Programming Assignment 2: Burrows–Wheeler Transform and Suffix Arrays

Revision: November 13, 2018

Introduction

Welcome to your second programming assignment of the String Processing and Pattern Matching Algorithms! In this programming assignment, you will be practicing implementing Burrows–Wheeler transform and suffix arrays.

Recall that starting from this programming assignment, the grader will show you only the first few tests.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. compute the Burrows–Wheeler transform (BWT) of a string;
2. compute the inverse of BWT;
3. use BWT for pattern matching;
4. construct the suffix array of a string.

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: Construct the Burrows–Wheeler Transform of a String

Problem Introduction

The Burrows–Wheeler transform of a string *Text* permutes the symbols of *Text* so that it becomes well compressible. Moreover, the transformation is reversible: one can recover the initial string *Text* from its Burrows–Wheeler transform. However, data compression is not its only application: it is also used for solving the multiple pattern matching problem and the sequence alignment problem.

$BWT(Text)$ is defined as follows. First, form all possible cyclic rotations of *Text*; a cyclic rotation is defined by chopping off a suffix from the end of *Text* and appending this suffix to the beginning of *Text*. Then, order all the cyclic rotations of Text lexicographically to form a $|Text| \times |Text|$ matrix of symbols denoted by $M(Text)$. $BWT(Text)$ is the last column of $M(Text)$

Problem Description

Task. Construct the Burrows–Wheeler transform of a string.

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

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

Output Format. $BWT(Text)$.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	0.5	0.5	2.5	1	2.5	1.5

Memory Limit. 512MB.

Sample 1.

Input:

```
AA$
```

Output:

```
AA$
```

$$M(Text) = \begin{bmatrix} \$ & A & A \\ A & \$ & A \\ A & A & \$ \end{bmatrix}$$

Sample 2.

Input:

```
ACACACAC$
```

Output:

```
CCCC$AAAA
```

$$M(\mathit{Text}) = \begin{bmatrix} \$ & A & C & A & C & A & C & A & C \\ A & C & \$ & A & C & A & C & A & C \\ A & C & A & C & \$ & A & C & A & C \\ A & C & A & C & A & C & \$ & A & C \\ A & C & A & C & A & C & A & C & \$ \\ C & \$ & A & C & A & C & A & C & A \\ C & A & C & \$ & A & C & A & C & A \\ C & A & C & A & C & \$ & A & C & A \\ C & A & C & A & C & A & C & \$ & A \end{bmatrix}$$

Sample 3.

Input:

```
AGACATA$
```

Output:

```
ATG$CAAA
```

$$M(\mathit{Text}) = \begin{bmatrix} \$ & A & G & A & C & A & T & A \\ A & \$ & A & G & A & C & A & T \\ A & C & A & T & A & \$ & A & G \\ A & G & A & C & A & T & A & \$ \\ A & T & A & \$ & A & G & A & C \\ C & A & T & A & \$ & A & G & A \\ G & A & C & A & T & A & \$ & A \\ T & A & \$ & A & G & A & C & A \end{bmatrix}$$

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: Reconstruct a String from its Burrows–Wheeler Transform

Problem Introduction

In the previous problem, we introduced the Burrows–Wheeler transform of a string *Text*. It permutes the symbols of *Text* making it well compressible. However, there were no sense in this, if this process would not be reversible. It turns out that it is reversible, and your goal in this problem is to recover *Text* from $BWT(Text)$.

Problem Description

Task. Reconstruct a string from its Burrows–Wheeler transform.

Input Format. A string *Transform* with a single “\$” sign.

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

Output Format. The string *Text* such that $BWT(Text) = Transform$. (There exists a unique such string.)

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	2	2	10	10	10	6

Memory Limit. 512MB.

Sample 1.

Input:

```
AC$A
```

Output:

```
ACA$
```

$$M(Text) = \begin{bmatrix} \$ & A & C & A \\ A & \$ & A & C \\ A & C & A & \$ \\ C & A & \$ & A \end{bmatrix}$$

Sample 2.

Input:

```
AGGGAA$
```

Output:

```
GAGAGA$
```

$$M(Text) = \begin{bmatrix} \$ & G & A & G & A & G & A \\ A & \$ & G & A & G & A & G \\ A & G & A & \$ & G & A & G \\ A & G & A & G & A & \$ & G \\ G & A & \$ & G & A & G & A \\ G & A & G & A & \$ & G & A \\ G & A & G & A & G & A & \$ \end{bmatrix}$$

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

To solve this problem, it is enough to implement carefully the corresponding algorithm covered in the lectures.

3 Problem: Matching Against a Compressed String

Problem Introduction

Not only the Burrows–Wheeler transform makes the input string *Text* well compressible, it also allows one to solve the pattern matching problem using the compressed strings instead of the initial string! This is another beautiful property of the Burrows–Wheeler transform which allows us to avoid decompressing the string, and thus to save lots of memory, while still solving the problem at hand.

The algorithm BWMatching counts the total number of matches of *Pattern* in *Text*, where the only information that we are given is *FirstColumn* and *LastColumn* = *BWT*(*Text*) in addition to the Last-to-First mapping. The pointers *top* and *bottom* are updated by the green lines in the following pseudocode.

```
BWMatching(FirstColumn, LastColumn, Pattern, LastToFirst):
top ← 0
bottom ← |LastColumn| − 1
while top ≤ bottom:
  if Pattern is nonempty:
    symbol ← last letter in Pattern
    remove last letter from Pattern
    if positions from top to bottom in LastColumn contain an occurrence of symbol:
      topIndex ← first position of symbol among positions from top to bottom in LastColumn
      bottomIndex ← last position of symbol among positions from top to bottom in LastColumn
      top ← LastToFirst(topIndex)
      bottom ← LastToFirst(bottomIndex)
    else:
      return 0
  else:
    return bottom − top + 1
```

The Last-to-First array, denoted *LastToFirst*(i), answers the following question: given a symbol at position i in *LastColumn*, what is its position in *FirstColumn*? For example, if *Text* = panamabananas\$, *BWT*(*Text*) = smnpbnnaaaaa\$a, *FirstCol*(*Text*) = \$aaaaaabmnnnps, then we can rewrite *BWT*(*Text*) = $\mathrm{s_1m_1n_1p_1b_1n_2n_3a_1a_2a_3a_4a_5\$_1a_6}$ and *FirstCol*(*Text*) = $\mathrm{\$_1a_1a_2a_3a_4a_5a_6b_1m_1n_1n_2n_3p_1s_1}$, and now we see that $\mathrm{a_3}$ in *BWT*(*Text*) corresponds to $\mathrm{a_3}$ in *FirstCol*(*Text*).

If you implement BWMatching, you probably will find the algorithm to be slow. The reason for its sluggishness is that updating the pointers *top* and *bottom* is time-intensive, since it requires examining every symbol in *LastColumn* between *top* and *bottom* at each step. To improve BWMatching, we introduce a function $Count_{symbol}(i, LastColumn)$, which returns the number of occurrences of symbol in the first i positions of *LastColumn*. For example,

$$Count_{\text{“n”}}(10, \text{“smnpbnnaaaaa\$a”}) = 3 \text{ and } Count_{\text{“a”}}(4, \text{“smnpbnnaaaaa\$a”}) = 0\,.$$

The green lines from BWMatching can be compactly described without the First-to-Last mapping by the following two lines:

```
top ← (Count_symbol + 1)-th occurrence of character symbol in FirstColumn
bottom ← position of symbol with rank Count_symbol(bottom + 1, LastColumn) in FirstColumn
```

Define *FirstOccurrence*(*symbol*) as the first position of symbol in *FirstColumn*. If *Text* = “panamabananas\$”, then *FirstColumn* is “\$aaaaaabmnnnps”, and the array holding all values of *FirstOccurrence* is [0, 1, 7, 8, 9, 12, 13]. For DNA strings of any length, the array *FirstOccurrence* contains only five elements.

The two lines of pseudocode from the previous step can now be rewritten as follows:

```
$top \leftarrow FirstOccurrence(symbol) + Count_{symbol}(top, LastColumn)$
$bottom \leftarrow FirstOccurrence(symbol) + Count_{symbol}(bottom + 1, LastColumn) - 1$
```

In the process of simplifying the green lines of pseudocode from BWMatching, we have also eliminated the need for both *FirstColumn* and *LastToFirst*, resulting in a more efficient algorithm called BetterBWMatching.

```
BWMatching(FirstOccurrence, LastColumn, Pattern, Count):
$top \leftarrow 0$
$bottom \leftarrow |LastColumn| - 1$
while $top \leq bottom$:
  if Pattern is nonempty:
    symbol ← last letter in Pattern
    remove last letter from Pattern
    if positions from top to bottom in LastColumn contain an occurrence of symbol:
      $top \leftarrow FirstOccurrence(symbol) + Count_{symbol}(top, LastColumn)$
      $bottom \leftarrow FirstOccurrence(symbol) + Count_{symbol}(bottom + 1, LastColumn) - 1$
    else:
      return 0
  else:
    return $bottom - top + 1$
```

Problem Description

Task. Implement BetterBWMatching algorithm.

Input Format. A string $BWT(Text)$, followed by an integer n and a collection of n strings $Patterns = \{p_1, \ldots, p_n\}$ (on one line separated by spaces).

Constraints. $1 \leq |BWT(Text)| \leq 10^6$; except for the one $ symbol, $BWT(Text)$ contains symbols A, C, G, T only; $1 \leq n \leq 5\,000$; for all $1 \leq i \leq n$, p_i is a string over A, C, G, T; $1 \leq |p_i| \leq 1\,000$.

Output Format. A list of integers, where the i-th integer corresponds to the number of substring matches of the i-th member of *Patterns* in *Text*.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	5	5	8	24	24	15

Memory Limit. 512MB.

Sample 1.

Input:

```
AGGGAA$
1
GA
```

Output:

```
3
```

In this case, *Text* = `GAGAGA$`. The pattern `GA` appears three times in it.

Sample 2.

Input:

```
ATT$AA
2
ATA A
```

Output:

```
2 3
```

Text = ATATA$ contains two occurrences of ATA and three occurrences of A.

Sample 3.

Input:

```
AT$TCTATG
2
TCT TATG
```

Output:

```
0 0
```

Text = ATCGTTTA does not contain any occurrences of two given patterns.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass the Burrows–Wheeler Transform to a preprocessing procedure to precompute some useful values, then pass each pattern along with BWT and precomputed values to the procedure which counts the number of occurrences of the pattern in the text, and then write the result to the standard output. You are supposed to implement these two procedure which are left blank if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch.

What To Do

To solve this problem, it is enough to carefully implement the algorithm described in the lectures. However, don't forget that you need to do the preprocessing of the *Text* only once, and then use the results. If you do the preprocessing of the *Text* each time, there is no point in such preprocessing, you don't save anything. But if you do the preprocessing once, save the results, and use them for searching each pattern, you save a lot on each search.

4 Problem: Construct the Suffix Array of a String

Problem Introduction

We saw that suffix trees can be too memory intensive to apply in practice. This becomes a serious issue for the case of massive datasets like the ones arising in bioinformatics.

In 1993, Udi Manber and Gene Myers introduced suffix arrays as a memory-efficient alternative to suffix trees. To construct *SuffixArray*(*Text*), we first sort all suffixes of *Text* lexicographically, assuming that "$" comes first in the alphabet. The suffix array is the list of starting positions of these sorted suffixes. For example,

$$\textit{SuffixArray}(\text{“panamabananas\$”}) = (13, 5, 3, 1, 7, 9, 11, 6, 4, 2, 8, 10, 0, 12)$$

E.g., the suffix tree of a human genome requires about 60 Gb, while the suffix array occupies around 12 Gb.

Problem Description

Task. Construct the suffix array of a string.

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

Constraints. $1 \le |Text| \le 10^4$; except for the last symbol, *Text* contains symbols A, C, G, T only.

Output Format. *SuffixArray*(*Text*), that is, the list of starting positions (0-based) of sorted suffixes separated by spaces.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	4	2	5	4

Memory Limit. 512MB.

Sample 1.

Input:

```
GAC$
```

Output:

```
3 1 2 0
```

Sorted suffixes:

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

Sample 2.

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 3.

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.

What To Do

To solve this problem, it is enough to just sort all suffixes of *Text*.

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 $1 \leq n \leq 10^5$, 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
```