

# CS 5050 - Program #4 - 30 points

## KMP String Matching for binary input

### Part 1

For this problem, write the code to do string matching as described in section 9.1 of the text. Starter code is provided, but feel free to ignore it if it doesn't help. Variable names were selected to match those used in the text. This is common practice when you are citing a specific algorithm.

In order to use subscripting, Strings are converted to character arrays.

This program is a modification of the KMP string matching. If the input is binary in nature (only two symbols are used - such as x/y or 0/1), when you fail to match at x, you know you are looking at a y. (The book describes the problem in C-9.6, but I think their formula is messed up.) I see it this way:

$Bfail(j)$  = the largest  $k < j$  such that the prefix of  $P$  ( $P[0..k]$ ) is a suffix of what you have seen ( $P[0..j-1] P_{opp}[j]$ ).

where  $P_{opp}[j]$  means the opposite character of  $P[j]$ . In other words, when you fail at position  $j$ , you know you have seen  $P[0..j-1] P_{opp}[j]$ . If you know how far to shift the pattern so it matches (the first part of the desired string is what you have just seen), you don't have to check them again. It's like you have a little helper saying to you, "Oh, you failed after matching  $j-1$  characters, just shift the string over  $X$  characters and try again. Since I know what you have just seen, I can tell you what  $X$  should be."

For example, suppose you were trying to match `xxyxyxy` in the input string. The "O" means you were okay in the match. You fail at the "carrot" below. Notice you never look at a character in the text string more than once! That was our goal.

Text String	x	x	y	x	x	y	x	x	y	x	x	y	y	y	y	x	x	x	x	x	x
Pattern	x	x	y	x	x	y	x	y													
Compare	0	0	0	0	0	0	0	^													
Shift				x	x	y	x	x	y	x	y										
Compare									0	0	^										
Shift							x	x	y	x	x	y	x	y							
Compare												0	^								
Shift														x	x	y	x	x	y	x	y
Compare														^							

Wouldn't it be nice if "someone" said, "Just shift the string you are trying to match over and try again (without backtracking)? The "starting over at the beginning of the string" is what we are trying to avoid. The "someone" in our case is  $Bfail[]$ .

Your program should make sure both the pattern string and the big string are binary before using this algorithm.

For the string xxyxyxyxyxy and aabaabaaab, my failure functions looked like:

<b>i</b>	<b>P[i]</b>	<b>Bfail</b>	<b>F</b>
0	x	0	0
1	x	0	1
2	y	2	0
3	x	0	1
4	x	0	2
5	y	2	3
6	x	0	4
7	y	5	0
8	x	0	1
9	y	2	0
10	x	0	1
11	y	2	0
<b>i</b>	<b>p[i]</b>	<b>Bfail</b>	<b>F</b>
0	a	0	0
1	a	0	1
2	b	2	0
3	a	0	1
4	a	0	2
5	b	2	3
6	a	0	4
7	a	0	5
8	a	6	2
9	b	2	3

The way I am using it, bfail[i] says, "If the character at location i in the string fails (is really the opposite character), then bfail[i] characters of the prefix of the pattern string match the last characters of the substring created by using the first i-1 characters of p and changing p[i] to the opposite character."

Similarly, F[i] says, "What is the most number of characters of the pattern that would match, if you shifted it over, if the string terminating at the i<sup>th</sup> position had to match everything before that in the shifted pattern?"

I found it was convenient to have both the original F array (that the book describes) and the previous values of the Bfail array to create the current value of Bfail. Then, it wasn't too terrible. Don't try copying from the text. It will be more understandable and likely quicker to just write your

own code from scratch. You have to earn the right to understand. Asking what YOU want to have happen is the best approach. This is a bit tricky, so make sure it “feels good” before you code it up (or it will never be right). To make things easier, the computation of the f[] function is included in the sample code.

## Hints

I found myself creating a routine *opposite(me)* which when given a character *me*, it returned the opposite character. Thus, if the two characters of the binary string were 'a' and 'b', then *opposite('a')* would return 'b'.

## Input

Allow the user to input both a text string and a pattern string.

## Output

Print the “bfail” table and the f table in a readable format. Using your bfail table, search the text string and indicate how many times the pattern string was found in the text string. (In generating this count, the matches don’t have to be disjoint. For example: xyxyxy occurs twice in xyxyxyxy.)

## Part 2

Generate statistics to confirm the running time. Randomly generate patterns and big strings and see how long the pattern matching takes. Create a table of timings of the following form. Display the table in the output TextArea.

Size (m+n)	Time

A good way to verify complexity experimentally is to create test cases which double in size. For linear algorithms, when the size of the input doubles, the time should approximately double. For log n algorithms, when the size of the input doubles, the time should increase by a constant. For n<sup>2</sup> algorithms, when the size of the input doubles, the time should quadruple. Verify that your algorithm achieves the desired complexity. You may need to have large strings for the experiment to show the desired results. If your results aren't as expected, you aren't done!

If you have trouble getting the time to register anything above zero, feel free to slow down each iteration by a constant amount. One way of doing this is to call a function you create (mydelay) which loops around for a while.

In Java, you can compute elapsed time as follows:

```
// Get current time
long start = System.currentTimeMillis();

// Do something ...

// Get elapsed time in milliseconds
long elapsedTimeMillis = System.currentTimeMillis()-start;
```

ReadMe File (to be submitted with your code)

1. How many hours did you spend on this assignment?
2. Anything the grader needs to know in running your code?