

Project 3: Bucket Sort

A non-comparison based sorting algorithm.

Overview	1
Learning Goals:	1
Style Guide:	2
Project Specification:	2
Export and Submit	6
Dev Tips	7

Overview

Most sorting algorithms you encounter are designed to be “general purpose” sorts, where any data type can be sorted as long as a method for comparing them is defined. In Java, this method is called “`compareTo`” in the `Comparable` interface. Any class that implements `Comparable` can be sorted because they can be compared.

There are other sorting algorithms that do not rely on comparing data directly. These are not general purpose in the sense that they have to be customized for different types of data. In this project you will implement one of these sorting algorithms called Bucket Sort. The version of Bucket Sort will be customized to work with integers, but it could work with any symbolic language.

Learning Goals:

- Implement a non-comparison based sorting algorithm.
- Use a two-dimensional array to support a sorting algorithm.
- Write methods that support other, higher-level methods.
- Gain experience debugging and running unit tests.
- Continue to use a required style guide.

Style Guide:

Follow the style guidelines as outlined in the Java Style Guide on Moodle.

Project Specification:

Background.

Bucket Sort uses an array of “buckets” to repeatedly distribute and then collect data until it is sorted.

In this project, you will work with integers. In Bucket Sort, one needs to know how many buckets are needed. One bucket is required for each integer. The number of digits, and therefore the number of buckets is 10, the individual digits, (base ten) being: 0,1,2,3,4,5,6,7,8,9.

Now that we know the number of buckets we need, let’s look at an example of how we can sort some integers. Let’s say we have these numbers to sort: 25, 492, 3, 22.

We set up ten buckets, one for each digit. We represent the buckets as a two-dimensional array, where the rows are the buckets and the columns are for storing data as in Fig. 1.

b u c k e t s	0				
	1				
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				

Fig 1: A bucket array.

ROUND ONE

The next step is to examine the individual numbers in the data starting with the rightmost number. Since we are dealing with base ten numbers, we can say that we will look at each 1's place number first, then the 10's place, then the 100's place, etc.

For the example that has 25, 492, 3, 22, the 1's place number in the integer 492 is 2, its 10's place number is 9, and its 100's place number is 4.

The Bucket Sort starts by looking at the 1's place numbers in the data and distributing the data into the bucket that corresponds to their 1's place number. The bucket array would look like this after this first round:

b u c k e t s	0				
	1				
	2	492	22		
	3	3			
	4				
	5	25			
	6				
	7				
	8				
	9				

Fig 2: Bucket array after 1st round.

Note that bucket 2 contains 492 and 22. That's because they both have 2 in their 1's place. Also note that the data is sorted on the 1's place.

ROUND TWO

The next step in the algorithm is to collect the data starting with bucket 0 and ending with bucket 9. The data is collected in the order it is stored in. For example, you would collect 492 before 22.

This is the data after it has been collected from the bucket array: 492, 22, 3, 25.

Now we do another round of distribution, but on the 10's place numbers. The bucket array is cleared out first. Fig. 3 shows the array after the second round:

b u c k e t s	0	3			
	1				
	2	22	25		
	3				
	4				
	5				
	6				
	7				
	8				
	9	492			

Fig 3: Second round of distribution.

Note that for the round with 492, 22, 3, 25, the number 492 is in bucket 9 because its 10's place number is 9. Note also that 3 is in bucket 0- why? Because it has no 10's place number, so we use 0 instead. After collecting the data from the array, we get: 3, 22, 25, 492. Remember, we have to reset the bucket array to clear it out at the end of each round.

The data is now sorted, but we are not done yet! Because at least one of the data has a 100's place number, we need to do another round. In fact, we need to know the maximum number of digits of the integers in the data before the sort can begin. This is because the algorithm needs to process each digit, one at a time. For the data in our example, the maximum number of digits in an integer is 3, because the integer 492 has three digits.

ROUND THREE

For round 3, for the numbers 3, 22, 25, 492, we would get this distribution in the array, using 0 for all the data without a 100's place number:

b u c k e t s	0	3	22	25	
	1				
	2				
	3				
	4	492			
	5				
	6				
	7				
	8				
	9				

Fig 4: Third round distribution.

After distribution, we collect the data for the last time and the final result is: 3, 22, 25, 492. The maximum number of digits in the data was 3, so we are done and the data is sorted.

Implementation note:

The data structure pictured above is a two-dimensional array, with the buckets as rows. The number of columns has to be equal to the size of the data, as it could be possible, though extremely unlikely, that all of the data has the same number for a specific place, and the array has to accommodate that possibility. Obviously, the space that this two-dimensional array requires is impractical for a large amount of data. There are other structures that can be used instead, such as linked lists, but for this project you will work with a two-dimensional array of integers.

Pseudocode:

Here is the pseudocode for the bucket sort algorithm.

Parameters:

numBuckets - number of digits (10 in the case of base ten integers).

The buckets are the rows of the bucketArray.

MAX_DIGIT_LIMIT - maximum number of digits for integers in the data.

dataLen - length of the data.

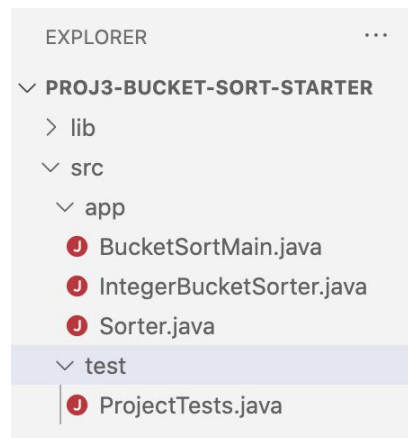
dataArray - an array of the data to be sorted.

bucketArray = int[numBuckets][dataLen]

```
//placeNum the integer's value at 1's, 10's, 100's, etc.
for placeNum = 1 to MAX_DIGIT_LIMIT {
    // distribution phase
    for each dataItem in dataArray
        curBucket = get value of dataItem at current placeNum
        columnIndex = get first available column in curBucket row
        write data item to bucketArray[curBucket][columnIndex]
    // collection phase
    for each row in bucketArray
        write each dataItem stored in the row to dataArray
    clear all values from the bucketArray
}
```

Code Structure and TODOs.

In VSCode when you open the project folder you should see:



1. BucketSortMain.java allows you to test your code by choosing different input data.
2. All of the 7 TODOs are found in the IntegerBucketSorter.java file. Each TODO asks you to implement a method. Explanations of what the methods should do are in the

comments for each method. We recommend that you implement the TODOs in order from 1 to 7.

3. The `Sorter.java` is an interface that `IntegerBucketSorter` implements.
4. The JUnit tests are in the `ProjectTests.java` file.

After all the ToDos are completed and you run `BucketSortMain.java`, the sorted output with the sample input data is:

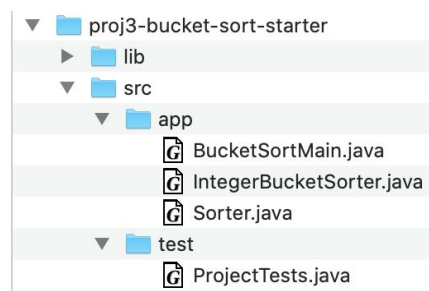
```
input:
100, 23, 92, 498, 12, 29, 48, 354, 1, 57, 33
output:
1, 12, 23, 29, 33, 48, 57, 92, 100, 354, 498
```

Export and Submit

Step 1: Export your project

Within VSCode click on the “*View > Command Palette...*” menu option. Then type into the Command Palette: “*Archive Folder*” and hit enter. This will produce a Zip file of your project folder. You can then upload that zip file to the corresponding project assignment in Gradescope. You can add the [Archive](#) extension to VSCode if you don’t have it.

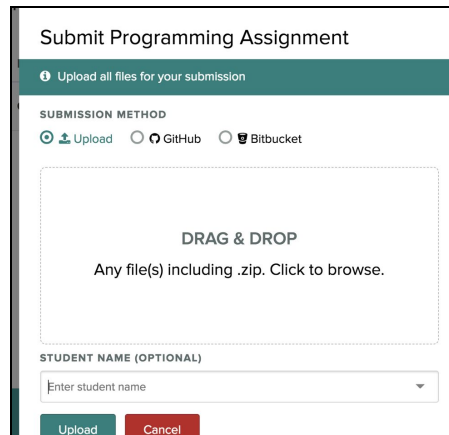
If your zip file is not in the correct form, the autograder will not process your code and you will not receive any credit. After creating the zip file, you can check that it’s correct by unzipping it and checking that you see this file structure:



Note that we will not grade code manually. If your code does not compile or if your zip file is not correctly structured you will not receive any credit for the project.

Step 2: Submit the zip file to Gradescope

Log into Gradescope, select the Project 2 assignment, and submit the zip file for grading.



There are usually more tests in the autograder than provided with the starter code. If your code passes all provided tests, it is a good indication that your code will pass all of the autograder tests. If that is not the case, you are likely not considering some of the “edge” cases in your algorithm. See the Dev Tips below for what to do.

The autograder will not run successfully if you do not submit a **correctly formatted zip file**- it has to have the same **names and directory structure as described on page 6 above**. The autograder will also not run if your code **does not compile**, or if you **import libraries** that were not specifically allowed in the instructions.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does. Attend office hours for help or post your questions in Piazza.

Academic Honesty Policy Reminder

Copying partial or whole solutions, obtained from other students or elsewhere, is **academic dishonesty**. Do not share your code with your classmates, and do not use your classmates' code. If you are confused about what constitutes academic dishonesty you should re-read the course policies.

Dev Tips

- Use your debugger! It is very useful in finding exactly where your program stops doing what you want it to do.
- Debug your code in your development environment- that is what it is designed for. Do not use gradescope as a way to develop your code- it will not be helpful.

- Start early! Projects are starting to get a little longer, so if you get stuck you need to make sure you have enough time to seek help.
- Seek help when you get stuck. We have office hours and Piazza chat specifically for you to ask questions when you need assistance. Use public posts as much as possible so we don't have to answer the same question multiple times, only use a private post if you need us to see your code or have questions specific to you.
- Submit to gradescope at least once, even if you aren't completely done. There is a huge difference between a 50% and a 0%. That said, aim for 100%.