Programming Project 2
CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development
November 13, 2019

0. Introduction.

For this project, you will implement an efficient algorithm to sort linear singly-linked lists of integers. (We'll discuss algorithms that sort arrays later in the course.) The project is intended to give you some practice working with linked data structures.

1. Theory.

To start, we need a notation for lists of integers. We'll write something like [5, 8, 4, 9, 1, 2, 3, 7, 6] to mean such a list. This list has nine nodes: its first node contains a 5, its second node contains an 8, its third node contains a 4, etc., all the way to its ninth node, which contains a 6. We'll write an empty list of integers as []. Java *does not* use this notation, so don't put it in your programs.

We want to sort lists like these into *nondecreasing order.* That means that the integers in the list don't decrease from left to right. For example, if we sort [2, 3, 1, 1, 2] into nondecreasing order, then we get [1, 1, 2, 2, 3]. Similarly, if we sort [5, 8, 4, 9, 1, 2, 3, 7, 6] into nondecreasing order, then we get [1, 2, 3, 4, 5, 6, 7, 8, 9]. We'll use this list as a running example in the rest of this description.

We also want our sorting algorithm to be as efficient as possible. To explain what we mean by that, let's consider an inefficient sorting algorithm. It might sort a list by traversing it, looking for pairs of adjacent integers that are in the wrong order, like 8 and 4 in the list shown above. Whenever it finds such a pair of integers, it exchanges their positions, so that 4 now comes before 8. The algorithm might repeatedly traverse the list in this way, exchanging integers until all are in the right order. Although this algorithm is simple, it needs $O(n^2)$ comparisons to sort n integers.

The sorting algorithm we'll describe here is more complex, but also more efficient. It works without exchanging adjacent integers, and without traversing lists repeatedly, both of which can make a sorting algorithm slow. Our algorithm needs only $O(n \log n)$ comparisons to sort a list of n integers. The algorithm works in four phases: *testing*, *halving*, *sorting*, and *combining.* We'll describe each phase in detail.

Testing. Suppose that the variable *unsorted* is an unsorted list of integers that we want to sort. We test if *unsorted* has length 0 or 1. If so, then the list is already sorted, so the algorithm simply stops and returns *unsorted* as its result.

Halving. In this phase, *unsorted* has two or more integers. We split *unsorted* into two halves, called *left* and *right,* with approximately equal lengths. The order of integers within *left* and *right* doesn't matter. We start (step 01) with *left* and *right* as empty lists. Then, for odd numbered steps (01, 03, etc.) we delete the first integer from *unsorted* and add it to the front of *left.* For even numbered steps (02, 04, etc.), we delete the first integer from *unsorted* and add it to the front of *right* instead. We continue in this way until *unsorted* becomes empty.

STEP	*left*	*right*	*unsorted*
01	[]	[]	[5, 8, 4, 9, 1, 2, 3, 7, 6]
02	[5]	[]	[8, 4, 9, 1, 2, 3, 7, 6]
03	[5]	[8]	[4, 9, 1, 2, 3, 7, 6]
04	[4, 5]	[8]	[9, 1, 2, 3, 7, 6]
05	[4, 5]	[9, 8]	[1, 2, 3, 7, 6]
06	[1, 4, 5]	[9, 8]	[2, 3, 7, 6]
07	[1, 4, 5]	[2, 9, 8]	[3, 7, 6]
08	[3, 1, 4, 5]	[2, 9, 8]	[7, 6]
09	[3, 1, 4, 5]	[7, 2, 9, 8]	[6]
10	[6, 3, 1, 4, 5]	[7, 2, 9, 8]	[]

We do the halving phase this way because it's easy to delete an integer from the front of a linked list (as in a linked stack), and easy to add a new integer to the front of a linked list (again as in a linked stack). Also, it works without having to know the length of *unsorted.*

Sorting. Now we have two unsorted lists from the halving phase, *left* and *right.* In the example, *left* is [6, 3, 1, 4, 5], and *right* is [7, 2, 9, 8]. We sort *left* and *right* into nondecreasing order, by recursively using the sorting algorithm (the same one we're describing!) on both lists. After that, *left* is [1, 3, 4, 5, 6], and *right* is [2, 7, 8, 9]. The recursion always terminates, because *left* and *right* are shorter than the original list *unsorted,* so they will eventually have only one integer, stopping the algorithm during its testing phase.

Combining. Here we combine *left* and *right* into one sorted list, called *sorted,* which is initially empty (step 01). We look at the first integers from *left* and *right,* choosing the one that's smallest. If the integer from *left* is smallest, then we delete it and add it to the end of *sorted* (as in step 01). If the integer from *right* is smallest, then we delete it and add it to the end of *sorted* (as in step 02). If both integers are equal, then we can do either one. We continue in this way until *left* or *right* becomes empty.

STEP	*sorted*	*left*	*right*
01	[]	[1, 3, 4, 5, 6]	[2, 7, 8, 9]
02	[1]	[3, 4, 5, 6]	[2, 7, 8, 9]

03	[1, 2]	[3, 4, 5, 6]	[7, 8, 9]
04	[1, 2, 3]	[4, 5, 6]	[7, 8, 9]
05	[1, 2, 3, 4]	[5, 6]	[7, 8, 9]
06	[1, 2, 3, 4, 5]	[6]	[7, 8, 9]
07	[1, 2, 3, 4, 5, 6]	[]	[7, 8, 9]

At this point, one of *left* and *right* may not be empty (step 07). If that happens, then we add the entire nonempty list to the end of *sorted.* In the example, we get the list [1, 2, 3, 4, 5, 6, 7, 8, 9]. It's sorted into nondecreasing order, so we stop the sorting algorithm and return this list as its result.

We did the combining phase this way because it's easy to delete an integer from the front of a linked list (as in a linked stack) and also easy to add an integer to the end of a linked list (as in a linked queue). It's also easy to add *left* or *right* to the end of *sorted*.

2. Implementation.

For this project, you must implement the sorting algorithm from the previous section in Java. The file **tests.java** is available on Canvas. It contains Java source code for the class `Sort`, which implements a linear singly-linked list of `int`'s that can be sorted. You must write a sorting method called `sortNodes` for `Sort`. The class `Sort` has the following members, all of which are static.

`private static class Node`

> A nested class. The linked lists that you must sort are made from instances of `Node`. Each instance of `Node` has an `int` slot called `number`. It also has a `Node` slot called `next`, which points to the next `Node` in the list, or to `null`. And it has a constructor that initializes the `number` and `next` slots.

`private static Node makeNodes(int ... numbers)`

> This method takes a series of `int`'s as its arguments. It constructs a linear singly-linked list of `Node`s that contains those `int`'s, then returns it. (It also uses Java syntax that was not discussed in the lectures—but you don't have to know how it works.)

`private static Node sortNodes(Nodes unsorted)`

> **THIS IS THE ONLY METHOD THAT YOU MUST WRITE.** It takes a linear singly-linked list of `Node`s called `unsorted` as its argument, sorts the list in nondecreasing order of its `number` slots, and returns the sorted list.

`private static void writeNodes(Node nodes)`

> This method writes the `int`'s in `nodes`, a linear singly-linked list of `Node`'s, separated by commas and surrounded by square brackets.

`public static void main(String[] args)`

> Test the method `sortNodes` by making a few linear singly-linked lists of `Node`'s, sorting them, and writing them.

As stated above, the method `sortNodes` is all you must write for this project. However, to make the sort algorithm more efficient, and to make the project more interesting, `sortNodes` must follow all of these ~~scary~~ helpful rules:

- You must not use the Java operator `new` in any way! You are not allowed to make new `Node`'s, new instances of classes, new arrays, etc. That means `sortNodes` must work in $O(1)$ space, not counting the memory used for recursion.
- Since you can't make new `Node`'s, `sortNodes` must work only by changing the `next` slots of existing `Node`'s. You are not allowed to change the `number` slots of `Node`'s.
- You must use the algorithm described above in the theory section, with its four phases: *testing, halving, sorting,* and *combining.* To simplify grading, you must use the same local variable names: *left, right, sorted,* and *unsorted.* If you need more local variables, then they can have any names you want.
- Your testing phase must not use a loop to count how many `Node`'s are in *unsorted*. It must work in $O(1)$ time.
- Your halving phase must take $O(u)$ time, where u is the length of *unsorted.* Adding a `Node` to *left* or *right* must take $O(1)$ time. That means the halving phase can't traverse *unsorted* more than once, and can't traverse *left* and *right* at all. It also can't count how many `Node`'s are in *unsorted*. Hint: recall how linked stacks work.
- Your sorting phase must call `sortNodes` recursively to sort *left* and *right.*
- Your combining phase must take $O(l + r)$ time, where l is the length of *left,* and r is the length of *right.* Adding a `Node` to *sorted* must take $O(1)$ time. That means the combining phase can't traverse *left* and *right* more than once, and it can't traverse *sorted* at all. Hint: recall how linked stacks and linked queues work.
- Your method `sortNodes` must work correctly for lists of any length. It must work for lists other than those in `main`. In particular, it must work correctly for the empty list, and it must work correctly for lists with duplicate elements.

- You may write additional private static helper methods, but they must also follow these rules. Your helper methods can have any names you want.

- You can't change any part of the class `Sort`, except its `sortNodes` method. But `main` is an exception to this rule: you can add more tests to show how your code works. Also, `main` is the only method that is allowed to print things.

If your code violates these rules, then you will lose a large number of points. As a result, if you have questions about whether you're following the rules correctly, then you should contact me or the TA's. Here are more hints:

- Draw box-and-arrow diagrams! It helps to use a whiteboard. There are whiteboards available for student use in **Bruininks Hall** and elsewhere on campus. If you can make the algorithm work with box-and-arrow diagrams, then you can make it work with Java code.

- My implementation of `sortNodes` uses about 80 lines of Java code, not using helper functions, not counting comments, and indenting as I do in the lectures. This should give you a very rough idea of how complex `sortNodes` should be. Your code may be shorter or longer than mine, and still be correct.

3. Deliverables.

Unlike the lab assignments, you must work on this project individually, without a partner. Although you may discuss the project in a general way with others, **IT MUST BE WRITTEN ENTIRELY BY YOURSELF, ALONE!** The project is worth 40 points: the testing phase is worth 5 points, the halving phase is worth 15 points, the sorting phase is worth 5 points, and the combining phase is worth 15 points. It will be due in **two weeks,** at **11:55 PM** on **Wednesday, November 27, 2019,** the day before the Thanksgiving holiday.

The TA's will read your Java code in detail, awarding partial credit wherever possible. As a result, you must submit (1) Java source code for the class `Sort` that includes your code for `sortNodes`, and (2) any output produced by its `main` method. You must submit exactly one `.java` file that contains these things. Output from `main` must appear in comments at the end of your file.