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
```