as the previous run.

    How many comparisons did you make?

12. Now, analyze the code and answer the following questions:

-
  -

Solution

The worst-case time complexity of Selection Sort is Θ(n^2) when counting comparisons.

- Now consider **swaps** as your basic operation.
  - Swaps: What is the Big- θ time complexity of Selection Sort?

Consider **comparisons** as your basic operation.

Comparisons: What is the Big-θ time complexity of Selection Sort?

Solution

The worst-case time complexity of Selection Sort is Θ(n) when counting swaps.

13. Finally, answer the following:

    - In your own words, describe how Selection Sort works.

    - Now describe how Insertion Sort works.

    - Finally, describe how Bubble Sort works.

> **Takeaway**
>
> Selection Sort also splits the array into a sorted part and unsorted part.
>
> It repeatedly **selects** the largest (or smalles) element from the unsorted part of the array and moves it to the end of the sorted part.

Part 4: Working Out Exact Time Complexity

(remaining time)

1. For each algorithm, consider the **number of comparisons** as the basic operation.

    - Work out the exact time complexity for each algorithm.
        - Use B(n) for best-case and W(n) for worst-case.

-

- Comparisons: Insertion Sort (best/worst case):

- Comparisons: Selection Sort (best/worst case):

Solution

Algorithm	Best Case	Worst Case
Bubble Sort	n(n-1)/2	n(n-1)/2
Insertion Sort	n-1	n(n-1)/2
Selection Sort	n(n-1)/2	n(n-1)/2

Comparisons: Bubble Sort (best/worst case):

2. Now, consider the **number of swaps** as the basic operation.

- Work out the exact time complexity for each algorithm.
  - Use B(n) for best-case and W(n) for worst-case.

-

Swaps: Bubble Sort (best/worst case):

- Swaps: Insertion Sort (best/worst case):

- Swaps: Selection Sort (best/worst case):

Solution

Algorithm	Best Case	Worst Case
Bubble Sort	0	n(n-1)/2
Insertion Sort	0	n(n-1)/2
Selection Sort	n	n

3. It may be helpful to visualize the sorting algorithms in the future. Visit the following link: VisuAlgo - Sorting
   - You can select the algorithm you want to visualize at the top and click "Sort" in the lower left.
   - Play around with this tool to get a feel for how the algorithms work.
4. If you still have time, can you write a swap method?
   - Write a method that swaps two elements in an array. You can assume the array is an integer array.

```
public static void swap(int[] arr, int i, int j) {
    // Your code here
}
```

   - What is the code for the swap method?

5. If you finish super-super-super early, think about the following:
    - We have considered comparisons and swaps as separate basic operations. Let's count array accesses instead. Say that each comparison requires 2 array accesses and each swap requires 4 array accesses.
    - Come up with the exact time complexity for each algorithm in terms of array accesses.
        - Array Accesses: Bubble Sort (best/worst case):
        - Array Accesses: Insertion Sort (best/worst case):
        - Array Accesses: Selection Sort (best/worst case):

Submission

For these activities, you will typically submit a PDF report to Canvas. First, click the "Export as PDF" button below.

Export as PDF | Reset Activity | Hide All | Show All

Then, submit the PDF to Canvas. Everyone must submit their own copy to receive credit.