Algorithm and Data Structures

Project 3

Pattern Matching Algorithms

PROJECT MEMBERS
Varshitha Yataluru
Neela Ayshwaria Alagappan

Boyer-Moore (BM):

Looking Glass Heuristics:

- ✓ comparing pattern characters to text from right to left

Boyer-Moore algorithm determines the shift size by considering two quantities

Bad-symbol shift

- ✓ It indicates how much to shift based on text's character, c, causing a mismatch
- ✓ If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after k > 0 matches
- ✓ **bad-symbol shift is computed by** $t_1(c) - k$
  where $t_1(c)$ is the entry in the precomputed table used by Horspool's algorithm (see above) and k is the number of matched characters
- ✓ The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$
- ✓ If $t_1(c) - k < 0$
  we obviously do not want to shift the pattern by 0 or a negative number of positions. So, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.
- ✓ So, $d_1 = \max\{t_1(c) - k, 1\}$

Good suffix shift

- ✓ It indicates how much to shift based on matched part (suffix) of the pattern
- ✓ It is applied after $0 < k < m$ last characters were matched.
- ✓ The distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix
- ✓ *Suff(k)* = The ending portion of the pattern (P) as its suffix of size k

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \begin{cases} d_1 & if\ k = 0 \\ \max\{d_1, d_2\} & if\ k > 0 \end{cases}$$

- ■ bad-symbol shift, $d_1 = \max\{t_1(c) - k, 1\}$
- ■ good-suffix shift, d_2

Data Structure: Array

Run time of the Code: 0.0013785362243652344

Sample Input/Output:

```
txt = "CRACK!!  CRACK!!  CRACK!!  CRACK!!"
pat = "CRACK"
```

Pattern occur at shift = 0
Pattern occur at shift = 9

Pattern occur at shift = 18
Pattern occur at shift = 27
Total no of comparison is 20

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- always makes a shift based on the text's character c aligned with <u>the last character</u> in the pattern according to the shift table's entry for c

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):

```
s0   ...             S               ...  sn-1
                     ≠
         B A R B E R
                       B A R B E R
```

Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

```
s0   ...             B             ...  sn-1
                     ≠
         B A R B E R
             B A R B E R
```

Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

```
s0   ...         M E R               ...  sn-1
                 ≠ || ||
           L E A D E R
                       L E A D E R
```

Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m-1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m-1$ characters in the pattern should be aligned with the text's c:

```
s0   . . .              A  R             . . .   sn-1
                        ≠  ||
             R E O R D  E  R
                   R E  O  R D E R
```

Shift table:

Shift sizes can be precomputed by the formula by scanning pattern before search begins and stored in a table called *shift table*.

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \qquad \textbf{(7.1)}$$

Data Structure: Array

Run time of the Code: 0.0013308525085449219

Sample Input/Output:

```
txt = "CRACK!!  CRACK!!  CRACK!!  CRACK!!"
pat = "CRACK"
```

Pattern found at index : 0
Pattern found at index : 9
Pattern found at index : 18
Pattern found at index : 27
Number of comparisions : 26

The KMP Algorithm:

- ✓ Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- ✓ When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- ✓ Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

- ✓ Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- ✓ The **failure function** $\boldsymbol{F(j)}$ is defined as the size of the largest prefix of $\boldsymbol{P}[0..\boldsymbol{j}]$ that is also a suffix of $\boldsymbol{P}[1..\boldsymbol{j}]$
- ✓ Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $\boldsymbol{P}[\boldsymbol{j}] \neq \boldsymbol{T}[\boldsymbol{i}]$ we set $\boldsymbol{j} \leftarrow \boldsymbol{F}(\boldsymbol{j} - 1)$
- ✓ The failure function can be represented by an array and can be computed in $\boldsymbol{O}(\boldsymbol{m})$ time
- ✓ At each iteration of the while-loop, either $\boldsymbol{i}$ increases by one, or the shift amount $\boldsymbol{i} - \boldsymbol{j}$ increases by at least one (observe that $\boldsymbol{F}(\boldsymbol{j} - 1) < \boldsymbol{j}$)
- ✓ Hence, there are no more than $2\boldsymbol{n}$ iterations of the while-loop
- ✓ Thus, KMP's algorithm runs in optimal time $\boldsymbol{O}(\boldsymbol{m} + \boldsymbol{n})$

Computing the Failure Function:

- ✓ The failure function can be represented by an array and can be computed in $\boldsymbol{O}(\boldsymbol{m})$ time
- ✓ The construction is similar to the KMP algorithm itself
- ✓ At each iteration of the while-loop, either
  - $\boldsymbol{i}$ increases by one, or
  - the shift amount $\boldsymbol{i} - \boldsymbol{j}$ increases by at least one (observe that $\boldsymbol{F}(\boldsymbol{j} - 1) < \boldsymbol{j}$)
- ✓ Hence, there are no more than $2\boldsymbol{m}$ iterations of the while-loop

Data Structure: Array

Run time of the Code: 0.0012290477752685547

Sample Input/Output:

```
txt = "CRACK!!  CRACK!!  CRACK!!  CRACK!!"
pat = "CRACK"
```

Found pattern at index 0
Found pattern at index 9
Found pattern at index 18
Found pattern at index 27
Number of comparisons 20

10 Different Texts and Patterns:

```
txt = "CRACK!!  CRACK!!  CRACK!!  CRACK!!"
pat = "CRACK"

txt1 = "Venice is city of water.Venice is one of the beautiful cities across the
world."
pat1 = "Venice"

txt2 = "Venice is city of water.Venice is one of the beautiful cities across the
world."
pat2 = "beautiful"

txt3 = "Venice"
pat3 = "Venice"

txt4 = "CRACK!!  CRACK!!  CRACK!!"
pat4 = "CRA"

txt5 = "CRACK!!  CRACK!!  CRACK!!  CRACK!!"
pat5 = "RACK!!"

txt6 = "Venice is city of water.Venice is one of the beautiful cities across the
world."
pat6 = " world"

txt7 = "onlinetextmessaging"
pat7 = "mess"

txt8 = "indiaismycountry"
pat8 = "india"

txt9 = "Liveincharlotte"
pat9 = "char"
```

Total No. of Comparisions for each algorithm :

	BM	Horspool	KMP
Text	20	26	20
Text1	13	27	12
Text2	9	17	9
Text3	6	6	6
Text4	9	14	12
Text5	25	29	24
Text6	5	20	6
Text7	4	8	4
Text8	5	7	6
Text9	4	7	4

READ ME:

1. Open the command prompt.
2. Navigate to the folder containing the right program.
3. To run the program, enter python <filename>.py. Eg: python kru.py.
4. The output gets displayed to the terminal.