PA 3
Due Date: Friday, 22 March 2024 by 11:59 PM

General Guidelines.

The instructions given below describe the required methods in each class. You may need to write additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

In general, you are not allowed to import or use any additional classes in your code without explicit permission from your instructor!

Note: It is okay to use the Math class if you want.

Project Overview

In this project, you must implement some sorting algorithms, which are mostly variations of the sorting algorithms we cover in class. There are 10 sorting algorithms for you to implement, and I have tried to organize them in order of difficulty. They fall into three main categories: (1) algorithms to sort an array (2) algorithms to sort a locality-aware array and (3) algorithms to sort a linked list. I am not providing test cases this time, so it is up to you to make sure that your code works correctly. I am providing some classes that may help when you write your test cases. Each of these algorithms will be worth 6 points, which means you can earn up to 60 points. (It also means that you can skip one and still get full credit.) Keep in mind that your PA points will still be capped at 250.

Before you start coding.

Before you start coding, it is a good idea to make sure you have a current account on lectura and you know your password. You should also go through the process of transferring files to lectura to make sure you know how to do it. The following are resources that may help with this process.

- mac to lectura tutorial
- lectura instructions for pc
- CS Help Desk (in case you need help with account information)
- TAs & instructors (We can walk you through the process in office hours or SI.)

Method Descriptions

Put all your methods in a file called `Sort.java`.
The API is given below.

Sort.java API

Method Signature	Description
static void iterativeMerge(Array A)	Sort Array A using an **iterative** Merge Sort algorithm. You may use a single extra array only for the merging.
static void insertMerge(Array A, int size)	Sort Array A using a combination of Insertion Sort and Merge Sort. Do this **iteratively**. The algorithm should first sort sections (of size *size*) of the Array with insertion sort, then merge the sorted parts together. You may use a single extra array only for the merging.
static void threeWayMerge(Array A)	Sort Array A using a **recursive** 3-way Merge Sort algorithm. This means you divide the array into thirds instead of halves. You may use a single extra array only for the merging.
static void fiveWayQuick(Array A)	Sort Array A using a five-way Quicksort. This means using two pivots and dividing the array into 5 sections: less than the first pivot, equal to the first pivot, in between the two pivots, equal to the second pivot, and greater than the second pivot. This should be done **in place**, so no extra data structures are allowed. But you can do it iteratively or recursively.

static void locSelect(Array A, int d)	In this algorithm, we know that the items in A are no more than d positions away from where they will be after being sorted, which means you can sort it more efficiently. Sort A using a locality-aware version of Selection Sort. This should be done in place, so no other data structures are allowed. Also, the runtime in the worst case should be O(dN).
static void locHeap(Array A, int d)	In this algorithm, we know that the items in A are no more than d positions away from where they will be after being sorted, which means you can sort it more efficiently. Sort A using a locality-aware version of Heapsort. This should be done in place, so no other data structures are allowed. Also, the runtime in the worst case should be O(Nlogd).
static LinkedList mergeSort(LinkedList list)	Sort the list using a **recursive** version of Merge Sort. You must use the LinkedList class as is.
static LinkedList quickSort(LinkedList list)	Sort the list using a **recursive** version of 2-way Quicksort. Don't worry about randomizing the pivot. You must use the LinkedList class as is.
static LinkedList insertionSort(LinkedList list)	Sort the list using a **recursive** version of Insertion Sort. You must use the LinkedList class as is.

`static LinkedList bubbleSort(LinkedList list)`	`Sort the list using a` **`recursive`** `version of Bubble Sort. You must use the LinkedList class as is.`

Provided Classes and Guidelines

- For the Array methods, you must use the provided wrapper class, which counts your accesses.
- For the LinkedList methods, you must use the provided LinkedList and Node classes as they are. You should look at those very carefully before you start those methods.
- You are also provided `ArrayGen.java`, which is useful for creating Arrays for testing (particularly locality-aware ones).
- Most of these methods will not require a huge amount of code, but the actual algorithm can be tricky. I suggest working it out by hand before you start coding.

LinkedList.java

The linked list implementation that you're given is a singly linked list that is built similarly to how linked lists are built in functional languages (which use a lot of recursion). There is no tail pointer. Instead, there are two things that are easy to access: the *head* (which is the first element in the list) and the *tail* (which is *the rest of the list*, so a linked list itself). If you've never used linked lists like this before, this may be difficult to adjust to. Definitely take some time to think through it before you attempt the sorting algorithms.

Testing & Submission Procedure.

Your code must compile and run on lectura, and it is up to you to check this before submitting.

To test your code on lectura:

- Transfer all the files (including test files) to lectura.
- Run *javac *.java* to compile all the java files. (You can also use *javac* with an individual java file to compile just the one file: *javac ArrList.java*)
- Run *java <filename>* to run a specific java file. (Example: *java ArrListTest.java*)

After you are confident that your code compiles and runs properly on lectura, you can submit the required files using the following **turnin** command. Please do not submit any other files than the ones that are listed.

```
turnin csc345pa3 Sort.java
```

Upon successful submission, you will see this message:

Turning in:

Sort.java – ok

All done.

Note: Only properly submitted projects are graded! No projects will be accepted by email or in any other way except as described in this handout. If you are worried about your submission, feel free to ask us to check that it got into the right folder.

In addition to the auto-grading, the following lists items we could potentially take points off for.

- Code does not compile or does not run on lectura (up to 100% of the points).
- Bad Coding Style (up to 5 points) – this is generally only deducted if your code is hard to read, which could be due to bad documentation, lack of helpful comments, bad indentation, repeating code instead of abstracting out methods, writing long methods instead of smaller, more reusable ones, etc.
- Not following directions (up to 100% of the points depending on the situation)
- Late Submission (5 points per day)--see the policy in the syllabus for specifics
- Inefficient code (up to 50% of the points)

Your score will be determined by the tests we do on your code minus any deductions that are applied when your code is manually graded. In addition to late deductions, coding style, and deductions for not following directions, you may also receive deductions for inefficient code.

See the late submission and resubmission policies in the syllabus.

Pseudocode for LinkedList methods

```
proc mergeSort (LinkedList L):
     if L is empty: return L
     if the size of L is 1: return L
     (L1, L2) = halve(L)
     S1 = mergeSort(L1)
     S2 = mergeSort(L2)
     return merge(S1, S2)
end mergeSort

Notes:
```

- *halve* should split the list into two halves–note that this does not have to preserve the original order of the elements
- *merge* should merge the two sorted lists together
- Both *halve* and *merge* could be done either recursively or iteratively, but the overall *mergeSort* should be done recursively.

```
proc quicksort (LinkedList L):
      if L is empty: return L
      if L has one element: return L
      (L1, L2) = pivot(L.head, L.tail)//L.head is the pivot
      S1 = quicksort(L1)
      S2 = quicksort(L2)
      return append(S1, [L.head], S2)
end quicksort
```

Notes:

- *pivot* can be iterative or recursive but should take in an integer and divide the list into two lists where the first list is <= the pivot and the second list is > the pivot
- *append* should append the three given lists together–this can be done with iteration

```
proc insertionSort (LinkedList L):
      if L is empty: return L
      S = insertionSort(L.tail)
      insert(L.head, S)
end insertionSort
```

Notes:
insert takes an element (or node) and a sorted list and inserts the new element (node) into the sorted list–this can be done iteratively or recursively

```
proc bubbleSort (LinkedList L):
      for i from 1 to L.size:
            bubble(L)
      end for
end bubbleSort
```

```
proc bubble (LinkedList L):
      if L is empty: return L
      if L.size = 1: return L
      a = L.head
      b = L.tail.head
      T = L.tail.tail
      if a <= b:
            T = bubble(T.add(b))
            T.add(a)
      else
            T = bubble(T.add(a))
            T.add(b)
      end if
      return T
end bubble
```