CSC 325 Adv Data Structures

Lecture 8

The KMP Substring Search Algorithm

Java has a method called "indexOf" that will search for a given pattern inside of a string and return the index of where it occurs. Here is some sample code:

```
public class Main {
  public static void main(String[] args) {
    String text = "My rig is a beast";
    String pattern1 = "beast";
    String pattern2 = "machine";
    String pattern3 = "g i";
    System.out.println(pattern1 + " is found at " + text.indexOf(pattern1));
    System.out.println(pattern2 + " is found at " + text.indexOf(pattern2));
    System.out.println(pattern3 + " is found at " + text.indexOf(pattern3));
  }
}
```

If you run the above code, you get:

```
$ javac Main.java
$ java Main
beast is found at 12
machine is found at -1
g i is found at 5
```

Essentially Java is searching the text for the given pattern. If the pattern of characters are found, the index of where the pattern starts is returned, otherwise if the pattern is not found, a -1 is returned from the indexOf function. Here is an illustration of that:

Indices	0	1	2	3	4	**5**	6	7	8	9	10	11	**12**	13	14	15	16	
text	M	y		r	i	**g**		i	s		a		**b**	e	a	s	t	
pattern1													**b**	e	a	s	t	
pattern2																		machine
pattern3						**g**		i										

The same function can be found in multiple other languages. Given a string object called "text" and a pattern called "pattern" (similar as above), the following would work:

Python
```
print (text.index(pattern));
```
C++
```
cout << text.find(pattern);
```
C#
```
Console.WriteLine(text.IndexOf(pattern));
```

How would you write such a function? Well, the obvious brute force approach is to check every character in the text until you find a match. Let's change the text and pattern to something better suited for a demonstration and then run our brute force algorithm:

```
text = "chestercheesecheetos"
pattern = "cheetos"
```

We try to match the first characters of the text with the pattern. If everything matches, we can report that the index found is 0 (i.e. the very first character in text is the first character where the pattern is fully matched). If it is not found, we slide the pattern over by 1 character and try again. We continue until the pattern either fully matches or never matches. If we never match, we return -1.

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern	c	h	e	e	t	o	s													

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	**c**	**h**	**e**	**s**	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern	**c**	**h**	**e**	**e**	t	o	s													

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	**h**	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern		**c**	h	e	e	t	o	s												

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	**e**	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern			**c**	h	e	e	t	o	s											

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	**s**	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern				**c**	h	e	e	t	o	s										

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	**t**	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern					**c**	h	e	e	t	o	s									

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	**e**	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern						**c**	h	e	e	t	o	s								

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	**r**	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern							**c**	h	e	e	t	o	s							

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	**c**	**h**	**e**	**e**	**s**	e	c	h	e	e	t	o	s
pattern								**c**	**h**	**e**	**e**	**t**	o	s						

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	**h**	e	e	s	e	c	h	e	e	t	o	s
pattern									**c**	h	e	e	t	o	s					

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	**e**	e	s	e	c	h	e	e	t	o	s
pattern										**c**	h	e	e	t	o	s				

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	**e**	s	e	c	h	e	e	t	o	s
pattern											**c**	h	e	e	t	o	s			

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	**s**	e	c	h	e	e	t	o	s
pattern												**c**	h	e	e	t	o	s		

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	**e**	c	h	e	e	t	o	s
pattern													**c**	h	e	e	t	o	s	

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	**c**	**h**	**e**	**e**	**t**	**o**	**s**
pattern														**c**	**h**	**e**	**e**	**t**	**o**	**s**

We eventually find the pattern at the end of the text and return a value of 13. While this approach works, it is not very efficient. Think about the time complexity here. In the worst case we will check almost every character in the pattern with every character in the text. Consider the following text and pattern that makes this more clear:

```
text = "aaaaaaaaaaaaaaaaaaaaab"
pattern = "aaab"
```

The "aaa" part of the pattern will be checked over and over for every character in the text until it finally finds the pattern at the end. If we consider n to be the length of the text and m to be the length of the pattern, it should be straightforward to see that this brute force algorithm is `O(n*m)`. This may be the first algorithm you have heard about with a time complexity showing more than just n. Since the size of the pattern as well as the size of the text are both input sizes, we must consider both of them, as they equally impact the runtime performance.

The problem with the brute force algorithm is that we are repeatedly checking the same characters over and over. After we find a mismatch, we only slide over 1 character in the text. This means we could be checking the same characters in the text that we just checked. Consider the following step in the above execution of the algorithm:

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern								c	h	e	e	t	o	s						

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern									c	h	e	e	t	o	s					

Here we match “c”, “h”, “e”, “e” before realizing that “s” and “t” don’t match. At this point we slide the pattern over by 1 and RECHECK “h”, then slide it over and recheck “e”, then slide over and recheck the second “e”. All this rechecking of previously checked characters is wasted time. Can’t we just jump to the point past were we had a mismatch? Let’s see that:

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern								c	h	e	e	t	o	s						

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern													c	h	e	e	t	o	s	

This just saved us a bunch of time. Let’s us this approach and try the entire match again:

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern	c	h	e	e	t	o	s													

…jumped…

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern					c	h	e	e	t	o	s									

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern						c	h	e	e	t	o	s								

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	**r**	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern							**c**	h	e	e	t	o	s							

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	**c**	**h**	**e**	**e**	**s**	e	c	h	e	e	t	o	s
pattern								**c**	**h**	**e**	**e**	**t**	o	s						

…jumped…

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	**e**	c	h	e	e	t	o	s
pattern													**c**	h	e	e	t	o	s	

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	**c**	**h**	**e**	**e**	**t**	**o**	**s**
pattern														**c**	**h**	**e**	**e**	**t**	**o**	**s**

This is much better! We jumped over twice (i.e. sliding the pattern over by more than 1) and saved about 7 comparisons. This is a great modification to our approach. Should we end the research here and tell the world what we've found? Let's just calm down for a second. When you find a better approach to something, your first instinct should be to test the hell out of it to make sure it actually works. Just because it works in this one case does NOT mean it will work in ALL cases (keep this in mind for your programming assignments). To that end, let's try this new approach on another text and pattern:

```
text = "oncononion"
pattern = "onion"
```

Indices	0	1	2	3	4	5	6	7	8	9
text	**o**	**n**	**c**	o	n	o	n	i	o	n
pattern	**o**	**n**	**i**	o	n					

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	**o**	**n**	**o**	n	i	o	n
pattern				**o**	**n**	**i**	o	n		

Indices	0	1	2	3	4	5	6	7	8	9	
text	o	n	c	o	n	o	**n**	i	o	n	
pattern							**o**	n	i	o	n

With our new algorithm, we can conclude that "onion" is not found in "oncononion". Ok, we are ready to tell the world… wait a minute… that answer is wrong! Onion is definitely found in oncon**onion**. Our algorithm failed because it jumped over where the pattern would have actually been found. Hmmm, well what if we

don't jump as far? What if we jump to the character that the mismatch occurred (instead of past it)? Let's try that:

Indices	0	1	2	3	4	5	6	7	8	9
text	**o**	**n**	**c**	o	n	o	n	i	o	n
pattern	**o**	**n**	**i**	o	n					

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	**c**	o	n	o	n	i	o	n
pattern			**o**	n	i	o	n			

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	**o**	**n**	**o**	n	i	o	n
pattern				**o**	**n**	**i**	o	n		

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	o	n	**o**	**n**	**i**	**o**	**n**
pattern						**o**	**n**	**l**	**o**	**n**

Ok, that fixed it. Time to call it a day. But just in case, let's try one more example and see if this new addition still works:

```
text = "aaaaaaaab"
pattern = "aaab"
```

Indices	0	1	2	3	4	5	6	7	8
text	**a**	**a**	**a**	**a**	a	a	a	a	b
pattern	**a**	**a**	**a**	**b**					

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	**a**	**a**	**a**	**a**	a	b
pattern				**a**	**a**	**a**	**b**		

Indices	0	1	2	3	4	5	6	7	8	
text	a	a	a	a	a	a	**a**	**a**	**b**	
pattern							**a**	**a**	**a**	b

Well damn. We jumped too far again. Clearly "aaab" is found in "aaaaaaaab" but our algorithm says it isn't. We keep jumping too far. But if we don't jump at all, then we go back to the brute force approach. If we want to speed up this algorithm, we need to jump, but we need a more intelligent way to determine how far to jump. If an algorithm that finds a pattern in a given text doesn't work for all patterns and texts, then it isn't very useful, so we can't trade speed for accuracy. At the end of the day, it has to work AND be fast. At this

point you may feel like this is an unsolvable problem and that the brute force approach with no jumping is the only approach that actually works in all cases. But don't give up just yet!

The KMP algorithm (named after its inventors Knuth, Morris and Pratt) is a substring matching algorithm (the same problem we have been trying to attack). It goes off of the idea of jumping forward as much as possible when we find a mismatch, but it is intelligent enough to figure out how far to jump without missing a potential match. So we jump just far enough each time. Far enough to save runtime but not too far as to miss a match. It does this by looking at the beginning and ending characters in the pattern. If there are parts of the pattern that show up in the beginning and end for a given section of the pattern, we can use this information to figure out how far to jump. After a jump we may not be starting out with the first character in the pattern. The goal is never to compare characters that we know aren't a match. So after a jump, we may be looking at index 0, 1, 2, 3, or some other index in the pattern.

To figure out how far to jump and what character we should be comparing, let's build our LPS table. LPS stands for "Longest Prefix Suffix" and it gives us information on the longest "proper" prefix that is also a suffix in our pattern. The term "proper" implies that the prefix cannot contain the entire list of characters. For example, the characters "nono" has a prefix of "no" that also matches its suffix of "no", i.e. "nono" where prefix is green and suffix is red. Sometimes these overlap, as in the characters "aaa". The proper prefix "aa" (first two characters) is also found as a suffix "aa" (last two characters) with one of the "a"s counting twice. Since we are interested in "proper" prefixes, we can't count the entire pattern of "aaa" as a prefix. For KMP to work, we need to know the longest proper prefix that is also a suffix for every part of the pattern (i.e. the first character, the first two characters, the first three characters, and so on). We are only interested in a number that tells us the length of the prefix. Let's build the LPS table for "onion":

pattern	o	n	i	o	n
longest prefix	0	0	0	1	2

For the first character only, "o", this is trivially zero since there aren't any proper prefixes (remember that we can't consider all the characters as a "proper" prefix). For "on" this is also zero as the prefix "o" is not the same as the suffix "n". For "oni" it is still zero. But for "onio" it is one, since "o" appears in the beginning (prefix) and end (suffix). For the entire pattern "onion" we get two, since "on" appears as a prefix and a suffix and is the longest prefix that matches the suffix. Now let's use this table to perform pattern matching on the text = "`oncononion`":

Indices	0	1	2	3	4	5	6	7	8	9
text	**o**	**n**	**c**	o	n	o	n	i	o	n
pattern	**o**	**n**	**i**	o	n					

We fail to match "c" and "i", so we need to shift/jump the pattern over. However, we already know that moving the pattern is a delicate process. Move it too much and we might skip over a part of the text that would have fully matched the pattern. Move it too little (for example, 1 spot) and we risk doing too much work thereby bringing down efficiency to that of the brute force algorithm. Let's use our LPS table to figure out how much to shift the pattern by:

pattern	o	n	i	o	n
longest prefix	0	0	0	1	2

The mismatch occurred at this character, “i", so we look to the left of that:

pattern	o	n	i	o	n
longest prefix	0	0	0	1	2

The number here is zero, telling us we should start back at index 0 with the pattern. So, we jump the pattern over to where index 0 of the pattern aligns with the character we failed to match, and then continue attempting to match:

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	o	n	o	n	i	o	n
pattern			o	n	i	o	n			

Any time the first character of the pattern doesn’t match, we just shift over by one, same as in the brute force algorithm:

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	o	n	o	n	i	o	n
pattern				o	n	i	o	n		

We failed to match at “i”. We look back to “n”, refer to the LPS table to see a zero, then reset the pattern back to index zero and align it with the character in the text that mismatched.

Indices	0	1	2	3	4	5	6	7	8	9
text	o	n	c	o	n	o	n	i	o	n
pattern						o	n	i	o	n

Looks like we jumped just enough each time. But let’s see if this jumping adapts to different patterns. It needs to jump by different amounts in our “aaab” example. First, here is our LPS table:

pattern	a	a	a	b
longest prefix	0	1	2	0

Let’s use this for our substring matching:

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	a	a	a	a	b
pattern	a	a	a	b					

Since “b” doesn’t match, we look one character to the left at the last “a” in the pattern. Looking up this “a” in the LPS table shows a value of 2. This means we need to continue attempting to match at index 2 in the pattern. So we shift the pattern over until index 2 of the pattern is aligned with the mismatched character:

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	**a**	a	a	a	a	b
pattern		a	a	**a**	b				

We just mismatched the fourth “a” in the text, and according to our LPS table, we should align that fourth “a” with the character at index 2 (the value from the LPS table) in the pattern. We continue attempting to match here. Note that we are NOT reevaluating the first two characters in the pattern. The LPS table tells us those already match, so double checking them would be wasteful. So even though we only jumped by one character, we start comparing at index 2 of the pattern (highlighted above). Let’s continue matching:

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	**a**	**a**	a	a	a	b
pattern		a	a	**a**	**b**				

We failed to match at “b”. Moving to the left in the pattern and looking up that corresponding position in the LPS table, we see we need to align the pattern with index 2 (same as we did before):

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	**a**	a	a	a	b
pattern			a	a	**a**	b			

We continue with this until reaching the end. Notice how unlike the brute force approach, we are not reevaluating the first two “a”s in the pattern. This saves us time:

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	**a**	**a**	a	a	b
pattern			a	a	**a**	**b**			

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	a	**a**	**a**	a	b
pattern				a	a	**a**	**b**		

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	a	a	**a**	**a**	b
pattern					a	a	**a**	**b**	

Indices	0	1	2	3	4	5	6	7	8
text	a	a	a	a	a	a	a	**a**	**b**
pattern						a	a	**a**	**b**

We found the pattern. Again I want to stress that even though we only jumped by 1 each time, we did not recheck the first two a’s in the pattern and therefore we did less work than in the brute force approach.

What about if the pattern doesn't have any prefix and suffix matches, i.e. the LPS table is all zeros. Let's examine the original example:

```
text = "chestercheesecheetos"
pattern = "cheetos"
```

LPS table:

pattern	c	h	e	e	t	o	s
longest prefix	0	0	0	0	0	0	0

Substring matching:

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	**c**	**h**	**e**	**s**	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern	**c**	**h**	**e**	**e**	t	o	s													

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	**s**	t	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern				**c**	h	e	e	t	o	s										

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	**t**	e	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern					**c**	h	e	e	t	o	s									

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	**e**	r	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern						**c**	h	e	e	t	o	s								

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	**r**	c	h	e	e	s	e	c	h	e	e	t	o	s
pattern							**c**	h	e	e	t	o	s							

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	**c**	**h**	**e**	**e**	**s**	e	c	h	e	e	t	o	s
pattern								**c**	**h**	**e**	**e**	**t**	o	s						

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	**s**	e	c	h	e	e	t	o	s
pattern												**c**	h	e	e	t	o	s		

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	**e**	c	h	e	e	t	o	s
pattern													**c**	h	e	e	t	o	s	

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
text	c	h	e	s	t	e	r	c	h	e	e	s	e	**c**	**h**	**e**	**e**	**t**	**o**	**s**
pattern														**c**	**h**	**e**	**e**	**t**	**o**	**s**

Even though we just have zeros in our LPS table, we still save time by jumping the pattern forward.

Now, to be clear, the “shifting” and “jumping” of the pattern doesn’t visually happen with the pattern. We can mimic this by keeping an index for the current character in the text to examine and a different index for the current character in the pattern to examine. By changing these indices, we can mimic a shift/jump. So while the physical moving of the pattern works well for explaining and visualizing the process, when implemented this is merely a change to a variable keeping track of an index in both the text and the pattern.