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: Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin-Karp. For all three algorithms, you should find **all** occurrences of the pattern in the text, not just the first match. The occurrences are returned as a list of integers; the list should contain the indices of occurrences in ascending order. There is information about all three algorithms in the javadocs with additional implementation details below. 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 and just return an empty list since there cannot be any occurrences of the pattern in the text.

CharacterComparator

`CharacterComparator` is a comparator that takes in two characters and compares them. This allows you to see how many times you have called `compare()`; besides this functionality, its return values are what you'd expect a properly implemented `compare()` method to return. You **must** use this comparator as the number of times you call `compare()` with it will be used when testing your assignment.

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

You must implement the algorithms as they were taught in class. We are expecting **exact** comparison counts for this homework. If you are getting fewer comparison counts than expected, it means one of two things, either you implemented the algorithm wrong (most likely) or you are using an optimization not taught in the class (unlikely).

Knuth-Morris-Pratt

Failure Table

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). This is an array of length m where each index will correspond to the characters at that index in the pattern. Each index i of the failure table should contain the length of the longest proper (not the entire string) prefix that matches a proper suffix of `pattern[0, ..., i]`. 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, and an array called `table` that is the length of the pattern. First, set index 0 of *table* to 0. 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, if the whole pattern is ever matched, 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. KMP treats a match as a "mismatch" on the character immediately following the match.

Boyer-Moore

Last Occurrence Table

The Boyer-Moore algorithm, similar to KMP, relies on preprocessing the pattern. Before actually searching, the algorithm generates a last occurrence table. The table allows the algorithm to skip sections of the text, resulting in more efficient string searching. The last occurrence table should be a mapping from each character in the alphabet (the set of all characters that may be in the pattern or the text) to the last index the character appears in the pattern. If the character is not in the pattern, then -1 is used as the value, though you should not explicitly add all characters that are not in the pattern into the table.

Searching Algorithm

Key properties of Boyer-Moore include matching characters starting at the end of the pattern, rather than the beginning and skipping along the text in jumps of multiple characters rather than searching every single character in the text.

The shifting rule considers the character in the text at which the comparison process failed (assuming that a failure occurred). If the last occurrence of that character is to the left in the pattern, shift so that the pattern occurrence aligns with the mismatched text occurrence. If the last occurrence of the mismatched character does not occur to the left in the pattern, shift the pattern over by one (to prevent the pattern from moving backwards). In addition, if the mismatched character does not exist in the pattern at all (no value in last table) then pattern shifts completely past this point in the text.

For finding multiple occurrences, if you find a match, shift the pattern over by one and continue searching.

Rabin-Karp

The Rabin-Karp algorithm relies on hashing to perform pattern matching. This algorithm, instead of using a sophisticated shift / skip through the text, uses a hash function to compare the given pattern with substrings of the text. This algorithm exploits the fact that if two strings are equal, their hash values must also be equal. The algorithm essentially reduces down to computing the hash value of the pattern and then looking for substrings of the text with that hash value. Once a substring of the text with that hash value is found, character by character comparisons are required to ensure equality (as two strings with the same hash may not actually be equal).

Note: You must use the exact rolling hash function specified in the javadocs. You are not allowed to use `Math.pow()` for the intial hash calculation, nor are you allowed to use it for updating the text hash. **This is because exponentiating a number is not an $O(1)$ operation, so creating your own custom power method will also not work.**

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	15pts
buildFailureTable	10pts
boyerMoore	15pts
buildLastTable	10pts
rabinKarp	25pts
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 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.

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 may 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)
- `Math.pow()` (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. `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. 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. `PatternMatching.java`