Lesson Plan 00 - Exploring Sorting Algorithms

Instructor: Andy Mina
Grade Level and Subject: AP Computer Science A, 11-12th grade
Length: 45m

NYS Computer Science and Digital Fluency Learning Standards	9-12.CT.4 ● **Concept:** Computational Thinking ● **Sub-concept:** Abstraction and Decomposition ● **Description:** Implement a program using a combination of student-defined and third-party functions to organize the computation 9-12.CT.9 ● **Concept:** Computational Thinking ● **Sub-concept:** Algorithms and Programming ● **Description:** Systematically test and refine programs using a range of test cases, based on anticipating common errors and user behavior.
Content Objective	By the end of this lesson, students will be able to: ● Define what a sorting algorithm is ● Identify applications of sorting, in the real world and in computer application ● Write a simple sorting algorithm, \`SelectionSort\`, for a list of numbers ● Write simple test cases for a sorting algorithm ● Talk about the time and space complexity of specific sorting algorithms using Big-O notation
Scaffolding Needed	Before this lesson, students should be able to: ● Identify the time complexity of simple iterative functions (for-loops, doubly nested for-loops, etc.) ● Demonstrate reasonable control over arrays including, but not limited to, the following

	functionality: ◦ Appending an item ◦ Removing an item ◦ Looping through an array ◦ Accessing elements by index ◦ Assigning elements by index
Key Vocabulary	• **Sorting algorithm:** an algorithm that puts a list of elements into a defined ordering. For numbers, this may mean value, or for letters, this might be alphabetical. • *Sorting Orders* ◦ **Non-decreasing:** the value of elements either increases OR remains the same as you move through the list ◦ **Increasing:** the value of elements will ALWAYS increase as you move through the list ◦ **Non-increasing:** the value of elements either decreases OR remains the same as you move through the list ◦ **Decreasing:** the value of elements will ALWAYS decrease as you move through the list • **Selection Sort:** a sorting algorithm that works by taking the largest/smallest element of a list, placing it in the correct position, and then repeating the process with the remaining elements • **Space complexity:** the total space/memory used by an algorithm with respect to its input size
Assessments	<u>Formative Assessment</u> Students will be asked to “roll for confidence” and respond by showing the instructor a number from 1 to 5 on one of their hands. Their scores are representative of how confidently they understand and can independently engage with the material thus far. Scores should be interpreted as follows: 1. **Not confident:** needs a re-explanation or summary of the lesson with emphasis on key points.

	2. **Needs review:** needs a brief recap and some teacher-guided practice to solidify concepts and understanding. 3. **On track:** needs some peer-guided practice and some more time to let things sink in. *Ideal rating after the lesson.* 4. **Confident.** needs some peer-guided practice for more challenging problems, but is comfortable with engaging with the class material. 5. **Self-sufficient:** needs little to no guidance and can tackle problems of exceptional difficulty with relative ease. Indicative of an under-challenged student. **These checks shouldn't take any longer than a few seconds.**
Materials	☐ Computers able to run Java code ☐ A deck of cards ☐ Starter Code (also see Appendix) ☐ Solution Code (also see Appendix)

Lesson Component	**Time**	**Execution of Lesson Component**
Essential Question		"What is a sorting algorithm and when should I use one?"

Warm-up	< 5m	Prompt students to complete one of the three following tasks: 1. Recall a time you've sorted **anything** in real life. Think carefully about the steps you took to do so. Write one example of the different methods we use to sort items. ○ Consider that we may use different methods/steps to sort different items: playing cards in your hand, books on a shelf, etc. 2. If you know already, define what a sorting algorithm is. List 1-3 examples of sorting algorithms you may know and describe in 1-2 sentences how they work. 3. What might be some practical use cases for sorting something in code? Think about how this might apply to websites, video games, or other forms of digital media you may have encountered. Write down 3-5 use cases of sorting in the digital world.
Presentation of Content	< 10m	**Background Knowledge** Ask students the context in which they've sorted items before and the metric for sorting. Did they sort: ● books alphabetically on a shelf? ● playing cards by their value in their hand? ● members of their family by height? Encourage students who answered *Warm-up #1* to share their responses. Emphasize that sorting is done by using a comparative metric, whether it be alphabetically or by some quantitative value (height, money, grades, etc.). **Definitions** Define what a sorting algorithm is for students. Encourage students who answered *Warm-up #2* to share their responses. ● **Sorting algorithm:** an algorithm that puts elements of a list into an order. Define the following vocabulary and give examples: ● *Sorting Orders* ○ **Non-decreasing:** the value of elements either increases OR remains the same

    as you move through the list
        - `[1, 2, 2, 3]`
    - **Increasing:** the value of elements will ALWAYS increase as you move through the list
        - `[1, 2, 8, 11]`
    - **Non-increasing:** the value of elements either decreases OR remains the same as you move through the list
        - `[49, 17, 17, 17]`
    - **Decreasing:** the value of elements will ALWAYS decrease as you move through the list
        - `[4, 3, 2, 1]`

Applications in the Digital World
Focus the conversation on computer science and ask students why they might sort items when programming. Encourage students who answered *Warm-up #3* to share their responses. Provide examples: “Take Spotify (or any other music streaming platform). We can sort songs in a playlist by a few different metrics: date added, length, number of global listens, and most notably, alphabetically.”

Stop for QCC if needed. “Roll for confidence” on sorting algorithms and different sorting orders.

Writing a Sorting Algorithm	< 15m	**Code Along Prep** Inform students that we're going to write a sorting algorithm called `Selection Sort`. Remind students that, just as in real life, sorting can look different or have different steps. Explain that Selection Sort works by 1. Selecting the largest element in the list 2. Moving the largest element to its rightful place by swapping it with the element currently there 3. Repeating steps 1-2 with the remaining unsorted portion of the array Model how Selection Sort works using a deck of cards (method: unplugged activity/modeling). Stop for QCC if needed. **Code Along** Once students understand how Selection Sort works, provide them with `Starter.java` which contains the function signature for Selection Sort and some helper functions: `findIndexOfMax`, `swap`, and `testFriendlyCase`. Helper functions will be visited later in the lesson to test our implementation (method: scaffolding). Conduct a code-along session using student input as the driving force (method: code along). See annotated `Solution.java` for details. Stop for QCC as needed. "Roll for confidence" on how SelectionSort works and implementation.
Testing the Code	< 5m	**Testing and Debugging** When students have implemented their Selection Sort algorithm, instruct them to call the `testHappyPath` function in their `main`. *"We are testing our code with pre-defined inputs and known outputs to be sure that it works as expected. This is a common practice in the industry; nobody likes broken code :("* Explain that the phrase "happy path" in the function name means we're testing "normal" input for our function. This helper test function makes sure that our Selection Sort algorithm returns the expected output and prints the result. Instruct students to come up with other test cases and inputs to be confident that their code works. They should aim to break their code with funky test cases as much as possible. We want to be sure our code works for any array of numbers. Consider the following cases:

		● The array passed is \`null\` ● The array passed is empty ● The array passed contains duplicates ● The array passed *only* contains duplicates
Time and Space Complexity	< 7m	**Analyzing the Algorithm** Students should already be familiar with Big-O notation. Walk students through deriving the Big-O time complexity for Selection Sort - \`O(n^2)\`. Inform students that although Big-O is commonly used to talk about time complexity, it can also be used to talk about space complexity. Ask students about the space this algorithm uses. Walk them through deriving the Big-O space complexity for Selection Sort - \`O(1)\` since it doesn't use any memory that scales with input size. "Roll for confidence" on time/space complexity of Selection Sort.
Debrief	< 3m	**Recap and Review** Briefly recap the topics covered during class: ● Learning about sorting ● Knowing how Selection Sort works ● Testing our code Ask students review questions and have them answer aloud. Questions may include: ● Define what a sorting algorithm is. ● Give an example of a non-decreasing list of elements. ● Is it possible for a list to be both non-increasing AND decreasing? ● Explain why the space complexity for Selection Sort is \`O(1)\`. ● Why might we sort something in the digital world?

Appendix

Starter.java

```java
import java.util.Arrays;

public class Starter {
  public static void main(String[] args) {
    // test your selection sort
    // print if the test passed or failed
  }

  // ================= SORTING SECTION ==================
  /**
    An implementation of the Selection Sort algorithm. Repeatedly finds the largest
    element of an array and swaps it into place for an unsorted array.

    @param numbers - a list of integers to sort

    @return a sorted list of integers
  */
  public static int[] selectionSort(int[] numbers) {

  }

  /**
    Helper function that finds the index of the largest element in the array.

    @param numbers - a list of integers
    @param endIdx - the index to stop searching inclusive
  @return the index of the largest element before or including endIdx
*/
public static int findIndexOfMax(int[] numbers, int endIdx) {
  int maxIdx = 0;
  for (int i = 0; i <= endIdx; i++) {
    if (numbers[i] > numbers[maxIdx]) {
      maxIdx = i;
    }
  }

  return maxIdx;
}

/**
  Helper function that swaps the values of two indices in an array.
  NB: we're using a temp variable here to store the value of numbers[idxA].
  What do you think would happen if we didn't use this temp value?

  @param numbers - a list of integers
  @param idxA - the first index to be swapped
  @param idxB - the second index to be swapped
*/
public static void swap(int[] numbers, int idxA, int idxB) {
  int temp = numbers[idxA];
  numbers[idxA] = numbers[idxB];
  numbers[idxB] = temp;
}

// ================= TESTING SECTION ===================
```

```java
  public static boolean testHappyPath() {
    // define the input and expected output
    int[] input = { 2, 5, 1, 4, 3 };
    int[] expectedOutput = { 1, 2, 3, 4, 5 };

    // sort the input using our implementation
    int[] actualOutput = selectionSort(input);

    // check that the arrays match
    return Arrays.equals(expectedOutput, actualOutput);
  }
}
```

Solution.java

```java
import java.util.Arrays;

public class Solution {
  public static void main(String[] args) {
    // test your selection sort
    // print if the test passed or failed
  }

  // ================= SORTING SECTION ==================
  /**
    An implementation of the Selection Sort algorithm. Repeatedly finds the largest
    element of an array and swaps it into place for an unsorted array.
  @param numbers - a list of integers to sort

  @return a sorted list of integers
*/
public static int[] selectionSort(int[] numbers) {
  /**
    DELIBERATE ERROR: it's possible that numbers is null. This is a
    suggested test case after the code along. Let students handle
    that extension on their own.
  */

  // COMMENT: we make a copy here so we don't modify the original.
  int[] sortedNumbers = numbers;

  /**
    PROVIDE: we loop backward here to make sure we don't accidentally
    swap an element that was already swapped into the correct place.
  */
  for (int idxToSwap = sortedNumbers.length - 1; idxToSwap > 0; idxToSwap--) {
    /**
      BIG-REVEAL: Because we're using well-made helper functions, this
      is the entire logic of our selection sort. It follows the description
      talked about in class exactly.
    */

    // STUDENT-PROMPT: use the function we worked on!
    int maxIdx = findIndexOfMax(idxToSwap);

    // STUDENT-PROMPT: use the function we worked on!
    // BIG-IDEA: the swap function is what DOES the sorting!
    swap(sortedNumbers, idxToSwap, maxIdx);
  }
}

/**
  Helper function that finds the index of the largest element in the array.

  @param numbers - a list of integers
  @param endIdx - the index to stop searching inclusive

  @return the index of the largest element before or including endIdx
*/
public static int findIndexOfMax(int[] numbers, int endIdx) {
  // QUESTION: why are we setting maxIdx to 0 here? what if we set it to -1?
  int maxIdx = 0;
  for (int i = 0; i <= endIdx; i++) {
    if (numbers[i] > numbers[maxIdx]) {
      maxIdx = i;
    }
  }

  return maxIdx;
}

/**
  Helper function that swaps the values of two indices in an array.

  @param numbers - a list of integers
  @param idxA - the first index to be swapped
  @param idxB - the second index to be swapped
```

```
   */
  public static void swap(int[] numbers, int idxA, int idxB) {
    /**
      MUST-ANSWER: we're using a temp variable here to store the value of
      numbers[idxA]. What do you think would happen if we didn't use this temp value?
    */
    int temp = numbers[idxA];
    numbers[idxA] = numbers[idxB];
    numbers[idxB] = temp;
  }

  // ================= TESTING SECTION ===================
  /**
    BIG-IDEA: this is how we test code! we come up with an example
    input and expected output for that input. then we pass the
    input to function to sort and checked that the provided
    output matches the expected output.
  */
  public static boolean testHappyPath() {
    // define the input and expected output
    int[] input = { 2, 5, 1, 4, 3 };
    int[] expectedOutput = { 1, 2, 3, 4, 5 };

    // sort the input using our implementation
    int[] actualOutput = selectionSort(input);

    // check that the arrays match
    return Arrays.equals(expectedOutput, actualOutput);
  }
}
```

[redacted]