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.

Pattern Matching

For this assignment you will be coding 3 different pattern matching algorithms: Boyer-Moore, Knuth-Morris-Pratt, and Brute Force. There is information about all three in the interface and more information about Boyer-Moore and KMP in the book (also under resources on T-Square). If you implement any of the three algorithms in an unexpected manner (i.e. contrary to what the Javadocs and PDF specify), **you may receive a 0**.

For all of the algorithms, make sure you check the simple failure cases as soon as possible. For example, if the pattern is longer than the text, don't do any preprocessing on the pattern/text.

Knuth-Morris-Pratt

Failure Table Construction

The Knuth-Morris-Pratt (KMP) algorithm relies on using the prefix of the pattern to determine how much to shift the pattern by. The algorithm itself uses what is known as the failure table (also called failure function). There are different ways of calculating the failure table, but we are expecting one specific format described below.

For any string `pattern`, have a pointer `i` starting at the first letter, a pointer `j` starting at the second letter, a table called `table` that is the length of the pattern. Then, while `j` is still a valid index within `pattern`:

- If the characters pointed to by `i` and `j` match, then write `i + 1` to index `j` of the table and increment `i` and `j`.
- If the characters pointed to by `i` and `j` do not match:

- If `i` is not at 0, then change `i` to `table[i - 1]`. Do not increment `j` or write any value to the table.
- If `i` is at 0, then write `i` to index `j` of the table. Increment only `j`.

For example, for the string `abacab`, the failure table will be:

a	b	a	c	a	b
0	0	1	0	1	2

For the string `ababac`, the failure table will be:

a	b	a	b	a	c
0	0	1	2	3	0

For the string `abaababa`, the failure table will be:

a	b	a	a	b	a	b	a
0	0	1	1	2	3	2	3

For the string `aaaaaa`, the failure table will be:

a	a	a	a	a	a
0	1	2	3	4	5

Searching Algorithm

For the main searching algorithm, the search acts like a standard brute-force search for the most part, but in the case of a mismatch:

- If the mismatch occurs at index 0 of the pattern, then shift the pattern by 1.
- If the mismatch occurs at index `j` of the pattern and index `i` of the text, then shift the pattern such that index `failure[j-1]` of the pattern lines up with index `i` of the text, where `failure` is the failure table. Then, continue the comparisons at index `i` of the text (or index `failure[j-1]` of the pattern). Do **not** restart at index 0 of the pattern.

In addition, when a match is found, instead of shifting the pattern over by 1 to continue searching for more matches, the pattern should be shifted over by `failure[j-1]`, where `j` is at pattern.length.

CharacterComparator

`CharacterComparator` is a comparator that takes in two characters. This allows you to see how many times you have called `compare()`. We will be looking at the number of times you call `compare()` while grading.

If you do not use the passed in comparator, this will cause tests to fail and will significantly lower your grade on this assignment.

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:	
kmp	20pts
buildFailureTable	10pts
boyerMoore	20pts
buildLastTable	10pts
bruteForce	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 `PatternMatchingStudentTests.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 T-Square under Resources, 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 in T-Square, under Resources, 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 Raymond Ortiz (rortiz9@gatech.edu) with the subject header of "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.

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 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.

- `break` may only be used in switch-case statements
- `continue`
- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes
- Lambda Expressions
- Method References

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 you will edit only one of them.

1. `PatternMatching.java`

   This is the class in which you will implement the different pattern matching 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. `PatternMatchingStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `PatternMatching` 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.**

3. `CharacterComparator.java`

   This is a comparator that will be used to count the number of comparisons used. **You must use this comparator. Do not modify this file.**

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. T-Square does **not** delete files from old uploads; you must do this manually. Failure to do so may result in a penalty.

After submitting, be sure you receive the confirmation email from T-Square, and then download your uploaded files to a new folder, copy over the interfaces, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `PatternMatching.java`