HSE DSBA - Algorithms and Data Structures

Seminar Programming Test - String Matching (November 2023)

Consider the class `StringFinder` that takes as argument a text and a pattern, and it allows to find occurrences of the pattern into different lines of the text. Assume that the text represents a book paragraph, consisting of lines separated by the end-of-line character.

This test has 4 problems: Problem 1 must be solved first. After that, Problems 2, 3 and 4 can be solved in any order of preference.

Problem 1. Code the `StringFinder` constructor, `next()`, `hasNext()`, and `count()` functions [5pts]

`StringFinder(`
`const std::string& text,`
`const std::string& pattern)`
	Class constructor. Receives a text (paragraph with multiple lines) and a pattern. Used to initialize class members of the class, and to execute other code that you consider helpful.
	
	HINT #1: The class StringFinder is used to find occurrences of the pattern into different lines of the text. So, you could first split the text into lines, separated by the end-of-line char, and put all lines into a vector.
	For each line #i in the vector you may call a string matching algorithm to find all positions where the pattern occurs in that line. Finally, you could save this information, so that the user of this class can iterate and query it later.

`std::pair<int, int> next()`	Returns a pair representing the next occurrence K of the pattern in the text: The `first` element in the pair is the line number, and the `second` element is the position in that line where the pattern occurs. As an effect, the function must make the class StringFinder to point to the next K + 1 occurrence of the pattern, so that a following call of next() will return the K + 1 occurrence (if exists). **HINT #2:** To keep track of which occurrence to return each time, you could have an index or iterator variable as a member of the class StringFinder. Initially, this iterator can point to the first occurrence.
`bool hasNext()`	Returns true if there is a next occurrence. Otherwise, it returns false.
`int count()`	Returns the number of occurrences of the given pattern in ALL lines of the text.

Problem 2. Code the `printLines()` function [2pts]

`void printLines()`	Prints in the standard output (via std::cout) all lines (separating them by end-of-line char) where the pattern occurs at least once. If the pattern occurs more than 1 time in the same line, the line is printed only once.

Problem 3. Code the `last()`, `hasPrevious()`, and `previous()` functions [1pt]

`void last()`	Makes the StringFinder to point to the last occurrence of the pattern in the text.
`std::pair<int, int> previous()`	Returns a pair representing the occurrence K of the pattern in the text currently pointed by StringFInder: The `first` element in the pair is the line number, and the `second` element is the position in that line where the pattern occurs. As an effect, the function must make the class StringFinder to point to the previous K - 1 occurrence of the pattern, so that a following call of next() will return the K - 1 occurrence (if exists).
`bool hasPrevious()`	Returns true if there is a previous occurrence of the pattern in the text. Otherwise, it returns false.

HINT #3: Functions in Problem 3 are very similar to the functions next() and hasNext() in Problem 1, with the difference that previous() and hasPrevious() are used to iterate through occurrences in reverse order.

Problem 4. Code the constructor that allows to make optional CASE SENSITIVE comparison [2pts]

`StringFinder(`
`const std::string& text,`
`const std::string& pattern,`
`bool caseSensitive)`
	The behavior of this constructor should be identical to the StringFinder constructor coded in Problem 1, with the difference that:
	
	If `caseSensitive` is `false` then the string matching algorithm will not distinguish between uppercase and lowercase letters (for example, 'A' will be equal to 'a') when comparing each line with the pattern.
	If `caseSensitive` is `true` , then the string matching algorithm runs identical as in Problem 1.

Examples: The following code are examples of how functions of StringFinder will be called by a user with following the `TEXT`:

```
aabbaacdf
bbaacbbaa
aa bb aa
bbccafdf
aa
```

Example:	Output:

`// Example for Problem 1`
 	
`std::string pattern = "aa";`
`StringFinder stringFinder(TEXT, pattern);`
 	
`while (stringFinder.hasNext())`
`{`
`  auto [line, index] = stringFinder.next();`
`  std::cout << line << ' ' << index << std::endl;`
`}`
`std::cout << stringFinder.count() << std::endl;`
	`0 0`
	`0 4`
	`1 2`
	`1 7`
	`2 0`
	`2 6`
	`4 0`
	`7`

`// Example for Problem 2`
 	
`std::string pattern = "aa";`
`StringFinder stringFinder(TEXT, pattern);`
`stringFinder.printLines();`
	`aabbaacdf`
	`bbaacbbaa`
	`aa bb aa`
	`aa`

`// Example for Problem 3`
 	
`std::string pattern = "Cat";`
 	
`StringFinder stringFinder(TEXT, pattern);`
 	
`stringFinder.last();`
 	
`while (stringFinder.hasPrevious())`
`{`
`  auto [line, index] = stringFinder.previous();`
`  std::cout << line << ' ' << index << std::endl;`
`}`
	`4 0`
	`2 6`
	`2 0`
	`1 7`
	`1 2`
	`0 4`
	`0 0`

`// Example for Problem 4 (the same as Problem 1 with the following changes)`
 	
`std::string pattern = "AA";`
`StringFinder stringFinder(TEXT, pattern, false);`
	Same output of Example for Problem 1.