Programming Assignment 3: Algorithmic Challenges

Revision: November 13, 2018

Introduction

Welcome to your third programming assignment of the String Processing and Pattern Matching Algorithms! In this programming assignment, you will be practicing implementing very efficient string algorithms.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. find all occurrences of a pattern in text;
2. construct the suffix array efficiently;
3. use suffix arrays for solving the multiple pattern matching problem;
4. construct the suffix tree from the suffix array in linear time.

Passing Criteria: 2 out of 4

Passing this programming assignment requires passing at least 2 out of 4 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Find All Occurrences of a Pattern in a String

Problem Introduction

In this problem, we ask a simple question: how many times one string occurs as a substring of another? Recall that different occurrences of a substring can overlap with each other. For example, ATA occurs three times in CGATATATCCATAG.

This is a classical pattern matching problem in Computer Science solved millions times per day all over the world when computer users use the common "Find" feature in text/code editors and Internet browsers.

Problem Description

Task. Find all occurrences of a pattern in a string.

Input Format. Strings *Pattern* and *Genome*.

Constraints. $1 \leq |Pattern| \leq 10^6$; $1 \leq |Genome| \leq 10^6$; both strings are over A, C, G, T.

Output Format. All starting positions in *Genome* where *Pattern* appears as a substring (using 0-based indexing as usual).

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	18	6.5	6.5	24

Memory Limit. 512MB.

Sample 1.

Input:
```
TACG
GT
```
Output:
```

```
The pattern is longer than the text and hence has no occurrences in the text.

Sample 2.

Input:
```
ATA
ATATA
```
Output:
```
0 2
```
The pattern appears at positions 1 and 3 (and these two occurrences overlap each other).

Sample 3.

Input:
```
ATAT
GATATATGCATATACTT
```
Output:
```
1 3 9
```
The pattern appears at positions 1, 3, and 9 in the text.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch.

2 Problem: Construct the Suffix Array of a Long String

Problem Introduction

The goal in this problem is to construct the suffix array of a given string again, but this time for a longer string. In particular, a quadratic algorithm will not fit into the time limit in this problem. This will require you to implement an almost linear algorithm bringing you close to the state-of-the-art algorithms for constructing suffix arrays.

Problem Description

Task. Construct the suffix array of a string.

Input Format. A string *Text* ending with a "$" symbol.

Constraints. $1 \leq |Text| \leq 2 \cdot 10^5$; except for the last symbol, *Text* contains symbols A, C, G, T only.

Output Format. *SuffixArray*(*Text*), that is, the list of starting positions of sorted suffixes separated by spaces.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	4	30	30	8

Memory Limit. 512MB.

Sample 1.

Input:

```
AAA$
```

Output:

```
3 2 1 0
```

Sorted suffixes:

```
3  $
2  A$
1  AA$
0  AAA$
```

Sample 2.

Input:

```
GAC$
```

Output:

```
3 1 2 0
```

Sorted suffixes:

```
3  $
1  AC$
2  C$
0  GAC$
```

Sample 3.

Input:

```
GAGAGAGA$
```

Output:

```
8 7 5 3 1 6 4 2 0
```

Sorted suffixes:

8	`$`
7	`A$`
5	`AGA$`
3	`AGAGA$`
1	`AGAGAGA$`
6	`GA$`
4	`GAGA$`
2	`GAGAGA$`
0	`GAGAGAGA$`

Sample 4.

Input:

```
AACGATAGCGGTAGA$
```

Output:

```
15 14 0 1 12 6 4 2 8 13 3 7 9 10 11 5
```

Sorted suffixes:

15	`$`
14	`A$`
0	`AACGATAGCGGTAGA$`
1	`ACGATAGCGGTAGA$`
12	`AGA$`
6	`AGCGGTAGA$`
4	`ATAGCGGTAGA$`
2	`CGATAGCGGTAGA$`
8	`CGGTAGA$`
13	`GA$`
3	`GATAGCGGTAGA$`
7	`GCGGTAGA$`
9	`GGTAGA$`
10	`GTAGA$`
11	`TAGA$`
5	`TAGCGGTAGA$`

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch.

3 Problem: Pattern Matching with the Suffix Array

Problem Introduction

In this problem, we will let you use the suffix array to solve the Multiple Pattern Matching Problem. This is what actually happens when one needs to solve the pattern matching problem for a massive string like the human genome: instead of downloading the genome itself, one downloads its suffix array and solves the pattern matching problem using the array.

Problem Description

Task. Find all occurrences of a given collection of patterns in a string.

Input Format. The first line contains a string *Text*. The second line specifies an integer n. The last line gives a collection of n strings $Patterns = \{p_1, \ldots, p_n\}$ separated by spaces.

Constraints. All strings contain symbols A, C, G, T only; $1 \le |Text| \le 10^5$; $1 \le n \le 10^4$; $\sum_{i=1}^{n} |p_i| \le 10^5$.

Output Format. All starting positions (in any order) in *Text* where a pattern appears as a substring (using 0-based indexing as usual). If several patterns occur at the same position of the *Text*, still output this position only **once**.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	1	1	4	14	12	8

Memory Limit. 512MB.

Sample 1.

Input:
```
AAA
1
A
```
Output:
```
0 1 2
```
The pattern `A` appears at positions 0, 1, and 2 in the text.

Sample 2.

Input:
```
ATA
3
C G C
```
Output:
```

```
There are no occurrences of the patterns in the text

Sample 3.

Input:
```
ATATATA
3
ATA C TATAT
```
Output:
```
4 2 0 1
```
The pattern `ATA` appears at positions 0, 2, and 4, the pattern `TATAT` appears at position 1.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch.

4 Advanced Problem: Construct the Suffix Tree from the Suffix Array

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

As we've mentioned earlier, known algorithms for constructing suffix trees in linear time are quite complex. It turns out, however, that one can first construct a suffix array in near-linear time (say, $O(n \log n)$) and then transform it into a suffix tree in linear time. This gives a near-linear time algorithm for constructing a suffix tree!

SuffixTree(*Text*) can be constructed in linear time from *SuffixArray*(*Text*) by using the longest common prefix (LCP) array of *Text*, *LCP*(*Text*), which stores the length of the longest common prefix shared by consecutive lexicographically ordered suffixes of *Text*. For example,

$$LCP(\text{“panamabananas\$”}) = (0, 1, 1, 3, 3, 1, 0, 0, 0, 2, 2, 0, 0).$$

Problem Description

Task. Construct a suffix tree from the suffix array and LCP array of a string.

Input Format. The first line contains a string *Text* ending with a “$” symbol, the second line contains *SuffixArray*(*Text*) as a list of $|Text|$ integers separated by spaces, the last line contains *LCP*(*Text*) as a list of $|Text| - 1$ integers separated by spaces.

Constraints. $1 \le |Text(Text)| \le 2 \cdot 10^5$; except for the last symbol, *Text* contains symbols A, C, G, T only.

Output Format. The output format in this problem differs from the output format in the problem “Suffix Tree” from the Programming Assignment 2 and is somewhat tricky. It is because this problem is harder: the input string can be longer, so it would take too long to output all the edge labels directly and compare them with the correct ones, as their combined length can be $\Theta(|Text|^2)$, which is too much when the *Text* can be as long as 200 000 characters.

Output the $Text$ from the input on the first line. Then output all the edges of the suffix tree **in a specific order** (see below), **each on its own line**. Output each edge as a pair of integers $(start, end)$, where *start* is the position in *Text* corresponding to the start of the edge label substring in the *Text* and *end* is the position right after the end of the edge label in the *Text*. Note that *start* must be a valid position in the *Text*, that is, $0 \le start \le |Text| - 1$, and end must be between 1 and $|Text|$ inclusive. Substring $Text[start..end-1]$ must be equal to the edge label of the corresponding edge. For example, if *Text* = “ACACAA$” and the edge label is “CA”, you can output this edge either as $(1, 3)$ corresponding to $Text[1..2]$ = “CA” or as $(3, 5)$ corresponding to $Text[3..4]$ = “CA” — both variants will be accepted.

The order of the edges is important here — if you output all the correct edges in the wrong order, your solution will not be accepted. However, **you don't need to construct this order yourself if you write in C++, Java or Python3, because it is implemented for you in the starter files**. Output all the edges in the order of sorted suffixes: first, take the leaf of the suffix tree corresponding to the smallest suffix of *Text* and output all the edges on the path from the root to this leaf. Then take the leaf corresponding to the second smallest suffix of *Text* and output all the edges on the path from the root to this leaf **except for those edges which were printed before**. Then take the leaf corresponding to the third smallest suffix, fourth smallest suffix and so on. Print each edge only once — as a part of the path corresponding to the smallest suffix of *Text* where this edge appears. This way, you will only output $O(|Text|)$ integers. See the examples below for clarification.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	2	2	22	10	10	40

Memory Limit. 512MB.

Sample 1.

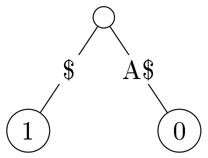
Input:

```
A$
1 0
0
```

Output:

```
A$
1 2
0 2
```

i	$SA[i]$	$LCP[i]$	suffix
0	1	0	`$`
1	0		`A$`



Sample 2.

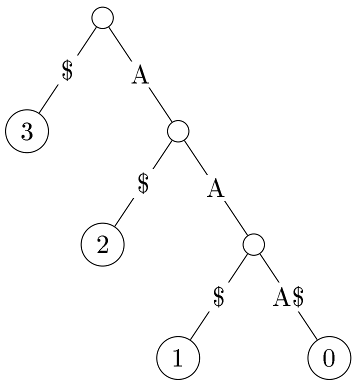
Input:

```
AAA$
3 2 1 0
0 1 2
```

Output:

```
AAA$
3 4
0 1
3 4
1 2
3 4
2 4
```

i	$SA[i]$	$LCP[i]$	suffix
0	3	0	`$`
1	2	1	`A$`
2	1	2	`AA$`
3	0		`AAA$`



Sample 3.

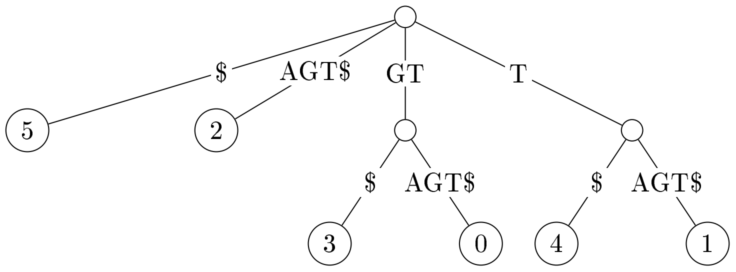
Input:

```
GTAGT$
5 2 3 0 4 1
0 0 2 0 1
```

Output:

```
GTAGT$
5 6
2 6
3 5
5 6
2 6
4 5
5 6
2 6
```

i	$SA[i]$	$LCP[i]$	suffix
0	5	0	`$`
1	2	0	`AGT$`
2	3	2	`GT$`
3	0	0	`GTAGT$`
4	4	1	`T$`
5	1		`TAGT$`



Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to the blank procedure which is supposed to build the suffix tree, then print the edges of the resulting suffix tree in the order specified in the output format description. **You don't have to construct the correct order of edges for the output yourself if you use these starter files**. You are only supposed to implement the procedure to build the suffix tree given the string, its suffix array and LCP array if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch.

5 Solving a Programming Challenge in Five Easy Steps

5.1 Reading Problem Statement

Start by reading the problem statement that contains the description of a computational task, time and memory limits, and a few sample tests. Make sure you understand how an output matches an input in each sample case.

If time and memory limits are not specified explicitly in the problem statement, the following default values are used.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	1	1	1.5	5	5	3

Memory limit: 512 Mb.

5.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly 10^8–10^9 operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1\,000$, and if $n = 100$, even an $O(n^3)$ solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as n is smaller than 20.

5.3 Implementing an Algorithm

Start implementing your algorithm in one of the following programming languages supported by our automated grading system: `C`, `C++`, `Haskell`, `Java`, `JavaScript`, `Python2`, `Python3`, or `Scala`. For all problems, we provide starter solutions for `C++`, `Java`, and `Python3`. For other programming languages, you need to implement a solution from scratch. The grading system detects the programming language of your submission automatically, based on the extension of the submission file.

We have reference solutions in `C++`, `Java`, and `Python3` (that we don't share with you) which solve the problem correctly under the given constraints, and spend at most 1/3 of the time limit and at most 1/2 of the memory limit. You can also use other languages, and we've estimated the time limit multipliers for them. However, we have no guarantee that a correct solution for a particular problem running under the given time and memory constraints exists in any of those other languages.

In the Appendix, we list compiler versions and flags used by the grading system. We recommend using the same compiler flags when you test your solution locally. This will increase the chances that your program behaves in the same way on your machine and on the testing machine (note that a buggy program may behave differently when compiled by different compilers, or even by the same compiler with different flags).

5.4 Testing and Debugging

Submitting your implementation to the grading system without testing it first is a bad idea! Start with small datasets and make sure that your program produces correct results on all sample datasets. Then proceed to checking how long it takes to process a large dataset. To estimate the running time, it makes sense to implement your algorithm as a function like `solve(dataset)` and then implement an additional procedure `generate()` that produces a large dataset. For example, if an input to a problem is a sequence of integers of length $1 \le n \le 10^5$, then generate a sequence of length 10^5, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Check the boundary values to ensure that your program processes correctly both short sequences (e.g., with 2 elements) and long sequences (e.g., with 10^5 elements). If a sequence of integers from 0 to, let's say, 10^6 is given as an input, check how your program behaves when it is given a sequence $0, 0, \dots, 0$ or a sequence $10^6, 10^6, \dots, 10^6$. Afterwards, check it also on randomly generated data. Check degenerate cases like an empty set, three points on a single line, a tree which consists of a single path of nodes, etc.

After it appears that your program works on all these tests, proceed to stress testing. Implement a slow, but simple and correct algorithm and check that two programs produce the same result (note however that this is not applicable to problems where the output is not unique). Generate random test cases as well as biased tests cases such as those with only small numbers or a small range of large numbers, strings containing a single letter "a" or only two different letters (as opposed to strings composed of all possible Latin letters), and so on. Think about other possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, complete graphs, bipartite graphs, etc. If you generate trees, try generating paths, binary trees, stars, etc. If you are generating integers, try generating both prime and composite numbers.

5.5 Submitting to the Grading System

When you are done with testing, submit your program to the grading system! Go to the submission page, create a new submission, and upload a file with your program (make sure to upload a source file rather than an executable). The grading system then compiles your program and runs it on a set of carefully constructed tests to check that it outputs a correct result for all tests and that it fits into the time and memory limits. The grading usually takes less than a minute, but in rare cases, when the servers are overloaded, it might take longer. Please be patient. You can safely leave the page when your solution is uploaded.

As a result, you get a feedback message from the grading system. You want to see the "`Good job!`" message indicating that your program passed all the tests. The messages "`Wrong answer`", "`Time limit exceeded`", "`Memory limit exceeded`" notify you that your program failed due to one of these reasons. If you program fails on one of the first two test cases, the grader will report this to you and will show you the test case and the output of your program. This is done to help you to get the input/output format right. In all other cases, the grader will *not* show you the test case where your program fails.

6 Appendix: Compiler Flags

C (`gcc 5.2.1`). File extensions: `.c`. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

C++ (`g++ 5.2.1`). File extensions: `.cc`, `.cpp`. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your `C/C++` compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and MacOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., `cygwin`.

Java (`Open JDK 8`). File extensions: `.java`. Flags:

```
javac -encoding UTF-8
java -Xmx1024m
```

JavaScript (`Node v6.3.0`). File extensions: `.js`. Flags:

```
nodejs
```

Python 2 (`CPython 2.7`). File extensions: `.py2` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python2"). No flags:

```
python2
```

Python 3 (`CPython 3.4`). File extensions: `.py3` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python3"). No flags:

```
python3
```

Scala (`Scala 2.11.6`). File extensions: `.scala`.

```
scalac
```