PWr Week 13-15
Data Structures and Algorithms,
Laboratory – **List 12**

Introduction

In this list it will be used very long `String` as an input parameter. Because of that you have to read line by line from an input stream and connect them together. But the '+' operation for `String` class is very complex, because when you do instruction like follow:

```
string=string+nextLine;
```

every time it is created an array of character with the length which is a sum of the lengths of `string` and `nextLine`, then all characters from `string` are copied to the new array and after them all characters from `nextLine`. So if every line has `k` characters and we have `l` such a lines, we need `k+2k+3k+…+l*k=(l+1)*l*k/2` character's copy operations. The total length of the result string will be `n=l*k`, so we have `n*(l+1)/2`. If the number of lines will be proportional to `n`, the complexity will be $O(n^2)$.

In this case it is better to use `StringBuffer` class with an operation `append`:

```
stringBuffer.append(nextLine);
```

Object from the class `StringBuffer` reserve bigger array of character for future operation `append()`. If the size of the array is to small for next append operation, the method creates twice so big array, that was previously. If we start, for example, from initial size equal 16, then the next size will be 32, then 64 etc. It's means then the number of character's copy operations during last increasing phase will need more operation then the whole previous increasing phases: 16+32<64, 16+32+64<128 etc. So in such a method the amortized time for connecting lines of total length *n* is not bigger then 3n (one for first copy of any new character and up to two for all following append operation), so the complexity will be only $O(n)$.

On the current list you have to implement an automaton for string matching problem. Because the complexity of preprocessing (in which you have to count the function δ stored in a two-dimensional array for a m-characters pattern P and an alphabet ∑) in this method is $O(m*|\sum|^3)$ it is desirable to limit the size of an alphabet ∑. If you do not this, you have to assume codes form 0 to 225. Very good idea is to analyze the input string and the pattern to recognize the lower (`lowerCode`) and the bigger (`biggerCode`) code, which occurs in them. Then you can simple transform the interval from `lowerCode` to `biggerCode` into interval from 0 to (`biggerCode-lowerCode`).

Another solution of matching problem is to use Knuth-Morris-Pratt algorithm. Here the preprocessing need time only O(m) to count the π function, which have to be stored in one-dimensional array.

It is very interesting, that this function can be used for faster counting the δ function. To do that you have to use the property, that $\delta(q,a) = \delta(\pi[q],a)$ when $q=m$ or $P[q+1] \neq a$. In this way the complexity for the first algorithm will be $O(m*|\sum|)$.

List of tasks.

1. Prepare a class `LinesReader` for reading a pattern and a text. The pattern and the text can consist of many lines of characters. The reader procedure have to concatenate all lines from the input stream into one `String`. During its work you have to use `StringBuffer` object.
2. Implement algorithm for string matching which bases on automaton. A class with this algorithm has to implement `IStringMatcher` interface.
3. Implement Knuth-Morris-Pratt algorithm for string matching. A class with this algorithm has to implement `IStringMatcher` interface.

For 100 points present solutions for this list till Week 14.
For 80 points present solutions for this list till Week 15.
After Week 15 the list is closed.

There is an appendix on next pages.

Appendix
The solution will be automated tested with tests from console of presented below format.

If a line is empty or starts from `'#'` sign, the line have to be ignored.
In any other case, your program should print an exclamation mark and write (copy) introduced a line and then, depending on the command follow the correct procedure / function.

If a line has a format:
`automaton <PatternLines> <TextLines>`
your program has to read `<PatternLines>` to a `String` object for a pattern and a `<textLines>` to a `String` for a text. Then you have to call `validShifts()` method for an `Automaton` object which will return the result. The result print in the format used by `LinkedList<Integer>.toString()` method.

If a line has a format:
`kmp <PatternLines> <TextLines>`
your program has to read `<PatternLines>` to a `String` object for a pattern and a `<textLines>` to a `String` for a text. Then you have to call `validShifts()` method for a `KMP` object which will return the result. The result print in the format used by `LinkedList<Integer>.toString()` method.

If a line has a format:
`ha`
your program has to end the execution, writing as the last line “END OF EXECUTION”. Every test ends with this line.

A simple test for this task:
INPUT:
```
#Test for Lab12
automaton 1 1
ababac
ababac
kmp 2 3
aba
bac
ggggab
ab
acaaaaaaaaa
automaton 1 3
abab
abababababadsdsdsdsdsdsdsaba
fffjjjabababababababab
cccccccc
kmp 1 1
abcdef
abcdeabcder
ha
```

OUTPUT:
```
START
```

```
START
!automaton 1 1
[0]
!kmp 2 3
[4]
!automaton 1 3
[0, 2, 4, 6, 36, 38, 40, 42, 44, 46, 48]
!kmp 1 1
[]
!ha
END OF EXECUTION
```