1 Overview

In this project, we’ll be implementing a few more sorting algorithms. This project is very similar to Project 1 - we’ll do both a simple sorting, and also traced versions. But I’ve introduced a few small changes:

- Instead of simply having an array of `int`, you will be sorting key/value pairs, represented by a Java class.
- I’ve added a couple of new restrictions about style (mostly, just to help you design good code and avoid duplication).
- The tracing will be a larger part of the automatic grading score.

So for the baseline, go back and review the spec for Project 1. This spec will only discuss the changes for this project.

2 Key/Value Pairs

Often, when we draw pictures of our sorting algorithms, we only show a single value at each node; the key, which we used to sort the data. We do this because it makes the pictures easier to draw. But in practice, we normally have “satellite data” associated with every key.

A data structure with this form is often called a “dictionary,” with the “key” denoting the thing that we sort by, and “value” being the rest of the data. Most of you have already seen this in Python:

```python
myDict = {}
myDict[123] = "foo"
myDict[0] = "bar"
print(myDict[0])        # prints "bar"
```

While Python dictionaries are usually implemented as hash tables, there are lots of ways to implement a dictionary; the term “dictionary” really just refers to the way we access the data (use a key to find a value), not the physical implementation of how that gets done.

In this project, you will be required to make use of a utility class that I’ve provided, `Proj02_DataPair`. This class has only two data fields (no methods); it contains a key, which is an `int`, and a value, which is a `String`.

1
In this project, you will *always* sort the data based on the key *only*; ignore the value entirely. However, when you move a key around as part of the sorting algorithm, always move the `DataPair` object; never move just the integer around. The `TestDriver` class (which I’ve updated for Project 2) will create all of the `DataPair` objects; you will never have to create one yourself.

3 Style Requirements

I’m adding some new style requirements, based on some common mistakes I saw in student code. While these limit your code a bit, the point is to help you write good, well designed code, that is easier to debug and fix.

3.1 Requirement 1: Unified Code

First, the two versions of your code - the “ordinary” version and the “traced” version - must be common code. By “common code,” I mean that they need to be the *same function*, at least for any non-trivial work that you do. (I don’t care if they are separate for tiny things.)

The idea here is that it should be possible to entirely debug your sorting algorithm with the traces turned on; when you turn the traces off, the code should work the *exact same way*. But if you make two functions, this is just too difficult! Instead, the two algorithms should be the *same function* - with a parameter of some sort to indicate whether debug is required.

Does that make the “good path” code more ugly? Yes, I agree with you that it does. But it also makes it a lot easier to debug and fix.

To be honest, the “trace” mode really doesn’t exist only for grading - I was also hoping that it would be an integral part of your development. Write your code with traces included *right from the start* - and use those traces to see if it’s working properly. Then later, once you believe that it’s working, you can turn the traces off. My grading script simply automates exactly this process!

3.2 Requirement 2: Only Implement Something Once

In my solution for Project 1, I wrote a `swap()` function for one of the sort algorithms; it was a helper function inside that first class. That first class also had an function that performed Insertion Sort on a small block of elements.

When I wrote my second algorithm, I just copied the file - and left those two functions (unmodified) in the class. So now I had two copies of each one. Of course, I just kept doing this in each new class.

This was a poor habit. *I am correcting myself - and I’m requiring you to do the same.*

In this project, if you use one algorithm as a subroutine inside another algorithm - like how you use Counting Sort inside of Radix Sort - you *must*
only have one copy of the algorithm in your code. Call that version, when you need it, from other code!

While I won’t be picky about truly tiny bits of code, I figure that anything longer than about 3 to 5 lines of code (which is performed more than once) generally ought to be a function. In particular, the following things should all be functions:

- A `swap()` function, which is used multiple places in Heap Sort.
- A “bubble this item down” function in Heap Sort.

### 3.3 Misc Requirements

Remember that the requirements from Project 1 still apply - such as not allowing global variables.

### 4 Required Algorithms

You must implement all of the following algorithms:

- Heap Sort
- Counting Sort
- Radix Sort

Details will be listed below.

### 5 Things to Look For

This section does not list all of the things that you need to print - but it will give a summary of some key things to look for.

I print out (in trace mode):

- **Counting Sort:**
  What does it print when you pass it different values of the `pos` parameter?

- **Counting Sort:**
  What does it print when you pass it an empty input array?

- **Counting Sort:**
  How many spaces are there, at various points in the line, when it prints out the contents of the array?

---

1. In my solution, I found it handy to define a constant. Even though this is a public static field inside my class, this is OK because it is also final - meaning that it will never change. That’s very different than a variable!
Similarly, there are some algorithmic things to pay attention to:

- What does Heap Sort do when the heap has an odd (or even) number of inputs?
- What does Heap Sort do when the number of elements exactly fills the bottom layer of the tree?
- What does Heap Sort do when you’re bubbling down, and the two children have the same key?

6 Notes on the Algorithms

Although I won’t give you every detail, I wanted to pass along a few hints and pointers.

6.1 Heap Sort

You don’t have to implement a complete Max Heap; you only need to implement the pieces required for Heap Sort.

Although it isn’t required (because you’ll only need it once) it seems to me only sensible that you have a function which converts the array to a heap. Yes, it would be legal to include that in the sort() function - but it just feels wrong, IMHO.

You might want to add helper functions to turn an index into its parent, or into its left and right child. The code is easier to write if people can tell why you are doing some particular math operation!

6.2 Counting Sort

You should assume that every key, in any array you are passed, is in the range 0 to 999,999 (inclusive).

Unlike the other sort functions (which simply take input data), the sort functions for Counting Sort include an int parameter, named pos - which is the digit to sort by. Our Counting Sort will always sort based on decimal digits - so if pos=0, your code should sort based on the one’s column; if pos=2, then you should sort based on the 100’s column; etc. (This is why the keys are limited; the possible values for pos are 0 through 5, inclusive.)

Sorting based on decimal values is bad for performance! It requires that we perform two divisions: first, we divide by some power of 10 to remove the low digits, and then we use modulo (which is also division) to remove the upper digits. (Play around with this on paper until you can figure out how to make this work, given a certain pos parameter.)

Unfortunately, division is very slow. So if you were implementing Counting Sort for a real program, you would almost certainly use bit shifting instead
of division. However, I wanted the traces to be easy to verify by a human - so we’re using the slow-but-pretty method of decimal digits.

**Requirement:** You must use the same array to store both the counts (how many elements are in each bin) and also the indices while you are copying data. Simply re-use the same array for the second purpose. (However, if you want, Java allows you to rename an array without re-allocating it!)

### 6.3 Radix Sort

Your Radix Sort code must call your Counting Sort code. If you implement things correctly, Radix Sort will be very tiny - basically only doing a little debug printing, in addition to calling Counting Sort multiple times.

Since Radix Sort uses Counting Sort under the covers, you may assume that the keys passed to Radix Sort have the same range as mentioned above, in the Counting Sort section.

### 7 Base Code

Download all of the files from the project directory

http://lecturer-russ.appspot.com/classes/cs345/spring18/projects/proj02/

If you want to access any of the files from Lectura, you can also find a mirror of the class website (on any department computer) at:

/home/russell11/cs345_website/

I have provided a (nearly empty) Java file for each of the required classes; fill in the `sort()` and `sort_trace()` functions.

### 8 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- Not use any other files (unless allowed by the project spec) - since our grading script won’t know to use them.
- Follow the spec precisely (don’t change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output exactly! Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I strongly recommend that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!
8.1 Testcases

You can find a set of testcases for this project at
http://lecturer-russ.appspot.com/classes/cs345/spring18/projects/proj02/

In this project, all testing will be done using a single Java class, named
Proj02.TestDriver. You should not change this file - since we’ll be providing
it for you when we do the grading.

For input, the grading script will look for any files named *.dat - each
file needs to be a series of integers, separated by whitespace. (We use Java’s
Scanner class to read the integers into memory.)

The grading script will run each of the required algorithms against each of
the *.dat files that it finds. For each combination, it will run the test twice:
one with the “tracing” turned on, and once with it turned off. If you exactly
match the output of the “example” class - with tracing turned off - then you
will earn 4/5 of the credit for that testcase. However, the last 1/5 of the credit
will require that you also match the example code exactly - with tracing turned
on!

8.2 Other Testcases

For many projects, we will have “secret testcases,” which are additional testcases
that we do not publish until after the solutions have been posted. These may
cover corner cases not covered by the basic testcase, or may simply provide
additional testing. You are encouraged to write testcases of your own,
in order to better test your code. You are also encouraged to share your
testcases on Piazza!

8.3 Automatic Testing

We have provided a testing script (in the same directory), named grade.proj02.
Place this script, all of the testcase files, and your program files in the same
directory. (I recommend that you do this on Lectura, or a similar department
machine. It might also work on your Mac, but no promises!)

9 Turning in Your Solution

Turn in the following files:

    Proj02_HeapSort.java
    Proj02_CountingSort.java
    Proj02_RadixSort.java

You must turn in your code using D2L, using the Assignment folder for this
project. Turn in only your program; do not turn in any testcases or other files.