Programming Assignment 2: Assembling Genomes Using de Bruijn Graphs

Revision: November 15, 2018

Introduction

Welcome to the second programming assignment of the Algorithms and Data Structures Capstone! In this assignment, you will be practicing assembling the phi X174 genome using *de Bruijn graphs*.

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 Dataset Problem: Puzzle Assembly

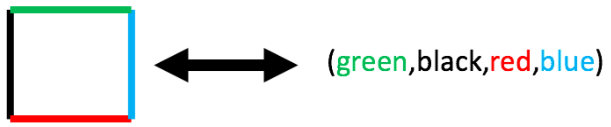
In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

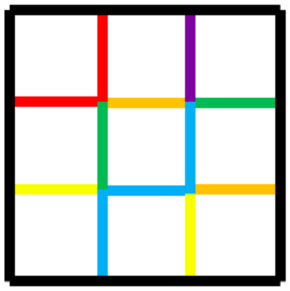
In this problem, we will consider a square puzzle consisting of n-by-n square pieces, where each square piece has a single color on each of its four edges. Given a set of n^2 square pieces, your task is to find a way to place them in an n-by-n grid such that all adjacent edges of square pieces are of the same color.

Problem Description

Task. Let each square piece be defined by the four colors of its four edges, in the format `(up,left,down,right)`. Below is an example of a square piece:



Let a "valid placement" be defined as a placement of n^2 square pieces onto an n-by-n grid such that all "outer edges" (i.e., edges that border no other square pieces), and only these edges, are black, and for all edges that touch an edge in another square piece, the two touching edges are the same color. Below is an example of a "valid placement" on a 3-by-3 square grid:



You will be given 25 square pieces in the format described above, and you will need to return a "valid placement" of them onto a 5-by-5 grid. To simplify the problem, we guarantee that all of the square pieces are given to you in the correct orientation (i.e., you will not need to rotate any of the pieces to have them fit in a "valid placement"). For example, the square `(green,black,red,blue)` and the similar square `(black,red,blue,green)` are **not** equivalent in this problem.

Dataset. Each line of the input contains a single square piece, in the format described above: `(up,left,down,right)`. You will be given 25 such pieces in total (so 25 lines of input). Note that all "outer edges" (i.e., edges that border no other square pieces on the puzzle) are black, and none of the "inner edges" (i.e., edges not on the outside border of the puzzle) are black.

Output. Output a "valid placement" of the inputted pieces in a 5-by-5 grid. Specifically, your output should be exactly 5 lines long (representing the 5 rows of the grid), and on each line of your output, you should output 5 square pieces in the **exact** format described, `(up,left,down,right)`, separated by **semicolons**. There should be no space characters at all in your output.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	10	10	15	50	50	30

Memory Limit. 512MB.

Sample 1.

Input:

```
(yellow,black,black,blue)
(blue,blue,black,yellow)
(orange,yellow,black,black)
(red,black,yellow,green)
(orange,green,blue,blue)
(green,blue,orange,black)
(black,black,red,red)
(black,red,orange,purple)
(black,purple,green,black)
```

Output:

```
(black,black,red,red);(black,red,orange,purple);(black,purple,green,black)
(red,black,yellow,green);(orange,green,blue,blue);(green,blue,orange,black)
(yellow,black,black,blue);(blue,blue,black,yellow);(orange,yellow,black,black)
```

2 Problem: Finding an Eulerian Cycle in Directed Graph

Problem Introduction

A cycle in a graph is called Eulerian if it traverses each edge of the graph exactly once. Assuming that there are no isolated vertices in a graph, it contains an Eulerian cycle if and only if it is strongly connected and for each vertex, its in-degree is equal to its out-degree. Recall from the lectures that a (circular) genome spells an Eulerian cycles in the de Bruijn graph constructed on all k-mers of the genome.

Problem Description

Task. Given a *directed* graph, find an Eulerian cycle in the graph or report that none exists.

Input Format. The first line contains integers n and m — the number of vertices and the number of edges, respectively. Each of the following m lines specifies an edge in the format "`u v`". (As usual, we assume that the vertices of the graph are $\{1, 2, \ldots, n\}$.) **The graph may contain self-loops (that is, edges of the form (v, v)) and parallel edges (that is, several copies of the same edge). It is guaranteed that the graph is strongly connected.**

Constraints. $1 \leq n \leq 10^4$; $n \leq m \leq 10^5$; $1 \leq u, v \leq n$.

Output Format. If the graph has no Eulerian cycle, output 0. Otherwise output 1 in the first line and a sequence $v_1, v_2, \ldots, v_m$ of vertices in the second line. This sequence should traverse an Eulerian cycle in the graph: $(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v_m, v_1)$ should all be edges of the graph and each edge of the graph should appear in this sequence exactly once. As usual, the graph may contain many Eulerian cycles (in particular, each Eulerian cycle may be traversed starting from any of its vertices). You may output any one of them.

Time Limits.

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

Memory Limit. 512MB.

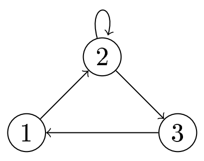
Sample 1.

Input:

```
3 4
2 3
2 2
1 2
3 1
```

Output:

```
1
1 2 2 3
```



There is an Eulerian cycle in this graph: $1 \to 2 \to 2 \to 3 \to 1$.

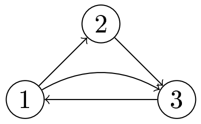
Sample 2.

Input:

```
3 4
1 3
2 3
1 2
3 1
```

Output:

```
0
```



There is no Eulerian cycle in this graph since, for example, the vertex 1 is imbalanced: its out-degree is higher that its in-degree.

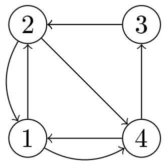
Sample 3.

Input:

```
4 7
1 2
2 1
1 4
4 1
2 4
3 2
4 3
```

Output:

```
1
4 3 2 4 1 2 1
```



There is an Eulerian cycle in this graph: $4 \to 3 \to 2 \to 4 \to 1 \to 2 \to 1 \to 4$.

Sample 4.

Input:

```
4 7
2 3
3 4
1 4
3 1
4 2
2 3
4 2
```

Output:

```
1
2 3 4 2 3 1 4
```

$$\begin{tikzcd}
2 \arrow[r] \arrow[r, bend left] & 3 \arrow[d] \arrow[dl] \\
1 \arrow[r] & 4 \arrow[ul] \arrow[ul, bend left]
\end{tikzcd}$$

There is an Eulerian cycle in this graph: $2 \to 3 \to 4 \to 2 \to 3 \to 1 \to 4$.

3 Problem: Finding a k-Universal Circular String

Problem Introduction

A k-universal circular string is a circular string that contains every possible k-mer constructed over a given alphabet.

Problem Description

Task. Find a k-universal circular binary string.

Input Format. An integer k.

Constraints. $4 \leq k \leq 14$.

Output Format. A k-universal circular string. (If multiple answers exist, you may return any one.)

Time Limits.

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

Memory Limit. 512MB.

Sample 1.

Input:

```
4
```

Output:

```
0000110010111101
```

4 Dataset Problem: Assembling the phi X174 Genome from its k-mer Composition

In dataset problems, you solution is going to be tested against a *single* dataset as opposed to problems in the previous classes in this specialization. For this reason, there is no Constraints section in the problem description below. The sample section shows just a similar dataset. Your program is not going to be tested on this dataset.

Problem Introduction

In this challenge, you will be given the task of performing Genome Assembly based on the "k-mer composition" of the genome.

Problem Description

Task. Let the "k-mer composition" of a string *Text* be defined as the list of every k-mer in *Text* (in any order). For example, the 3-mer composition of the circular string `ACGTA` is [`ACG`, `CGT`, `GTA`, `TAC`, `AAC`]. Given the k-mer composition of some unknown string, perform the task of Genome Assembly and return the circular genome from which the k-mers came. In other words, return a string whose k-mer composition is equal to the given list of k-mers.

Dataset. Each of the 5396 lines of the input contains a single k-mer. The k-mers are given to you in alphabetical order because their true order is hidden from you. Each k-mer is 10 nucleotides long.

Output. Output the assembled genome on a single line.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	3	3	4.5	15	15	9

Memory Limit. 512MB.

Sample 1.

Input:

```
AAC
ACG
CGT
GTA
TAA
```

Output:

```
ACGTA
```

5 Solving a Programming Challenge in Five Easy Steps

5.1 Reading Problem Statement

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

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

Time Limits.

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

Memory limit: 512 Mb.

5.2 Designing an Algorithm

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

5.3 Implementing an Algorithm

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

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

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

5.4 Testing and Debugging

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

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

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

5.5 Submitting to the Grading System

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

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

6 Appendix: Compiler Flags

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

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

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

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

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

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

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

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

```
nodejs
```

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

```
python2
```

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

```
python3
```

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

```
scalac
```