Module: BWT and Suffix Arrays (Week 2 out of 4)
Course: Algorithms on Strings (Course 4 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 2: Burrows–Wheeler Transform and Suffix Arrays

Revision: July 3, 2017

Introduction

Welcome to your second programming assignment of the Algorithms on Strings class! 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 (see the questions 6.4 and 6.5 in the FAQ section).

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	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	0.5	0.5	0.75	0.5	0.75	1	2.5	2.5	1.5

Memory Limit. 512MB.

Sample 1.

Input:
```
AA$
```

Output:
```
AA$
```

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

3

Sample 2.

Input:
```
ACACACAC$
```

Output:
```
CCCC$AAAA
```

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

Sample 3.

Input:
```
AGACATA$
```

Output:
```
ATG$CAAA
```

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

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. Filename: `bwt`

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Reconstruct a String from its Burrows–Wheeler Transform COMPLETE!

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	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	2	2	3	10	3	4	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. Filename: `bwtinverse`

What To Do

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

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

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      // Pointer to top of column
  bottom ← |LastColumn| − 1  // Pointer to bottom of column
  while top ≤ bottom:
    if Pattern is nonempty:
      symbol ← last letter in Pattern   // Let symbol be the last character in Pattern
      remove last letter from Pattern   // Let symbol be the last character in 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 $BWT(Text) = s_1m_1n_1p_1b_1n_2n_3a_1a_2a_3a_4a_5\$_1a_6$ and $FirstCol(Text) = \$_1a_1a_2a_3a_4a_5a_6b_1m_1n_1n_2n_3p_1s_1$, and now we see that a_3 in *BWT*(*Text*) corresponds to 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 ARRAY

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

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 BETTERBW-MATCHING.

```
BWMATCHING(FirstOccurrence, LastColumn, Pattern, Count):
top ← 0      // initialise to top of last column
bottom ← |LastColumn| − 1      // initialise to bottom of last column
while top ≤ bottom:
  if Pattern is nonempty:
    symbol ← last letter in Pattern
    remove last letter from Pattern      // initialise to bottom of last column
    if positions from top to bottom in LastColumn contain an occurrence of symbol:
        top ← FirstOccurrence(symbol) + Count_symbol(top, LastColumn)
        bottom ← 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	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	4	4	6	24	6	8	24	24	12

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. Filename: `bwmatching`.

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.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

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,

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

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. $1 \leq |Text| \leq 10^4$; 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	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	1	1	2	1	1.5	2	5	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. Filename: `suffix_array`

What To Do

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

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

5 General Instructions and Recommendations on Solving Algorithmic Problems

Your main goal in an algorithmic problem is to implement a program that solves a given computational problem in just few seconds even on massive datasets. Your program should read a dataset from the standard input and write an answer to the standard output.

Below we provide general instructions and recommendations on solving such problems. Before reading them, go through readings and screencasts in the first module that show a step by step process of solving two algorithmic problems: link.

5.1 Reading the Problem Statement

You start by reading the problem statement that contains the description of a particular computational task as well as time and memory limits your solution should fit in, and one or two sample tests. In some problems your goal is just to implement carefully an algorithm covered in the lectures, while in some other problems you first need to come up with an algorithm yourself.

5.2 Designing an Algorithm

If your goal is to design an algorithm yourself, one of the things it is important to realize is the expected running time of your algorithm. Usually, you can guess it from the problem statement (specifically, from the subsection called constraints) as follows. Modern computers perform roughly 10^8–10^9 operations per second. So, if the maximum size of a dataset in the problem description is $n = 10^5$, then most probably an algorithm with quadratic running time is not going to fit into time limit (since for $n = 10^5$, $n^2 = 10^{10}$) while a solution with running time $O(n \log n)$ will fit. However, an $O(n^2)$ solution will fit if n is up to $10^3 = 1000$, and if n is at most 100, even $O(n^3)$ solutions will fit. In some cases, the problem is so hard that we do not know a polynomial solution. But for n up to 18, a solution with $O(2^n n^2)$ running time will probably fit into the time limit.

To design an algorithm with the expected running time, you will of course need to use the ideas covered in the lectures. Also, make sure to carefully go through sample tests in the problem description.

5.3 Implementing Your Algorithm

When you have an algorithm in mind, you start implementing it. Currently, you can use the following programming languages to implement a solution to a problem: C, C++, C#, Haskell, Java, JavaScript, Python2, Python3, Ruby, Scala. For all problems, we will be providing starter solutions for C++, Java, and Python3. If you are going to use one of these programming languages, use these starter files. For other programming languages, you need to implement a solution from scratch.

5.4 Compiling Your Program

For solving programming assignments, you can use any of the following programming languages: C, C++, C#, Haskell, Java, JavaScript, Python2, Python3, Ruby, and Scala. However, we will only be providing starter solution files for C++, Java, and Python3. The programming language of your submission is detected automatically, based on the extension of your submission.

We have reference solutions in C++, Java and Python3 which solve the problem correctly under the given restrictions, and in most cases 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.

Your solution will be compiled as follows. We recommend that when testing your solution locally, you use the same compiler flags for compiling. 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).

- 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`.

- C# (`mono 3.2.8`). File extensions: `.cs`. Flags:

```
mcs
```

- Haskell (`ghc 7.8.4`). File extensions: `.hs`. Flags:

```
ghc -O2
```

- 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
```

- Ruby (`Ruby 2.1.5`). File extensions: `.rb`.

```
ruby
```

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

```
scalac
```

5.5   Testing Your Program

When your program is ready, you start testing it. It makes sense to start with small datasets (for example, sample tests provided in the problem description). Ensure that your program produces a correct result.

You then proceed to checking how long does it take your program to process a massive dataset. For this, 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 exactly 10^5, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Also, check the boundary values. Ensure that your program processes correctly sequences of size $n = 1, 2, 10^5$. If a sequence of integers from 0 to, 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 $10^6, 10^6, \ldots, 10^6$. Check also on randomly generated data. For each such test check that you program produces a correct result (or at least a reasonably looking result).

In the end, we encourage you to stress test your program to make sure it passes in the system at the first attempt. See the readings and screencasts from the first week to learn about testing and stress testing: link.

5.6   Submitting Your Program to the Grading System

When you are done with testing, you submit your program to the grading system. For this, you go the submission page, create a new submission, and upload a file with your program. The grading system then compiles your program (detecting the programming language based on your file extension, see Subsection 5.4) and runs it on a set of carefully constructed tests to check that your program always outputs a correct result and that it always fits into the given time and memory limits. The grading usually takes no more 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. The feedback message that you will love to see is: `Good job!` This means that your program has passed all the tests. On the other hand, the three messages `Wrong answer`, `Time limit exceeded`, `Memory limit exceeded` notify you that your program failed due to one these three reasons. Note that the grader will not show you the actual test you program have failed on (though it does show you the test if your program have failed on one of the first few tests; this is done to help you to get the input/output format right).

5.7   Debugging and Stress Testing Your Program

If your program failed, you will need to debug it. Most probably, you didn't follow some of our suggestions from the section 5.5. See the readings and screencasts from the first week to learn about debugging your program: link.

You are almost guaranteed to find a bug in your program using stress testing, because the way these programming assignments and tests for them are prepared follows the same process: small manual tests, tests for edge cases, tests for large numbers and integer overflow, big tests for time limit and memory limit checking, random test generation. Also, implementation of wrong solutions which we expect to see and stress testing against them to add tests specifically against those wrong solutions.

Go ahead, and we hope you pass the assignment soon!

6 Frequently Asked Questions

6.1 I submit the program, but nothing happens. Why?

You need to create submission and upload the file with your solution in one of the programming languages C, C++, Java, or Python (see Subsections 5.3 and 5.4). Make sure that after uploading the file with your solution you press on the blue "Submit" button in the bottom. After that, the grading starts, and the submission being graded is enclosed in an orange rectangle. After the testing is finished, the rectangle disappears, and the results of the testing of all problems is shown to you.

6.2 I submit the solution only for one problem, but all the problems in the assignment are graded. Why?

Each time you submit any solution, the last uploaded solution for each problem is tested. Don't worry: this doesn't affect your score even if the submissions for the other problems are wrong. As soon as you pass the sufficient number of problems in the assignment (see in the pdf with instructions), you pass the assignment. After that, you can improve your result if you successfully pass more problems from the assignment. We recommend working on one problem at a time, checking whether your solution for any given problem passes in the system as soon as you are confident in it. However, it is better to test it first, please refer to the reading about stress testing: link.

6.3 What are the possible grading outcomes, and how to read them?

Your solution may either pass or not. To pass, it must work without crashing and return the correct answers on all the test cases we prepared for you, and do so under the time limit and memory limit constraints specified in the problem statement. If your solution passes, you get the corresponding feedback "Good job!" and get a point for the problem. If your solution fails, it can be because it crashes, returns wrong answer, works for too long or uses too much memory for some test case. The feedback will contain the number of the test case on which your solution fails and the total number of test cases in the system. The tests for the problem are numbered from 1 to the total number of test cases for the problem, and the program is always tested on all the tests in the order from the test number 1 to the test with the biggest number.

Here are the possible outcomes:

Good job! Hurrah! Your solution passed, and you get a point!

Wrong answer. Your solution has output incorrect answer for some test case. If it is a sample test case from the problem statement, or if you are solving Programming Assignment 1, you will also see the input data, the output of your program and the correct answer. Otherwise, you won't know the input, the output, and the correct answer. Check that you consider all the cases correctly, avoid integer overflow, output the required white space, output the floating point numbers with the required precision, don't output anything in addition to what you are asked to output in the output specification of the problem statement. See this reading on testing: link.

Time limit exceeded. Your solution worked longer than the allowed time limit for some test case. If it is a sample test case from the problem statement, or if you are solving Programming Assignment 1, you will also see the input data and the correct answer. Otherwise, you won't know the input and the correct answer. Check again that your algorithm has good enough running time estimate. Test your program locally on the test of maximum size allowed by the problem statement and see how long it works. Check that your program doesn't wait for some input from the user which makes it to wait forever. See this reading on testing: link.

Memory limit exceeded. Your solution used more than the allowed memory limit for some test case. If it is a sample test case from the problem statement, or if you are solving Programming Assignment 1,

you will also see the input data and the correct answer. Otherwise, you won't know the input and the correct answer. Estimate the amount of memory that your program is going to use in the worst case and check that it is less than the memory limit. Check that you don't create too large arrays or data structures. Check that you don't create large arrays or lists or vectors consisting of empty arrays or empty strings, since those in some cases still eat up memory. Test your program locally on the test of maximum size allowed by the problem statement and look at its memory consumption in the system.

`Cannot check answer. Perhaps output format is wrong.` This happens when you output something completely different than expected. For example, you are required to output word "Yes" or "No", but you output number 1 or 0, or vice versa. Or your program has empty output. Or your program outputs not only the correct answer, but also some additional information (this is not allowed, so please follow exactly the output format specified in the problem statement). Maybe your program doesn't output anything, because it crashes.

`Unknown signal 6 (or 7, or 8, or 11, or some other).` This happens when your program crashes. It can be because of division by zero, accessing memory outside of the array bounds, using uninitialized variables, too deep recursion that triggers stack overflow, sorting with contradictory comparator, removing elements from an empty data structure, trying to allocate too much memory, and many other reasons. Look at your code and think about all those possibilities. Make sure that you use the same compilers and the same compiler options as we do. Try different testing techniques from this reading: link.

`Internal error: exception...` Most probably, you submitted a compiled program instead of a source code.

`Grading failed.` Something very wrong happened with the system. Contact Coursera for help or write in the forums to let us know.

6.4   How to understand why my program fails and to fix it?

If your program works incorrectly, it gets a feedback from the grader. For the Programming Assignment 1, when your solution fails, you will see the input data, the correct answer and the output of your program in case it didn't crash, finished under the time limit and memory limit constraints. If the program crashed, worked too long or used too much memory, the system stops it, so you won't see the output of your program or will see just part of the whole output. We show you all this information so that you get used to the algorithmic problems in general and get some experience debugging your programs while knowing exactly on which tests they fail.

However, in the following Programming Assignments throughout the Specialization you will only get so much information for the test cases from the problem statement. For the next tests you will only get the result: passed, time limit exceeded, memory limit exceeded, wrong answer, wrong output format or some form of crash. We hide the test cases, because it is crucial for you to learn to test and fix your program even without knowing exactly the test on which it fails. In the real life, often there will be no or only partial information about the failure of your program or service. You will need to find the failing test case yourself. Stress testing is one powerful technique that allows you to do that. You should apply it after using the other testing techniques covered in this reading.

6.5   Why do you hide the test on which my program fails?

Often beginner programmers think by default that their programs work. Experienced programmers know, however, that their programs almost never work initially. Everyone who wants to become a better programmer needs to go through this realization.

When you are sure that your program works by default, you just throw a few random test cases against it, and if the answers look reasonable, you consider your work done. However, mostly this is not enough. To

make one's programs work, one must test them really well. Sometimes, the programs still don't work although you tried really hard to test them, and you need to be both skilled and creative to fix your bugs. Solutions to algorithmic problems are one of the hardest to implement correctly. That's why in this Specialization you will gain this important experience which will be invaluable in the future when you write programs which you really need to get right.

It is crucial for you to learn to test and fix your programs yourself. In the real life, often there will be no or only partial information about the failure of your program or service. Still, you will have to reproduce the failure to fix it (or just guess what it is, but that's rare, and you will still need to reproduce the failure to make sure you have really fixed it). When you solve algorithmic problems, it is very frequent to make subtle mistakes. That's why you should apply the testing techniques described in this reading to find the failing test case and fix your program.

6.6 My solution does not pass the tests? May I post it in the forum and ask for a help?

No, please do not post any solutions in the forum or anywhere on the web, even if a solution does not pass the tests (as in this case you are still revealing parts of a correct solution). Recall the third item of the Coursera Honor Code: "I will not make solutions to homework, quizzes, exams, projects, and other assignments available to anyone else (except to the extent an assignment explicitly permits sharing solutions). This includes both solutions written by me, as well as any solutions provided by the course staff or others" (link).

6.7 My implementation always fails in the grader, though I already tested and stress tested it a lot. Would not it be better if you give me a solution to this problem or at least the test cases that you use? I will then be able to fix my code and will learn how to avoid making mistakes. Otherwise, I do not feel that I learn anything from solving this problem. I am just stuck.

First of all, you always learn from your mistakes.

The process of trying to invent new test cases that might fail your program and proving them wrong is often enlightening. This thinking about the invariants which you expect your loops, ifs, etc. to keep and proving them wrong (or right) makes you understand what happens inside your program and in the general algorithm you're studying much more.

Also, it is important to be able to find a bug in your implementation without knowing a test case and without having a reference solution. Assume that you designed an application and an annoyed user reports that it crashed. Most probably, the user will not tell you the exact sequence of operations that led to a crash. Moreover, there will be no reference application. Hence, once again, it is important to be able to locate a bug in your implementation yourself, without a magic oracle giving you either a test case that your program fails or a reference solution. We encourage you to use programming assignments in this class as a way of practicing this important skill.

If you have already tested a lot (considered all corner cases that you can imagine, constructed a set of manual test cases, applied stress testing), but your program still fails and you are stuck, try to ask for help on the forum. We encourage you to do this by first explaining what kind of corner cases you have already considered (it may happen that when writing such a post you will realize that you missed some corner cases!) and only then asking other learners to give you more ideas for tests cases.