Programming Assignment 1: Suffix Trees

Revision: November 13, 2018

Introduction

Welcome to your first programming assignment of the String Processing and Pattern Matching Algorithms! In this programming assignment, you will be practicing implementing a fundamental data structure — suffix tree. Once constructed, a suffix tree allows to solve many non-trivial computational problems for a given string (or strings). As usual, the problems are arranged in order of increasing difficulty: in the first problems your goal is to implement carefully known algorithms, while in the last problem your goal is to first design an algorithm and then to implement it.

In this programming assignment, the grader will show you the input data if your solution fails on any of the tests. This is done to help you to get used to the algorithmic problems in general and get some experience debugging your programs while knowing exactly on which tests they fail. However, for all the following programming assignments, the grader will show the input data only in case your solution fails on one of the first few tests.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. construct a trie from a collection of patterns;
2. use this trie to find all occurrences of patterns in a given text without scanning the text many times;
3. do this again, but in a situation when it is allowed for some patterns to be prefixes of some other patterns;
4. construct the suffix tree of a string;
5. use suffix trees to find the shortest non-shared substring.

Passing Criteria: 3 out of 5

Passing this programming assignment requires passing at least 3 out of 5 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

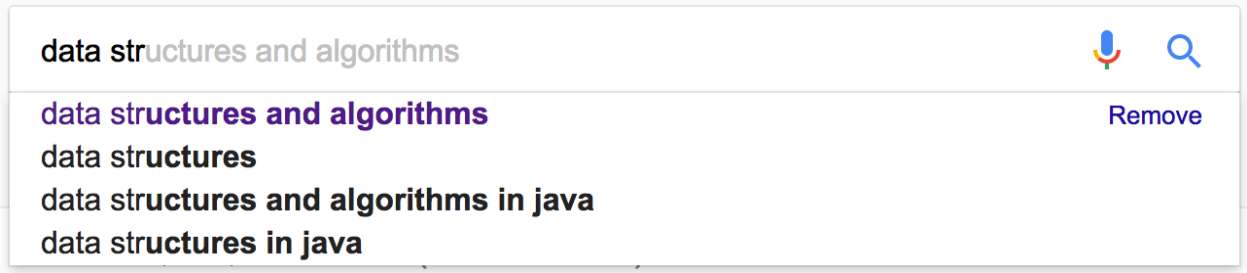
1 Problem: Construct a Trie from a Collection of Patterns

Problem Introduction

For a collection of strings *Patterns*, *Trie*(*Patterns*) is defined as follows.

- The trie has a single root node with indegree 0.
- Each edge of *Trie*(*Patterns*) is labeled with a letter of the alphabet.
- Edges leading out of a given node have distinct labels.
- Every string in *Patterns* is spelled out by concatenating the letters along some path from the root downward.
- Every path from the root to a leaf (i.e, node with outdegree 0), spells a string from *Patterns*.

Tries are a common way of storing a dictionary of words and are used, e.g., for implementing an auto-complete feature in text editors (on your laptop or mobile phone), code editors, and web search engines like Google or Yandex. Just imagine how much time is saved everyday by this feature.



Problem Description

Task. Construct a trie from a collection of patterns.

Input Format. An integer n and a collection of strings $Patterns = \{p_1, \ldots, p_n\}$ (each string is given on a separate line).

Constraints. $1 \leq n \leq 100$; $1 \leq |p_i| \leq 100$ for all $1 \leq i \leq n$; p_i's contain only symbols A, C, G, T; no p_i is a prefix of p_j for all $1 \leq i \neq j \leq n$.

Output Format. The adjacency list corresponding to *Trie*(*Patterns*), in the following format. If *Trie*(*Patterns*) has n nodes, first label the root with 0 and then label the remaining nodes with the integers 1 through $n-1$ in any order you like. Each edge of the adjacency list of *Trie*(*Patterns*) will be encoded by a triple: the first two members of the triple must be the integers i, j labeling the initial and terminal nodes of the edge, respectively; the third member of the triple must be the symbol c labeling the edge; output each such triple in the format `u->v:c` (with no spaces) on a separate line.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	0.5	0.5	3	2	2	4

Memory Limit. 512MB.

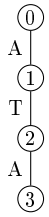
Sample 1.

Input:

```
1
ATA
```

Output:

```
0->1:A
2->3:A
1->2:T
```



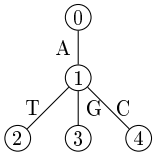
Sample 2.

Input:

```
3
AT
AG
AC
```

Output:

```
0->1:A
1->4:C
1->3:G
1->2:T
```



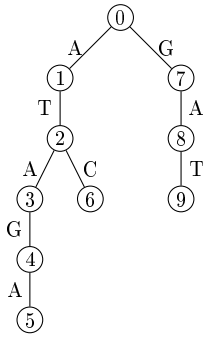
Sample 3.

Input:

```
3
ATAGA
ATC
GAT
```

Output:

```
0->1:A
1->2:T
2->3:A
3->4:G
4->5:A
2->6:C
0->7:G
7->8:A
8->9:T
```



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: Multiple Pattern Matching

Problem Introduction

Another problem that can be solved efficiently with tries is the following.

Multiple Pattern Matching Problem
Find all occurrences of a collection of patterns in a text.

Input: A string *Text* and a collection *Patterns* containing (shorter) strings.
Output: All starting positions in *Text* where a string from *Patterns* appears as a substring.

Again, the multiple pattern matching problem has many applications like highlighting programming language keywords (like `if`, `else`, `elif`) in your favorite IDE (see the screenshot below) and locating reads in a reference genome.

```
if q[0] == "push":
    value = int(q[1])
    stack.append(value)
    if len(max_stack) > 0:
        max_stack.append(max(value, max_stack[-1]))
    else:
        max_stack.append(value)
elif q[0] == "max":
    assert(len(stack) > 0)
    answers.append(max_stack[-1])
```

Problem Description

Input Format. The first line of the input contains a string *Text*, the second line contains an integer n, each of the following n lines contains a pattern from $Patterns = \{p_1, \ldots, p_n\}$.

Constraints. $1 \leq |Text| \leq 10\,000$; $1 \leq n \leq 5\,000$; $1 \leq |p_i| \leq 100$ for all $1 \leq i \leq n$; all strings contain only symbols A, C, G, T; no p_i is a prefix of p_j for all $1 \leq i \neq j \leq n$.

Output Format. All starting positions in *Text* where a string from *Patterns* appears as a substring in increasing order (assuming that *Text* is a 0-based array of symbols).

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	3	7	7	6

Memory Limit. 512MB.

Sample 1.

Input:

```
AAA
1
AA
```

Output:

```
0 1
```

The pattern `AA` appears at positions 0 and 1. Note that these two occurrences of the pattern overlap.

Sample 2.

Input:

```
AA
1
T
```

Output:

```

```

There are no occurrences of the pattern in the text.

Sample 3.

Input:

```
AATCGGGTTCAATCGGGGT
2
ATCG
GGGT
```

Output:

```
1 4 11 15
```

The pattern `ATCG` appears at positions 1 and 11, the pattern `GGGT` appears at positions 4 and 15.

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: Generalized Multiple Pattern Matching

Problem Introduction

The goal in this problem is to extend the solution for the previous problem such that it will be able to handle cases when one of the patterns is a prefix of another pattern. In this case, some patterns are spelled in a trie by traversing a path from the root to an internal vertex, but not to a leaf.

Problem Description

Input Format. The first line of the input contains a string *Text*, the second line contains an integer n, each of the following n lines contains a pattern from $Patterns = \{p_1, \ldots, p_n\}$.

Constraints. $1 \leq |Text| \leq 10\,000$; $1 \leq n \leq 5\,000$; $1 \leq |p_i| \leq 100$ for all $1 \leq i \leq n$; all strings contain only symbols A, C, G, T; **it can be the case that p_i is a prefix of p_j for some i, j.**

Output Format. All starting positions in *Text* where a string from *Patterns* appears as a substring in increasing order (assuming that *Text* is a 0-based array of symbols). **If more than one pattern appears starting at position i, output i once.**

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	3	7	7	6

Memory Limit. 512MB.

Sample 1.

Input:

```
AAA
1
AA
```

Output:

```
0 1
```

The pattern `AA` appears at positions 0 and 1. Note that these two occurrences of the pattern overlap.

Sample 2.

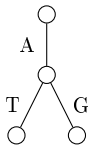
Input:

```
ACATA
3
AT
A
AG
```

Output:

```
0 2 4
```

Text contains occurrences of `A` at positions 0, 2, and 4, as well as an occurrence of `AT` at position 2. Note that the trie looks as follows in this case:



When spelling *Text* from position 0, we don't reach a leaf. Still, there is an occurrence of the pattern `A` at this position.

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 Problem: Construct the Suffix Tree of a String

Problem Introduction

A suffix tree is an extremely powerful indexing data structure having applications in areas like pattern matching, data compression, and bioinformatics. Your goal in this problem is to implement this data structure.

Storing *Trie*(*Patterns*) requires a great deal of memory. So let's process *Text* into a data structure instead. Our goal is to compare each string in *Patterns* against *Text* without needing to traverse *Text* from beginning to end. In more familiar terms, instead of packing *Patterns* onto a bus and riding the long distance down *Text*, our new data structure will be able to "teleport" each string in *Patterns* directly to its occurrences in *Text*.

A **suffix trie**, denoted *SuffixTrie*(*Text*), is the trie formed from all suffixes of *Text*. From now on, we append the dollar-sign ("$") to *Text* in order to mark the end of *Text*. We will also label each leaf of the resulting trie by the starting position of the suffix whose path through the trie ends at this leaf (using 0-based indexing). This way, when we arrive at a leaf, we will immediately know where this suffix came from in *Text*.

However, the runtime and memory required to construct *SuffixTrie*(*Text*) are both equal to the combined length of all suffixes in *Text*. There are $|Text|$ suffixes of *Text*, ranging in length from 1 to $|Text|$ and having total length $|Text| \cdot (|Text| + 1)/2$, which is $\Theta(|Text|^2)$. Thus, we need to reduce both the construction time and memory requirements of suffix tries to make them practical.

Let's not give up hope on suffix tries. We can reduce the number of edges in *SuffixTrie*(*Text*) by combining the edges on any non-branching path into a single edge. We then label this edge with the concatenation of symbols on the consolidated edges. The resulting data structure is called a **suffix tree**, written *SuffixTree*(*Text*).

To match a single *Pattern* to *Text*, we thread *Pattern* into *SuffixTree*(*Text*) by the same process used for a suffix trie. Similarly to the suffix trie, we can use the leaf labels to find starting positions of successfully matched patterns.

Suffix trees save memory because they do not need to store concatenated edge labels from each non-branching path. For example, a suffix tree does not need ten bytes to store the edge labeled "mabananas$" in *SuffixTree*("panamabananas$"); instead, it suffices to store a pointer to position 4 of "panamabananas$", as well as the length of "mabananas$". Furthermore, suffix trees can be constructed in linear time, without having to first construct the suffix trie! We will not ask you to implement this fast suffix tree construction algorithm because it is quite complex.

Problem Description

Task. Construct the suffix tree of a string.

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

Constraints. $1 \leq |Text| \leq 50\,000$; except for the last symbol, *Text* contains symbols A, C, G, T only.

Output Format. The strings labeling the edges of *SuffixTree*(*Text*) in any order.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	1	1	4	15	10	6

Memory Limit. 512MB.

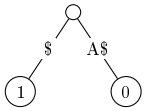
Sample 1.

Input:

```
A$
```

Output:

```
A$
$
```



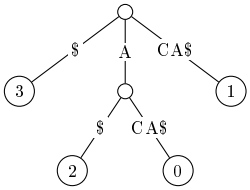
Sample 2.

Input:

```
ACA$
```

Output:

```
$
A
$
CA$
CA$
```



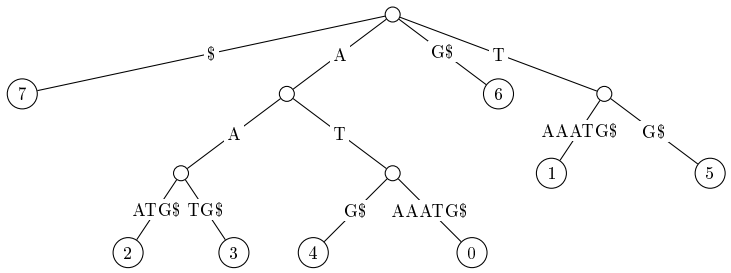
Sample 3.

Input:

```
ATAAATG$
```

Output:

```
AAATG$
G$
T
ATG$
TG$
A
A
AAATG$
G$
T
G$
$
```



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.

What To Do

You can construct a trie from all the suffixes of the initial string as in the first problem. Then you can "compress" it into the suffix tree by deleting all nodes of the trie with only one child, merging the incoming and the outgoing edge of such node into one edge, concatenating the edge labels. However, if you do this and also store the substrings as edge labels directly, this will be too slow and also use too much memory. Use the hint from the lecture to only store the pair (start, length) of the substring of text corresponding to the edge label instead of storing this substring itself. Also note that when you create an edge from a node to a leaf of the tree, you don't need to go through the whole substring corresponding to this edge character-by-character, you already know the start and the length of the corresponding substring. If it's still too slow, you'll need to build the suffix tree directly without building the suffix trie first. To do that, you'll need to do almost the same, but creating the nodes only when branching happens by breaking the existing edge in the middle.

5 Advanced Problem: Find the Shortest Non-Shared Substring of Two Strings

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).

Shortest Non-Shared Substring Problem
Find the shortest substring of one string that does not appear in another string.

Input: Strings $Text_1$ and $Text_2$.
Output: The shortest (non-empty) substring of $Text_1$ that does not appear in $Text_2$.

A naive way to solve this problem is to iterate through all substrings of $Text_1$ and for substring to check whether it appears in $Text_2$. This gives an algorithm with the running time $O(n^4)$ (where $n = \max\{|Text_1|, |Text_2|\}$. Surprisingly, one can solve this problem in time $O(n)$ using suffix trees! (This, however, requires linear time construction of suffix trees. Since the corresponding algorithm is quite complex, we will not ask you to implement it.) For this, construct the suffix tree of a string $Text_1\#Text_2\$$ (where # and \$ are new symbols). Do you see how to traverse the constructed tree to find the shortest non-shared substring?

If in the first stage of this algorithm a linear time suffix tree construction algorithm is used, then the resulting algorithm has linear time, too. This is a truly remarkable fact. To appreciate it, try to invent a linear time algorithm for this problem that does not exploit suffix trees.

Problem Description

Input Format. Strings $Text_1$ and $Text_2$.

Constraints. $1 \le |Text_1|, |Text_2| \le 2000$; strings have equal length ($|Text_1| = |Text_2|$), are not equal ($Text_1 \ne Text_2$), and contain symbols A, C, G, T only.

Output Format. The shortest (non-empty) substring of $Text_1$ that does not appear in $Text_2$. (Multiple solutions may exist, in which case you may return any one.)

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	5	8	5	10

Memory Limit. 1024MB.

Sample 1.

Input:

```
A
T
```

Output:

```
A
```

$Text_2$ does not contain the string `A`, hence it is clearly a shortest such string.

Sample 2.

Input:

```
AAAAAAAAAAAAAAAAAAAA
TTTTTTTTTTTTTTTTTTTT
```

Output:

```
A
```

Again, $Text_2$ does not contain the string `A`, so it is a shortest one.

Sample 3.

Input:

```
CCAAGCTGCTAGAGG
CATGCTGGGCTGGCT
```

Output:

```
AA
```

In this case, $Text_2$ contains all symbols `A`, `C`, `G`, `T`, that is, all substrings of $Text_1$ of length 1. At the same time, $Text_2$ does not contain `AA`, hence it is a shortest substring of $Text_1$ that does not appear in $Text_2$.

Sample 4.

Input:

```
ATGCGATGACCTGACTGA
CTCAACGTATTGGCCAGA
```

Output:

```
ATG
```

The string `ATG` is a substring of $Text_1$ and it does not appear in $Text_2$. At the same time, $Text_2$ contains all 16 strings of length 2 and all 4 strings of length 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.

6 Solving a Programming Challenge in Five Easy Steps

6.1 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.

6.2 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.

6.3 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).

6.4 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 $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. 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.

6.5 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.

7 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
```