Lesson Plan 01 - Comparing 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.6 ● **Concept:** Computational Thinking ● **Sub-concept:** Algorithms and Programming ● **Description:** Demonstrate how at least two classic algorithms work and analyze the trade-offs related to two or more algorithms for completing the same task.
Content Objective	By the end of this lesson, students will be able to: ● Identify the time and memory complexity of ○ Merge Sort, a classic, divide-and-conquer sorting algorithm ○ Bogo Sort, an intuitive, yet inefficient, sorting algorithm ● Compare the tradeoffs of Merge Sort and Bogo Sort ● Code a working implementation of Merge Sort and Bogo Sort with helper code
Scaffolding Needed	Before this lesson, students should be able to: ● Identify simple iterative functions' time and space complexity (for-loops, doubly nested for-loops, etc.) ● Identify the time and space complexity of Selection Sort ● 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

	• Understand and implement a recursive algorithm
Key Vocabulary	• **Divide-and-conquer algorithm:** an algorithm that continually breaks down large problems into smaller ones for simpler solutions and then uses the simple solutions to solve the original problem • **Merge Sort:** a *divide-and-conquer* sorting algorithm that works by merging sorted subarrays into sorted order • **Bogo Sort:** a probabilistic sorting algorithm that works by randomly shuffling elements in an array until they are in sorted order
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 represent 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 teacher-guided practice to solidify concepts and understanding. 3. **On track:** needs some peer-guided practice and 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 (enough cards to distribute 4-8 to each student) ☐ Starter Code (also see Appendix) ☐ Solution Code (also see Appendix)

Lesson Component	Time	Execution of Lesson Component
Essential Question		“Why might we choose one sorting algorithm over another?”
Warm-up	< 3m	Students walk in to see a set of cards at their desks. As the lesson warm-up, they will briefly race against a partner to see who can sort their list of cards in the fewest number of “steps.” The following actions are each considered one “step”: ● Comparing the value of two cards ● Swapping the position of two cards ● Each card that is processed while looping or “scanning” through the list ● Creating a new sub-list of cards Students are to devise an algorithm for sorting cards with the fewest number of steps and write down their method (method: CS unplugged).
A brief recap of Lesson 00	< 4m	Remind students that a sorting algorithm is an algorithm that sorts items in a list. There are many different use cases for sorting in the real world and the digital world. For example, we may want to sort books on a shelf in real life or sort songs by their streams on Spotify. “There are many different sorting algorithms, all of which are defined by the steps they take to sort. Last class, we looked at and programmed Selection Sort. Can someone remind me how Selection Sort works in a sentence or two?” Student answer or elicit: 1. First, the algorithm loops through the array. 2. While looping through, it finds the largest element. 3. Once the largest element is found, it is swapped with the element in its correct position (the very last one). 4. Repeat steps 1-3 for the remaining elements “What is the time and space complexity of Selection Sort in Big-O notation?” Student answer or elicit:

		• Time Complexity: O(n^2) • Space Complexity: O(1) “Today we’re going to explore another sorting algorithm, Merge Sort, and compare it to Selection Sort. We’ll be talking about how both algorithms work and talk about when it might be better to use over the other.”
Presentation of Merge Sort	< 6m	“Merge Sort is a sorting algorithm that sorts an array by splitting it into smaller arrays and merging them back in sorted order.” “Merge Sort is a special type of sorting algorithm called a *divide-and-conquer* algorithm. You may have heard this phrase in your history classes as it’s also used to describe common battle tactics. Merge Sort gets this *divide-and-conquer* label because it works by *dividing* the problem into smaller, more manageable problems. For example, sorting a list of 1000 elements might be difficult, but sorting a list of 1 element is really easy; it’s already sorted! Once the *divided* problem has been solved, it can be considered *conquered*.” Using a deck of cards, model how Merge Sort works for the class (method: CS unplugged). For demonstrative examples, use arrays where the size is a power of two; this makes halving neat. Emphasized that an array of size 1 is sorted, by definition. If students are confused by this concept, ask, “How many ways can we order one item?” Use the following script for modeling: “On a more technical level, Merge Sort works by splitting the array in half, into subarrays. It does this until there’s only one item in each subarray. As we mentioned earlier, there’s no work required to sort an array of size 1 since it’s already sorted. At this point, there are multiple sorted subarrays and the algorithm merges them together in sorted order, finally giving you the original sorted list.” Inform students that since Merge Sort essentially sorts smaller arrays, it’s easiest implemented with recursion. The array will be “divided” using a start and end index. Stop for QCC. Roll for confidence about how Merge Sort works.

Implementing Merge Sort	< 12m	Direct students to rejoin the pairs they were working in at the beginning of class and open the Starter.java code given to them. Students will pair program a working implementation of Merge Sort as described in class. The starter code given to them thoroughly documented with their two tasks: create a helper function `mergeArrays` and create the driver function `mergeSort`. Tests are provided in Tests.java and are called in the `main` function so students can be sure their code works as intended. For students that finish early, encourage them to write more thorough tests for their code. The existing tests are small and relatively simple cases. They should write tests with more elements and more unique cases.
Time and space complexity	< 7m	Reconvene and walk through the time complexity for Merge Sort. Guide students to discover this answer on their own with the following investigative questions: ● How many subarrays do we end up creating? What is the space complexity of Merge Sort? ○ We create n subarrays since we eventually divide the original array into subarrays of individual elements. ● How long does it take us to reach a subarray of size 1? ○ We continuously divide the array in half to reach the subarray of size 1. The repeated division of a number is analogous to taking the log of a number. In this case, we take the log base 2 of the array size. In Big-O, it would take O(log n) time to reach a single subarray of size 1. ● How long does it take to merge two sorted arrays? ○ Since we must go to each index of both arrays it takes n time where n = leftArr.length + rightArr.length. ● If we have to create a subarray of size 1, which takes log n time, for each element of the n elements and then merge sorted arrays, what is the total time complexity of Merge Sort in Big-O notation? ○ The total time is the sum of the time it takes to create subarrays of 1 and merge the subarrays: $n\log_2 n + n$. Since Big-O only cares about the dominating terms, Merge Sort is an $O(n\log_2 n)$ algorithm. ● Since we create n subarrays of size 1, what is the space complexity of Merge Sort? ○ O(n).

		Roll for confidence on analyzing the time and space complexity of Merge Sort.
Comparing sorting algorithms	< 10m	Now that students have worked through two sorting algorithms, reiterate the time and space complexity of each. • Merge Sort: time - O(n log n); space O(n) • Selection Sort: time - O(n^2); space O(1) Point out that according to the Big-O analysis we did, Merge Sort is faster, but uses more memory than Selection Sort. However, Selection Sort doesn't use any extra memory at all, so it is suitable for smaller, less powerful machines. With most modern technology, we almost always want to optimize the speed of the algorithm over the memory used. Ask students what algorithm, Merge Sort or Selection Sort, would be better for this case. However, in some special cases like machines with limited hardware, we want to optimize the memory used, even if the algorithm takes longer. In these situations, Selection Sort would be better suited. For example, the Apollo 11 rocket which took humans to the moon only had about ~40kb of total memory, making our Selection Sort the ideal option. Most modern phones have about 64GB which is 1.6 million times more, so using extra memory in Merge Sort is not a big deal. Roll for confidence about the tradeoffs of Merge Sort vs. Selection Sort and when each one might be more useful than the other.
Summary	< 3m	Briefly recap how Merge Sort and Selection Sort work. Merge Sort works by continuously dividing the array until each subarray is sorted and then merges those subarrays back in sorted order. Selection Sort works by looping through the array to find the largest element and putting it in the correct place until the list is sorted. Each algorithm has its own benefits and drawbacks. Merge Sort is quick but uses more memory than Selection Sort. Selection Sort uses no extra memory but isn't as fast as Merge Sort.

Appendix

Starter.java

```java
/**
  Name:
  Date:
  Activity: Merge Sort

  - Comments with the "TODO" header are tasks to be completed.
  - Comments with the "HINT" header are there to help you.
  - There are tests for Merge Sort and the helper merge function
    in main to make sure things work.
*/

public class Starter {
  public static void main(String[] args) {
    System.out.println("------ Running mergeArrays tests -----");
    System.out.println("Happy path: " + testMergeArrays_HappyPath());
    System.out.println("Empty left array: " + testMergeArrays_EmptyLeft());
    System.out.println("Empty right array: " + testMergeArrays_EmptyRight());

    System.out.println("------- Running mergeSort tests ------");
    System.out.println("Happy path: " + testMergeSort_HappyPath());
    System.out.println("Backward array: " + testMergeSort_BackwardArray());
    System.out.println("All duplicates: " + testMergeSort_AllDuplicates());
  }

  /**
    ========================= TODO =========================
```

```
  Using the description below as a guide, write a function
  called "mergeArrays" that merges two sorted arrays of
  integers in sorted order, and returns the result. Your
  function must:
    - accept two arrays of integers as a parameters
    - return the merged array in sorted order

  ==================== MERGING STEPS ====================
  1. Create an array to store the result of the merged
     input arrays.
  2. Loop through both of the input arrays simultaneously.
     You want to be able access both of their elements at
     the same time.
  3. For every index, compare the current value in the
     first input array to the current value in the second
     input.
  4. Add the smaller value found in Step 3 to the result
     array.
  5. Repeat Steps 2-4 until one of the arrays is empty.
  6. Add the remaining elements to the result array.
  7. Return the result array created.

  ======================== HINT ========================
  Creating multiple variable to track the current index
  being processed in all three arrays is helpful.
*/

/**
  ======================== TODO ========================
  Using the description below as a guide, write a working
```

```
    implementation of the Merge Sort algorithm that sorts an
    array of integers. Your implementation must:
      - accept an array of integers as a parameter, but you
        are free to add other parameters you find helpful
      - use recursion
      - return a sorted array of integers

    ============== MERGE SORT STEPS =================
    1. If the array only has one element, it is sorted.
    2. Calculate what index represents the middle of the array.
    3. Using that index, sort the left half and the right half.
    4. Merge the sorted left and right halves together in sorted order.
    5. Return the sorted, merged array.

    ================ HINT ====================
    Since Merge Sort works on the same array but only processes a small
    window of numbers, you may find it helpful to include start and end
    indices for the function as parameters.
  */
}
```

Tests.java

```java
public class Tests {
  /** ============= MERGE ARRAYS TESTS ================== */
  public static String testMergeArrays_HappyPath() {
    int[] leftArr = { 1, 2, 3 };
    int[] rightArr = { 4, 5, 6 };
    int[] expectedOutput = { 1, 2, 3, 4, 5, 6 };
```

```
  int[] actualOutput = mergeArrays(leftArr, rightArr);

  return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";

}

public static String testMergeArrays_EmptyLeft() {
  int[] leftArr = {};
  int[] rightArr = { 1, 2, 3 };
  int[] expectedOutput = rightArr;

  int[] actualOutput = mergeArrays(leftArr, rightArr);

  return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";
}

public static String testMergeArrays_EmptyRight() {
  int[] leftArr = { 4, 5, 6 };
  int[] rightArr = {};
  int[] expectedOutput = leftArr;

  int[] actualOutput = mergeArrays(leftArr, rightArr);

  return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";
}

/** ============ MERGE SORT TESTS ================== */
public static String testMergeSort_HappyPath() {
  int[] input = { 4, 561, 51, 34, 68 };
```

```
    int[] expectedOutput = { 4, 34, 51, 68, 561 };
    int[] actualOutput = input;

    mergeSort(actualOutput, 0, actualOutput.length - 1);

    return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";
  }

  public static String testMergeSort_BackwardArray() {
    int[] input = { 41, 34, 29, 29, 5};
    int[] expectedOutput = {5, 29, 29, 34, 41};
    int[] actualOutput = input;

    mergeSort(actualOutput, 0, actualOutput.length - 1);

    return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";

  }

  public static String testMergeSort_AllDuplicates() {
    int[] input = { 28, 28, 28, 28, 28 };
    int[] expectedOutput = input;
    int[] actualOutput = input;

    mergeSort(actualOutput, 0, actualOutput.length - 1);

    return Arrays.equal(actualOutput, expectedOutput) ? " PASSED ✅" : " FAILED ❌";
  }
}
```

Solution.java

```java
/**
  Name:
  Date:
  Activity: Merge Sort

  - Comments with the "TODO" header are tasks to be completed.
  - Comments with the "HINT" header are there to help you.
  - There are tests for Merge Sort and the helper merge function
    in main to make sure things work.
*/

public class Solution {
  public static void main(String[] args) {
    System.out.println("------ Running mergeArrays tests -----");
    System.out.println("Happy path: " + testMergeArrays_HappyPath());
    System.out.println("Empty left array: " + testMergeArrays_EmptyLeft());
    System.out.println("Empty right array: " + testMergeArrays_EmptyRight());

    System.out.println("------- Running mergeSort tests ------");
    System.out.println("Happy path: " + testMergeSort_HappyPath());
    System.out.println("Backward array: " + testMergeSort_BackwardArray());
    System.out.println("All duplicates: " + testMergeSort_AllDuplicates());
  }

  /**
    ========================= TODO =========================
    Using the description below as a guide, write a function
    called "mergeArrays" that merges two sorted arrays of
    integers in sorted order, and returns the result. Your
```

```
    function must:
      - accept two arrays of integers as a parameters
      - return the merged array in sorted order

    ==================== MERGING STEPS ====================
    1. Create an array to store the result of the merged
       input arrays.
    2. Loop through both of the input arrays simultaneously.
       You want to be able access both of their elements at
       the same time.
    3. For every index, compare the current value in the
       first input array to the current value in the second
       input.
    4. Add the smaller value found in Step 3 to the result
       array.
    5. Repeat Steps 2-4 until one of the arrays is empty.
    6. Add the remaining elements to the result array.
    7. Return the result array created.

    ========================= HINT ========================
    Creating multiple variable to track the current index
    being processed in all three arrays is helpful.
  */
  public static void mergeArrays(int[] leftArr, int[] rightArr) {
    // ========== STEP 1 ==============
    int[] mergedArr = new int[leftArr.length + rightArr.length];

    // Taken from the hint for ease.
    int leftIdx = 0;
    int rightIdx = 0;
```

```
  int mergedIdx = 0;

  // ============ STEP 2 ===============
  while (leftIdx < leftArr.length & rightIdx < rightArr.length) {
    // ============ STEP 3 ==========
    /**
      If the current item in leftArr is smaller than the current
      item of rightArr, then it should be inserted next.
    */
    if (leftArr[leftIdx] < rightArr[rightId]) {
      mergedArr[mergedIdx] = leftArr[leftIdx];
      leftIdx++;
    }

    /**
      If the current item in rightArr is less than *OR* equal to
      the current item in leftArr, then it should be inserted next.
    */
    if (rightArr[rightIdx] <= leftArr[leftId]) {
      mergedArr[mergedIdx] = rightArr[rightId];
      rightIdx++;
    }

    // since we've inserted an item, adjust the mergedIdx
    mergedIdx++;
  }

  /**
    ============== STEP  6 ============
    If any items remain in either array, we know that they are all
```

```
    bigger than what was in the other array. In this case, just
    append them to the mergedArr.
  */

  // insert whatever items remain in leftArr
  while (leftIdx < leftArr.length) {
    mergedArr[mergedIdx] = leftArr[leftIdx];
    mergedIdx++;
    leftIdx++;
  }

  // insert whatever items remain in rightArr
  while (rightidx < rightArr.length) {
    mergedArr[mergedIdx] = rightArr[rightIdx];
    mergedIdx++;
    rightIdx++;
  }

  // ================ STEP 7 ============
  return mergedArr;
}


/**
  ========================== TODO ==========================
  Using the description below as a guide, write a working
  implementation of the Merge Sort algorithm that sorts an
  array of integers. Your implementation must:
    - accept an array of integers as a parameter, but you
      are free to add other parameters you find helpful
```

```
    - use recursion
    - return a sorted array of integers

  ============== MERGE SORT STEPS =================
  1. If the array only has one element, it is sorted.
  2. Calculate what index represents the middle of the array.
  3. Using that index, sort the left half and the right half.
  4. Merge the sorted left and right halves together in sorted order.
  5. Return the sorted, merged array.

  ================ HINT ====================
  Since Merge Sort works on the same array but only processes a small
  window of numbers, you may find it helpful to include start and end
  indices for the function as parameters.
*/
public static int[] mergeSort(int[] arr, int leftIdx, int rightIdx) {
  /**
    ============== STEP 1 ==============
    If leftIdx == rightIdx, that means we're only looking at one element.
    Return since this is sorted already.

    If leftIdx > rightIdx, that means we've already sorted everything
    in the original array. Again, return the array since it's sorted
    already.
  */
  if (leftIdx >= rightIdx) {
    return arr;
  }

  /**
```

```
      ============== STEP 2 ===========
      Finding the middle index of the array. Note that leftIdx is added
      to make sure we're in bounds. Consider what would happen if we didn't
      add leftIdx to the result:
        - leftIdx = 3
        - rightIdx = 6
        - midIdx = (rightIdx - leftIdx) / 2 = (6 - 3) / 2 = 1 OUT OF BOUNDS!!!
    */
    int midIdx = leftIdx + (rightIdx - leftIdx) / 2;

    /** ============== STEP 3 ============= */
    int[] leftArr = mergeSort(arr, leftIdx, midIdx);
    int[] rightArr = mergeSort(arr, midIdx, rightIdx);


    /** ========== STEP 4 ================= */
    int[] sortedArr = mergeArray(leftArr, rightArr);


    /** ========== STEP 5 ================= */
    return sortedArr;
  }
}
```