Homework 10: Strings

Due: April 7, 11:55 PM

Important

There are a few guidelines you must follow in this homework. If you fail to follow any of the following guidelines you will receive a **0** for the entire assignment.

1. All submitted code must compile under **JDK 7**. This includes unused code, don't submit extra files that don't compile. (Java is backwards compatabile so if it compiles under JDK 6 it *should* compile under JDK 7)
2. Don't include any package declarations in your classes.
3. Don't change any *existing* class headers, constructors, or method signatures. (It is fine to add extra methods and classes)
4. Don't import anything that would trivialize the assignment. (e.g. don't import `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
5. You must submit your source code, the `.java` files, not the compiled `.class` files.

After you submit your files redownload them and run them to make sure they are what you intended to submit. We are not responsible if you submit the wrong files.

Assignment

You will be implementing KMP, Boyer-Moore, and Rabin Karp. See the descriptions of the algorithms in the additional pdfs for how these algorithms work.

Deliverables

You must submit all of the following files.

1. `StringSearches.java`

You may attach them each individually, or submit them in a zip archive.

Boyer Moore

Description

The main idea of this algorithm is to compare from right to left untill you find a mismatch. On a mismatch you shift the needle so that the mismatched character in the haystack lines up with the last occurence of that character in the needle, or shift the needle all the way past the character if it is not in the needle.

This algorithm performs well for large alphabets. This is because in a large alphabet it is more likely for a letter to appear in the haystack, but not the needle. In those cases you can shift by the entire length of the needle.

This is quicker in practice than brute force, but worst case it is still $O(nm)$. There are some optimizations that can make Booyer-Moore $O(n+m)$ in every case, but we will not cover them in this course.

Last Function

The last function determines how far you need to shift when you see each character. It is generated by creating a map for each letter. Then, for every character `c` in the needle, `map[c] = max(needle.length() - needle.lastIndexOf(c) - 1, 1)`. If a character `c` is not in the needle, then `map[c] = needle.length()`.

Algorithm

To run the algorithm line up the first character in the needle to the first character in the haystack. Then start comparing from the end of the needle. When there is a mismatch on character `c` in the haystack, shift the needle to the right by `map[c]` places. Then restart comparing from the end of the needle.

Example

Needle 1: "aardvark"

```
map[a] = max(8 - 1 - 5, 1) = 2
map[r] = max(8 - 1 - 6, 1) = 1
map[d] = max(8 - 1 - 3, 1) = 4
map[v] = max(8 - 1 - 4, 1) = 3
map[k] = max(8 - 1 - 7, 1) = 1
map[*] = 8 (* represents all other characters)
```

Haystack 1: "anteaters are aardvarks"

```
anteaters are aardvarks
aardvark.............          shift by map[r] = 1
.aardvark............          shift by map[s] = 8
.........aardvark......        shift by map[r] = 1
..........aardvark.....        shift by map[d] = 4
..............aardvark.        match at 14
```

Haystack 2: "crush kkyle with aardvarks"

```
crush kkyle with aardvarks
aardvark.................          shift by map[k] = 1
.aardvark................          shift by map[y] = 8
.........aardvark........          shift by map[ ] = 8
.................aardvark.         match at 17
```

Knuth Morris Prat

Description

This algorithm uses a prefix table to maximize the amount of work that can be reused. The table ensure that you never have to go backwards through the haystack when searching for the needle. This greatly speeds up string searching.

Because this algorithm relies on prefixes, a smaller alphabet is more beneficial. This makes sense, because with a smaller alphabet (think DNA sequences) it is more likely that there are more common prefixes and suffixes of substrings within the needle.

The running time of this algorithm in the worst case is $O(n+m)$, while it is truley linear, generally other string matching algorithms, such as Rabin Karp, perform better than it.

Prefix Table

The prefix table tells you where to go in the needle when there is a mismatch. `prefix[i]` should be equal to the length of the longest prefix of the needle that is also a suffix of `needle.subString(0, i)`, only consider prefixes with a length strictly less than `i`. (remember for subString, the second index is exclusive). The following values are defined: `prefix[0] = -1, prefix[1] = 0`.

Algorithm

When running the algorithm, let `i` be the index in the haystack, and let `j` be the index in the needle. If there is a mismatch when `j != 0`, set `j = prefix[j]` and leave `i` the same. If there is a mismatch and `j == 0`, then set `i = i + 1` and leave `j` the same.

Example

Needle 1: "abcdabcabe"

index	0	1	2	3	4	5	6	7	8	9
letter	a	b	c	d	a	b	c	a	b	e
value	-1	0	0	0	0	1	2	3	1	2

Haystack 1: "abcdabcabcdabfabcdabcabe"

```
abcdabcabcdabcfabcdabcabe
abcdabcabe..............          mismatch at j = 9, set j = 2
.......abcdabcabe........          mismatch at j = 7, set j = 3
...........abcdabcabe....          mismatch at j = 3, set j = 0
............abcdabcabe.          mismatch at j = 0, set i = i + 1
.............abcdabcabe          match found at index 15
```

Description

This algorithm leverages the power of hash functions to speed up string searching. The idea is to be able to hash a part of a string in a special way so you can in constant time calculate a hash function for the same string without the first character, and in constant time calculate the hash function for the same string with an extra character at the end. Then to run the algorithm, you find the hashcode of the needle, and compare it to the hashcode of all the needle length substrings of the haystack. This is easy to do because we have constructed our hashfunction in such a way that we can add and remove characters from it in constant time. So it should be possible to "advance" the hash function one position in the haystack in constant time. Then, if the hash functions are ever the same, you do the brute force check to see if the needle actually occurs at that position in the haystack.

The challenge of this algorithm lies in creating the hash function. If we were able to create a perfect hash function, then the running time would be $O(n+m)$. Practically that is the running time of the algorithm, but in the worst case Rabin Karp can take $O(nm)$ time.

Hash Function

The most commonly used hash function treats the substring as a number in a large base. Generally the base chosen is a large prime number. So for the string "apple", let's compute the hash function of substrings of length 4, using the base 1337.

$$\begin{aligned}
\texttt{hash(appl)} &= \texttt{a} \cdot 1337^3 + \texttt{p} \cdot 1337^2 + \texttt{p} \cdot 1337^1 + \texttt{l} \cdot 1337^0 \\
&= 97 \cdot 1337^3 + 112 \cdot 1337^2 + 112 \cdot 1337^1 + 108 \cdot 1337^0 \\
&= 232028393621
\end{aligned}$$

$$\begin{aligned}
\texttt{hash(pple)} &= \left(\texttt{hash(appl)} - \texttt{a} \cdot 1337^3\right) \cdot 1337 + \texttt{e} \cdot 1337^0 \\
&= \left(\texttt{a} \cdot 1337^3 + \texttt{p} \cdot 1337^2 + \texttt{p} \cdot 1337^1 + \texttt{l} \cdot 1337^0 - \texttt{a} \cdot 1337^3\right) \cdot 1337 + \texttt{e} \cdot 1337^0 \\
&= \left(\texttt{p} \cdot 1337^2 + \texttt{p} \cdot 1337^1 + \texttt{l} \cdot 1337^0\right) \cdot 1337 + \texttt{e} \cdot 1337^0 \\
&= \texttt{p} \cdot 1337^3 + \texttt{p} \cdot 1337^2 + \texttt{l} \cdot 1337^1 + \texttt{e} \cdot 1337^0 \\
&= 267878084561
\end{aligned}$$

You can see from the above equations, you can compute `hash(pple)` in constant time if you have `hash(appl)`. You just subtract the first letter, multiply the result by the base, and add the new letter.

Algorithm

The algorithm is very simple once you understand the hash function, simply iterate over the haystack updating the hash function as you go. If you find a substring that has the same hash as the needle you do the brute force comparison to see if it is actually a match.