String Searching

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.

Implementation

Brute Force Approach

If there's a will (and lots of time) there's a way! Often when designing an algoriothm, the best approach is to start simple and get a solution (something, something,"*premature optimization is the root of all evil*"), because only then will you know all the different aspects of the problem. This approach is not always elegant and usual is terribly slow.

With this in mind, let's write a method to find a given pattern within a piece of text. The outline of our brute-force approach is as follows:

1. Starting from the beginning of the input string, compare each character of the pattern against subsequent characters of the input.
2. Once it finishes checking the first character it repeats for the following characters.

Pseudocode

```
bruteForceSearch(string, pattern)
    n = string.length
    m = pattern.length

    for i in range 0..(n-m)
        if string[i..i+(m-1)] == pattern
            return true

    return false
```

The pseudocode is a simplified version of the outline where we can check slices of our input string against the pattern.

Java

Let's convert the outline and the pseudocode to actual Java code.

```
public static boolean bruteForceSearch(final String input, final String pattern) {}
```

Our method takes in two strings, an input string and a pattern string and we return a boolean to indicate whether or not the pattern is contained within the input. **Note** that since we are returning a bool we are not telling the end-user where this pattern is in the string or how many times the pattern occurs. This is more slightly more advanced and beyond the scope of our brute force approach.

```
public static boolean bruteForceSearch(final String input, final String pattern) {
    final int n = input.length();
    final int m = pattern.length();

    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (input.charAt(i + j) != pattern.charAt(j)) {
                break;
            }
        }
        if (j == m) {
            return true;
        }
    }

    return false;
}
```

We can then test our implementation with the following:

```
public static void main(final String[] args) {
    final String input = "ACAADAABA";
    final String pattern = "AABA";
    System.out.println(input + " contains " + pattern + " : " +
bruteForceSearch(input, pattern));
}
```

We should see the following output

```
ACAADAABA contains AABA : true
```

Brute Force Complexity

While the brute force algorithm requires no preprocessing phase, it still needs to check whether the pattern you are searching for starts at all of the positions in the search string between `0` and `n-m`. Then after each try, you shift the search one position to the right and compare the pattern to the string again.

The worst case time complexity of a brute force approach to the substring search problem is `O(m*n)`. It actually is requires `O(m*(n-m+1))` in the worst case, so you could call it a `O(m(n-m))` algorithm. However since Big-O notation is an upper bound and mn>= m(n-m) this is correct and no bother is treated by the simplification since typically you'd expect the length of the search string to be proportional to the pattern.

A Better Approach

Knuth-Morris-Prath Algorithm

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 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 there are no match is possible.
2. We can also skip some iterations in the inner loop.

```
kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may
be returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

There are two main steps to the algorithm:

1. **Pre-processing**: In this stage the algorithm computes an array the size of `M` that identifies how many characters skips can be made.
2. **Searching**: Where the brute force approach compares each character in the pattern to those in the input string, the KMP algorithm uses the `lps[]` to decide what the next characters to be matched are (i.e. tell us how many characters we can skip).

Java Implementation

```java
public static boolean kmpSearch(final String input, final String pattern) {}
```

We have the same signature as previously.

```
static void computeLPSArray(final String pattern, final int m, int[] lps) {
    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0
    // the loop calculates lps[i] for i = 1 to M-1

    while (i < m) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar to search step.
            if (len != 0) {
                // Also, note that we do not increment i here
                len = lps[len - 1];
            } else {
                // if (len == 0)
                lps[i] = len;
                i++;
            }
        }
    }
}
```

```
static boolean KMPSearch(String input, String pattern) {
    int M = pattern.length();
    int N = input.length();
    // create lps[] that will hold the longest
    // prefix suffix values for pattern
    int[] lps = new int[M];
    int j = 0; // index for pat[]
    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pattern, M, lps);
    int i = 0; // index for txt[]


    while (i < N) {
        if (pattern.charAt(j) == input.charAt(i)) {
            j++;
            i++;
        }
        if (j == M) {
            return true;
        }
        // mismatch after j matches
        else if (i < N && pattern.charAt(j) != input.charAt(i)) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i = i + 1;
            }
        }
    }
    return false;
}
```

We can test our implementation using the following:

```
public static void main(String args[]) {
    final String input = "ABABDABACDABABCABAB";
    final String pattern = "ABABCABAB";
    System.out.println(input + " contains " + pattern + " : " + KMPSearch(input,
pattern));
}
```

With an expected output of:

```
ABABDABACDABABCABAB contains ABABCABAB : true
```

Try extending the above code to match a pattern as many times as possible and also limiting it to only `n` mathches

KMP Complexity

KMP doesn't work well where there are many matching characters in the pattern. However, KMP's approach of pre-processing the pattern to work out how many jumps are possible improves the worst case complexity from the brute force algorithm to `O(n)`.