Practical 7: Advanced Sorting Algorithms

What am I doing today?
String searching is a very important problem in computer science. When we do search for a string in our IDE, Word or a database, pattern searching algorithms are used to show the search results. The goal of string searching is to locate a specific text pattern within a larger body of text. As with the previous algorithms we have considered, we care about algorithms that are fast and efficient.

This week's practical focuses on:

1. Implement a bruteforce substring search algorithm
2. Implement a version of Knuth-Morris-Pratt algorithm
3. Assess the performance difference between the two algorithms with different inputs

Instructions
Try all the questions. Ask for help from the demonstrators if you get stuck.

Algorithmic Development

Part 1

Let's start by implementing a brute force algorithm that searches a string for a pattern (using the pseudo-code below).

First implement Brute Force algorithm:

Given a text input (text [0..n-1] and a pattern (pat[0-m-1]), write a function that searches through the text input and prints all occurance of the pattern.

Assume n > m

Steps:

1. The brute-force algorithm works by starting at the beginning of the string and compare each character of your pattern against the subsequent characters in the string
2. Once your algorithm finishes checking the first pattern then increment the pointer to the next character in the string and start the process again
3. Your algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the pattern.

Pseudo-code for Brute Force substring search:

```
int n = txt.length();
Int m = pat.length();
for ( pos = 0; pos < n-m; pos++ )
   {
      if ( T[pos..pos+(m-1)] == P )
      {
         return(pos);       // Found match....
      }
   }
```

Starter code: https://drive.google.com/file/d/18eMRe_wNGZIAqMU53_CPkBVzkWaIaMNi/view?usp=sharing

Step 2 Knuth-Morris-Pratt Algorithm
As we saw in our lecture, the brute force approach doesn't work well when there are many false positives - several matching characters in the search text followed by a mis-match close to the end of the pattern search.

KMP improves upon this by taking advantage of prior knowledge about the text being searched. The key idea then behind KMP is that if we already have matched the first **x characters** of the pattern and then we encounter a mismatch, we don't necessarily have to move to the very next character in the search text. We might be able to take advantage of our knowledge of the pattern we're looking for and the characters we've already encountered in the search text to jump ahead a little.

KMP has two efficiencies by taking advantage of prior knowledge:

1) We can skip some iterations for which we know no match is possible and
2) We can also skip some iterations in the inner loop

Pseudo-code
In your code example, the table creation pre-processing method is implemented for you, your job is to complete the KMP comparison method and ensure the whole program performs as expected.

step 1 pre-processing: In the *preprocessing* stage the algorithm computes an array the size of M that identifies how many characters skips can be made.

step 2 searching algorithm: where the brute force approach compares each char in the pattern to those in the search string, with KMP we use the value from lps[] to decide what the next characters to be matched are (or put another way to tell us how many character compares we can skip).

Starter code:
https://drive.google.com/file/d/11GLc3UhBx9cSlXWKgUc1pkdxEorS2DBV/view?usp=sharing

Pseudo-code:

```
int M = length of pattern
int N = length of text

// create lps[] that will hold the longest prefix suffix values for pattern
lps array = new[M];

// create an index for the pattern
int j = 0;

// Preprocess the pattern (calculate lps  array)
computeLPSArray(pattern, length of pattern, lps array)
```

```
//create an index for the string
int i = 0

while (i < N) {
 if (pattern.charAt(j) == text.charAt(i)) {
//increment indexes
}

if (j == M) {
//printout the index of a match

//compute the jump
 }

// mismatch after j matches
else if (i < N && pat.charAt(j) != txt.charAt(i)) {
// Do not match lps[0..lps[j-1]] characters, they will match anyway
// work out how many jumps (if any) you can perform

}
}
```

Step 3 Benchmark the performance of your two algorithms

Modify your algorithms to take different inputs (either within your program or by reading in text files). Run your algorithm looking for patterns in various sizes of texts. Observe, note and graph the performance of your algorithms.

Q. 1 What would you say the complexity of the Brute Force substring search algorithm is?
O(n) → it roughly takes the whole string length. Brute Force is dependent on the length of the string and the length of the pattern, because it will look (stringLength - patternLength) times through. Worst case scenario, the pattern length = 1 and string length = n, so the time complexity is (n-1) which is n.

Q. 2 What would you say the complexity of the KMP algorithm is?
O(n) → similarly, in the worst case scenario, the pattern is not found and there KMP does not provide any optimizations. It cannot jump ahead as parts of the pattern are found and then no longer found.