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 11**. 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. You may create helper methods, but any helper method you create should be **private**.
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 consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

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 it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. 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.

Additionally, make sure you include your name, version, user ID, and GT ID in any file submitted to Gradescope.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

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", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

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 any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs

- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, 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.

Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. PatternMatching.java

PatternMatching

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.

Note that for pattern matching, we refer to the text length as n and the pattern length as m.

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). Before actually searching, the algorithm generates a failure table. This is an array of length m where each index will correspond to the substring in the pattern up to that index. Each index i of the failure table should contain the length of the longest proper prefix that matches a proper suffix of `pattern[0, ..., i]`. A proper prefix/suffix does not equal the string itself. There are different ways of calculating the failure table, but we are expecting the 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 so that the pattern at index `failure[j-1]`, where `j` is at pattern.length, aligns with the index after the match in the text. 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. The `getOrDefault()` method from Java's Map will be useful for this.

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 the same hash value. Once a substring of the text with the same hash as the pattern is found, the substring is compared character by character with the pattern 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 is also inefficient.**

Grading

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

- PatternMatching – 90 pts
- Checkstyle – 10 pts

- Total: 100 pts

Provided

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

These are the classes to be implemented. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

1. `PatternMatching.java`

This is a comparator that will be used to count the number of comparisons used. **Do not alter these files.**

1. `CharacterComparator.java`

These are test classes that contain a set of tests covering the basic operations on their respective data structures. 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.**

1. `PatternMatchingStudentTest.java`