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 17**. 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. For example, do not add `throws` to the method headers since they are not necessary.
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 `ArrayList` 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 are expected to implement all of the methods in this homework. Each unimplemented method will result in a deduction.
8. You must submit your source code, the `.java` files, not the compiled `.class` files.
9. Only the last submission will be graded. Make sure your last submission has **all** required files. Resubmitting will void all previous submissions.
10. 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.

Collaboration Policy

Every student is expected to read, understand and abide by the Georgia Tech Academic Honor Code.

When working on homework assignments, you **may not** directly copy code from any source (other than your own past submissions). You are welcome to collaborate with peers and consult external resources, but you **must** personally write all of the code you submit. **You must list, at the top of each file in your submission, every student with whom you collaborated and every resource you consulted while completing the assignment**.

You may not directly share any files containing assignment code with other students or post your code publicly online. If you wish to store your code online in a personal private repository, you can use Github Enterprise to do this for free.

The only code you may share is JUnit test code on a pinned post on the official course Piazza. Use JUnits from other students at your own risk; **we do not endorse them**. See each assignment's PDF for more details. If you share JUnits, they **must** be shared on the site specified in the Piazza post, and not anywhere else (including a personal GitHub account).

Violators of the collaboration policy for this course will be turned into the Office of Student Integrity.

Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you, and is located on Canvas. It can be found under Files, along with instructions on how to use it. A point is deducted for every style error that occurs. If there is a discrepancy between what you wrote in accordance with good style and the style checker, then address your concerns with the Head TA.

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.’’);`

In addition, you may not use try catch blocks to catch an exception unless you are catching an exception you have explicitly thrown yourself with the `throw new ExceptionName(‘‘Exception Message’’);` syntax (replacing `ExceptionName` and `Exception Message` with the actual exception name and message respectively).

Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedList<Integer>()` instead of `new LinkedList()`. 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.

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code (by any measure), nor do they 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.

Sorting

For this assignment you will be coding 5 different sorts: insertion sort, cocktail shaker sort, merge sort, LSD radix sort, and heap sort. You will also be coding the kth select algorithm that is very similar to 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.

For each of the sorting algorithms, you may assume that the arrays you are sorting will not contain `null`. You should also assume that arrays may contain any number of duplicate elements.

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 and heap 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. See the Java API for details about how the Comparator works and the meaning of the returned value.

Generic Methods

Most of the assignments for this class so far have utilized generics by incorporating them into the class declaration. However, the rest of the assignments will have you implement various algorithms as static methods in a utility class. Thus, the generics from here on will use generic methods instead of generic classes (hence the `<T>` in each of the method headers and javadocs). This also means any helper methods you create will also need to be static with the same `<T>` in the method header. If this sounds confusing, just check out the method headers in the homework and write your helper method headers in the same way.

In-Place Sorts

Some of the sorts below are in-place sorts. This means that the items in the array passed in **should not** get copied over to another data structure. Note that you can still create variables that hold only one item, but you cannot create another data structure such as an array or list in the method.

Stable Sorts

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

Adaptive Sorts

Some of the sorts below are adaptive sorts. This means that the algorithm takes advantage of existing order in the input array by not comparing elements that are already ordered.

Algorithms

Insertion Sort

Insertion sort should be in-place, stable, and adaptive. 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, indices 0 and 1 should be relatively sorted. After the second pass, indices 0-2 should be relatively sorted. After the third pass, indices 0-3 should be relatively sorted, and so on.

Cocktail Sort

Cocktail sort should be in-place, stable, and adaptive. 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. Failure to implement this optimization will result in loss of points.

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.

Merge Sort

Merge sort should be out-of-place, stable, and not adaptive. It should have a worst case running time of $O(n \log n)$ and a best case running time of $O(n \log n)$. When splitting an odd size array, the extra data should go on the **right**.

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 of pivot selection 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 index $k - 1$ if the array was perfectly sorted in ascending order.

LSD Radix Sort

LSD Radix sort should be out-of-place, stable, and not adaptive. 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 `ints`. Note that you CANNOT change the `ints` into `Strings` 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()`!

Heap Sort

Heap sort should be out-of-place, unstable, and not adaptive. It should have a worst case running time of $O(nlogn)$ and a best case running time of $O(nlogn)$. Use `java.util.PriorityQueue` as the heap. Refer to the javadocs for more details. You may also find this heap sort video helpful.

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

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 helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

2. `SortingStudentTest.java`

   This is the test class that contains a set of tests covering the basic algorithms in 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) to the course Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission will be graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present. If you resubmit, be sure only one copy of each file is present in the submission. If there are multiple files, do not zip up the files before submitting; submit them all as separate files.

Once submitted, double check that it has uploaded properly on Gradescope. To do this, download your uploaded file(s) to a new folder, copy over the support file(s), recompile, and run. It is your sole responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `Sorting.java`