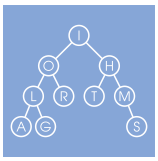


Module: Suffix Trees (Week 1 out of 4)
Course: Algorithms on Strings (Course 4 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 1: Suffix Trees

Revision: August 2, 2016

Introduction

Welcome to your first programming assignment of the Algorithms on Strings class! 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). You will solve one such problem in the end of this assignment.

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 (please review the questions 7.4 and 7.5 in the FAQ section for a more detailed explanation of this behavior of the grader).

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

Reads will form a collection of strings *Patterns* that we wish to match against a reference genome *Text*. For each string in *Patterns*, we will first find all its exact matches as a substring of *Text* (or conclude that it does not appear in *Text*). When hunting for the cause of a genetic disorder, we can immediately eliminate from consideration areas of the reference genome where exact matches occur.

> **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.

To solve this problem, we will consolidate *Patterns* into a directed tree called a trie (pronounced "try"), which is written *Trie*(*Patterns*) and has the following properties.

- The trie has a single root node with indegree 0, denoted *root*.
- 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**, or node with outdegree 0, spells a string from *Patterns*.

The most obvious way to construct *Trie*(*Patterns*) is by iteratively adding each string from *Patterns* to the growing trie, as implemented by the following algorithm.

```
TrieConstruction(Patterns)
Trie ← a graph consisting of a single node root
for each string Pattern in Patterns:
  currentNode ← root
  for i from 0 to |Pattern| − 1:
    currentSymbol ← Pattern[i]
    if there is an outgoing edge from currentNode with label currentSymbol:
      currentNode ← ending node of this edge
    else:
      add a new node newNode to Trie
      add a new edge from currentNode to newNode with label currentSymbol
      currentNode ← newNode
return Trie
```

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	C#	Haskell	JavaScript	Ruby	Scala
time in seconds	0.5	0.5	2	2	0.75	1	2	2	4

Memory Limit. 512Mb.

Sample 1.

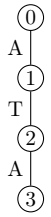
Input:

```
1
ATA
```

Output:

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

Explanation:



Sample 2.

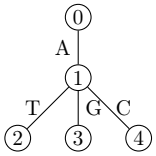
Input:

```
3
AT
AG
AC
```

Output:

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

Explanation:



Sample 3.

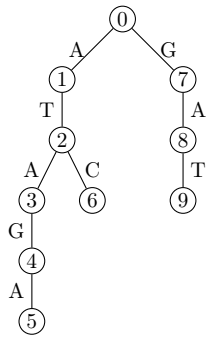
Input:

```
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
```

Explanation:



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: `trie`

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.

2 Problem: Implement TrieMatching

Problem Introduction

Given a string *Text* and *Trie*(*Patterns*), we can quickly check whether any string from *Patterns* matches a *prefix* of *Text*. To do so, we start reading symbols from the beginning of Text and see what string these symbols "spell" as we proceed along the path downward from the root of the trie, as illustrated in the pseudocode below. For each new symbol in *Text*, if we encounter this symbol along an edge leading down from the present node, then we continue along this edge; otherwise, we stop and conclude that no string in *Patterns* matches a prefix of *Text*. If we make it all the way to a leaf, then the pattern spelled out by this path matches a prefix of *Text*.

This algorithm is called PrefixTrieMatching.

```
PrefixTrieMatching(Text, Trie)
symbol ← first letter of Text
v ← root of Trie
while forever:
  if v is a leaf in Trie:
    return the pattern spelled by the path from the root to v
  else if there is an edge (v, w) in Trie labeled by symbol:
    symbol ← next letter of Text
    v ← w
  else:
    output "no matches found"
    return
```

PrefixTrieMatching finds whether any strings in *Patterns* match a prefix of *Text*. To find whether any strings in *Patterns* match a substring of *Text* starting at position k, we chop off the first $k-1$ symbols from *Text* and run PrefixTrieMatching on the shortened string. As a result, to solve the Multiple Pattern Matching Problem, we simply iterate PrefixTrieMatching $|Text|$ times, chopping the first symbol off of *Text* before each new iteration.

```
TrieMatching(Text, Trie)
while Text is nonempty:
  PrefixTrieMatching(Text, Trie)
  remove first symbol from Text
```

Note that in practice there is no need to actually chop the first $k-1$ symbols of *Text*. Instead, we just read *Text* from the k-th symbol.

Problem Description

Task. Implement TrieMatching algorithm.

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	C#	Haskell	JavaScript	Ruby	Scala
time in seconds	1	1	3	7	1.5	2	7	7	6

Memory Limit. 512Mb.

Sample 1.

Input:

```
AAA
1
AA
```

Output:

```
0 1
```

Explanation:
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:

```

```

Explanation:
There are no occurrences of the pattern in the text.

Sample 3.

Input:

```
AATCGGGTTCAATCGGGGT
2
ATCG
GGGT
```

Output:

```
1 4 11 15
```

Explanation:
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. Filename: `trie_matching`

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: Extend TrieMatching

Problem Introduction

The goal in this problem is to extend the algorithm from 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

Task. Extend TrieMatching algorithm so that it handles correctly cases when one of the patterns is a prefix of another one.

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	C#	Haskell	JavaScript	Ruby	Scala
time in seconds	1	1	3	7	1.5	2	7	7	6

Memory Limit. 512Mb.

Sample 1.

Input:

```
AAA
1
AA
```

Output:

```
0 1
```

Explanation:

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

Sample 2.

Input:

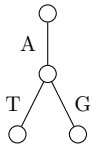
```
ACATA
3
AT
A
AG
```

Output:

```
0 2 4
```

Explanation:

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. Filename: `trie_matching_extended`

What To Do

To solve this problem, you may want to store in each node of the trie an additional flag indicating whether the path from the root to this node spells a pattern.

Need Help?

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

4 Problem: Construct the Suffix Tree of a String

Problem Introduction

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`	`C#`	`Haskell`	`JavaScript`	`Ruby`	`Scala`
time in seconds	1	1	3	10	1.5	2	10	10	6

Memory Limit. 512Mb.

Sample 1.

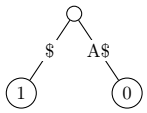
Input:

```
A$
```

Output:

```
A$
$
```

Explanation:



Sample 2.

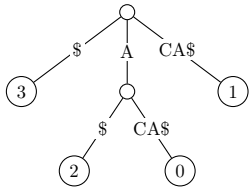
Input:

```
ACA$
```

Output:

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

Explanation:



Sample 3.

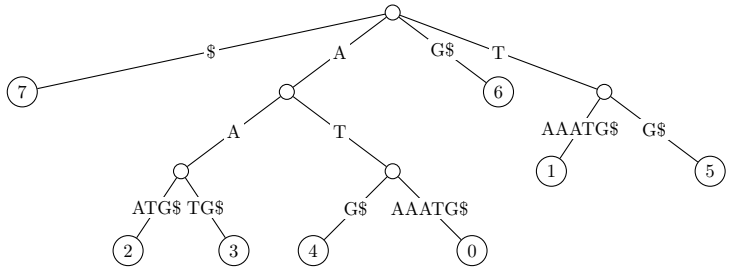
Input:

```
ATAAATG$
```

Output:

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

Explanation:



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_tree`

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.

Need Help?

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

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).

Problem Introduction

The longest repeat in a string and the longest substring shared by two strings can be found using a suffix tree. Another such problem is shown below.

Problem Description

Task. Find the shortest substring of one string that does not appear in another string.

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

Constraints. $1 \leq |Text_1|, |Text_2| \leq 2\,000$; strings have equal length ($|Text_1| = |Text_2|$), are not equal ($Text_1 \neq Text_2$), 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	C#	Haskell	JavaScript	Ruby	Scala
time in seconds	1	1	5	8	1.5	2	8	8	10

Memory Limit. 1024Mb.

Sample 1.

Input:

```
A
T
```

Output:

```
A
```

Explanation:

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

Sample 2.

Input:

```
AAAAAAAAAAAAAAAAAAA
TTTTTTTTTTTTTTTTTTT
```

Output:

```
A
```

Explanation:

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

Sample 3.

Input:

```
CCAAGCTGCTAGAGG
CATGCTGGGCTGGCT
```

Output:

```
AA
```

Explanation:
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
```

Explanation:
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. Filename: `non_shared_substring`

What To Do

Don't Treat it as a single text, this means combined suffix tree of Text1 and Text2

Hint: construct the suffix tree of a string $Text_1\#Text_2\$$ (where # and $ are new symbols).

Need Help?

https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/?ref=lbp

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

6 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.

6.1 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.

6.2 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.

6.3 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.

6.4 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 -O
```

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

```
javac -encoding UTF-8
```

- `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
```

6.5 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.

6.6 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 6.4) 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).

6.7 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 6.5. 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!

7 Frequently Asked Questions

7.1 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 6.3 and 6.4). 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.

7.2 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.

7.3 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.

7.4 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.

7.5 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.

7.6 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).

7.7 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.