Theory: Searching a substring in Java

24 minutes 0 / 5 problems solved

Problems connected with substring searching often arise in programming. For example, developers that work on a text editor should provide functionality allowing users to search for a substring in text. Or, engineers who implement a browser should allow users to search for a particular word on a page.

There exist several substring searching algorithms, each having its own pros and cons. In this topic, we will consider one of the simplest substring searching algorithms and will see how this algorithm can be implemented in Java.

Current topic:

Searching a substring in Java

Topic depends on:

✓ Searching a substring

✗ Algorithms in Java

Topic is required for:

Knuth-Morris-Pratt algorithm in Java

Rabin-Karp algorithm in Java

§1. The simplest algorithm

Below is an implementation of the simplest substring searching algorithm in Java:

```
public static boolean containsPattern(String text, String pattern) {
    if (text.length() < pattern.length()) {
        return false;
    }

    for (int i = 0; i < text.length() - pattern.length() + 1; i++) {
        boolean patternIsFound = true;

        for (int j = 0; j < pattern.length(); j++) {

            if (text.charAt(i + j) != pattern.charAt(j)) {

                patternIsFound = false;

                break;

            }

        }



        if (patternIsFound) {

            return true;

        }

    }



    return false;

}
```

Table of contents:

↑ Searching a substring in Java

§1. The simplest algorithm

§2. Examples

§3. Summary

Feedback & Comments

The method `containsPattern` takes two strings named `text` and `pattern` as arguments and returns `true` if the `pattern` is contained as a substring in the `text` and `false` otherwise.

1. First, we check if the length of the `text` is less than the length of the `pattern`. If it's so, we should return `false` since in this case there is no `pattern` in the `text`.
2. Then, we start moving along the text and compare the corresponding symbols of the `pattern` and the `text`. At the beginning of each iteration, the `pattern` is considered to be found. If there is a mismatch in the corresponding symbols, we set the flag `patternIsFound` to `false` and break the iteration. If at the end of some iteration the flag is not

changed, we should return `true`, since this means that the `pattern` is found.
3. If none of the iterations was successful, we need to return `false` indicating that there is no `pattern` in the `text`.

§2. Examples

Below are several examples of how to use the method:

```
containsPattern("ACBACAD", "ACA"); // true
containsPattern("Substring", "string"); // true
containsPattern("Hello, world!", "Hi!"); // false
```

And here is some corner cases:

```
containsPattern("", ""); // true
containsPattern("abc", ""); // true
containsPattern("abc", "abc"); // true
containsPattern("abc", "abcd"); // false
```

Note that the `String` class in Java has a similar method for strings called `contains`. Below is an example of how this method works:

```
String s = "Hello";
s.contains("ello"); // true
```

§3. Summary

In this topic, we have learned how the simplest substring searching algorithm can be implemented in Java and have considered several examples of how to use the implemented method. Being simple in understanding and implementing, this algorithm, however, has a disadvantage: for a pattern with length n and a text with length m, its time complexity is $O(n \cdot m)$, which might be too slow for large strings. In the following topics, we will consider other substring searching algorithms more efficient than the described one.