Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.ArrayList` for an Array List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files, redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

Sorting

For this assignment you will be coding 5 different sorts: cocktail sort, insertion sort, selection sort, merge sort, and LSD radix sort. You will also be coding the kth select algorithm that follows the quick sort algorithm. In addition to the requirements for each sort, to test for efficiency, we will be looking at the number of comparisons made between elements while grading.

Your implementations must match what was taught in lecture and recitation to receive credit. Implementing a different sort or a different implementation for a sort will receive no credit even if it passes comparison checks.

Comparator

Each method (except radix sort) will take in a comparator and use it to compare the elements of the array in various algorithms described below and in the sorting file. You **must** use this comparator as the number of comparisons performed with it will be used when testing your assignment.

Note that `comparator.compare(`x, y`)` is equivalent to x`.compareTo(`y`)`.

Inplace Sorts

Some of the sorts below are inplace sorts. This means that the items in the array passed in aren't copied over to another array or list. Note that you can still create variables that hold only one item; you cannot create another data structure such as array or list in the method.

Stable Sorts

Some of the sorts below are stable sorts. This means that duplicates should remain in the same relative positions after sorting as they were before sorting.

Cocktail Sort

Cocktail sort should be inplace and stable. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n)$. **Note: Implement cocktail sort with the optimization where it utilizes the last swapped index.** Remembering where you last swapped will enable some optimization for cocktail sort. For example, traversing the array from smaller indices to larger indices, if you remember the index of your last swap, you know after that index, there are only the largest elements in order. Therefore, on the next traversal down the array, you start at the last swapped index, and on the next traversal up the array, you stop at the last swapped index. Make sure that both on the way up and on the way down, you only look at the indices that you do not know are sorted. Do not make extra comparisons.

Example of one pass of cocktail sort with last swapped optimization:
Start of cocktail sort:

1	2	6	5	3	4	7	8	9

Start going up the array:

Compare 1 (at index 0) with 2 (at index 1) and don't swap

1	**2**	6	5	3	4	7	8	9

Compare 2 (at index 1) with 6 (at index 2) and don't swap

1	**2**	**6**	5	3	4	7	8	9

Compare 6 (at index 2) with 5 (at index 3) and **swap**

1	2	**5**	**6**	3	4	7	8	9

Compare 6 (at index 3) with 3 (at index 4) and **swap**

1	2	5	**3**	**6**	4	7	8	9

Compare 6 (at index 4) with 4 (at index 5) and **swap**

1	2	5	3	**4**	**6**	7	8	9

Compare 6 (at index 5) with 7 (at index 6) and don't swap

1	2	5	3	4	**6**	**7**	8	9

Compare 7 (at index 6) with 8 (at index 7) and don't swap

1	2	5	3	4	6	**7**	**8**	9

Compare 8 (at index 7) with 9 (at index 8) and don't swap

1	2	5	3	4	6	7	**8**	**9**

Start going down the array:

Note: Skip over indices 5 - 8 since no swaps occurred there.

Compare 4 (at index 4) with 3 (at index 3) and don't swap

1	2	5	**3**	**4**	6	7	8	9

Compare 3 (at index 3) with 5 (at index 2) and **swap**

1	2	**3**	**5**	4	6	7	8	9

Compare 3 (at index 2) with 2 (at index 1) and don't swap

1	**2**	**3**	5	4	6	7	8	9

Compare 2 (at index 1) with 1 (at index 0) and don't swap

1	**2**	3	5	4	6	7	8	9

Finished one pass of cocktail sort.
Note: Next time going up, skip over indices 0 - 2 since no swaps occurred there.

Insertion Sort

Insertion sort should be inplace and stable. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n)$.

Note that, for this implementation, you should sort from the beginning of the array. This means that after the first pass, index 0 and 1 should be considered sorted. After the second pass, index 0-2 should be considered sorted. After the third pass, index 0-3 should be considered sorted, and so on.

Selection Sort

Selection sort should be inplace. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n^2)$.

Merge Sort

Merge sort should be stable. It should have a worst case running time of $O(n \log n)$ and a best case running time of $O(n \log n)$.

Radix Sort

Radix sort should be stable. It should have a worst case running time of $O(kn)$ and a best case running time of $O(kn)$, where k is the number of digits in the longest number. You will be implementing the least significant digit version of the sort. You will be sorting `int`s. Note that you CANNOT change the `int`s into `String`s at any point in the sort for this exercise. The sort **must** be done in base 10. Also, as per the forbidden statements section, you cannot use anything from the `Math` class besides `Math.abs()`. However, be wary of handling overflow if you use `Math.abs()`!

Kth Select

Kth select should be inplace. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n)$. Your implementation must be randomized as specified in the method's javadocs. Logically, it is similar to a one-sided quick sort. When asked for the kth smallest, you should return what would be at the $k-1$ index if the array was perfectly sorted.

Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in this PDF, and in other various circumstances.

Methods:	
cocktailSort	10pts
insertionSort	10pts
selectionSort	10pts
mergeSort	15pts
lsdRadixSort	15pts
kthSelect	15pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

A note on JUnits

We have provided a **very basic** set of tests for your code, in `SortingStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor does it guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza.

If you need help on running JUnits, there is a guide, available on Canvas under Files, to help you run JUnits on the command line or in IntelliJ.

Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located on Canvas, under Files, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Tim Aveni (tja@gatech.edu) with the subject header of "[CS 1332] CheckStyle XML".

Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs. If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs. Any Javadocs you write must be useful and describe the contract, parameters, and return value of the method; random or useless javadocs added only to appease Checkstyle will lose points.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies not only to comments/javadocs but also things like variable names.

Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. "Error", "BAD THING HAPPENED", and "fail" are not good messages. The name of the exception itself is not a good message.

For example:

Bad: `throw new IndexOutOfBoundsException("Index is out of bounds.");`

Good: `throw new IllegalArgumentException("Cannot insert null data into data structure.");`

Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedNode<Integer>()` instead of `new LinkedNode()`. Using the raw type of the class will result in a penalty.

Forbidden Statements

You may not use these in your code at any time in CS 1332.

- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Thread` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)
- Anything besides `Math.abs()` in the `Math` class (for this homework only)
- `String` class (for this homework only)

If you're not sure on whether you can use something, and it's not mentioned here or anywhere else in the homework files, just ask.

Debug print statements are fine, but nothing should be printed when we run your code. We expect clean runs - printing to the console when we're grading will result in a penalty. If you submit these, we will take off points.

Provided

The following file(s) have been provided to you. There are several, but we've noted the ones to edit.

1. `Sorting.java`

   This is the class in which you will implement the different sorting algorithms. Feel free to add private static helper methods but **do not add any new public methods, new classes, instance variables, or static variables**.

2. `SortingStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `Sorting` class. It is not intended to be exhaustive and does not guarantee any type of grade. **Write your own tests to ensure you cover all edge cases.**

Deliverables

You must submit **all** of the following file(s). Please make sure the filename matches the filename(s) below, and that *only* the following file(s) are present. If you make resubmit, make sure only one copy of the file is present in the submission.

After submitting, double check to make sure it has been submitted on Canvas and then download your uploaded files to a new folder, copy over the support files, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `Sorting.java`