

Daily Programming Challenges from Reddit

February 11, 2018

Contents

1 Introduction	9
1.1 Introduction	9
1.1.1 How to Use This Book	9
1.1.2 A Note on Solutions	10
2 Easy	11
2.1 Introduction	11
2.2 3SUM	11
2.3 Abundant and Deficient Numbers	13
2.4 Detecting Alliteration	14
2.5 Anagram Detector	16
2.6 Approximate PI	17
2.7 Atbash Cipher	18
2.8 Balancing Words	20
2.9 Baum-Sweet Sequence	22
2.10 Gold and Treasure: The Beale Cipher	23
2.11 Broken Keyboard	29
2.12 Cellular Automata: Rule 90	30
2.13 Concatenated Integers	32
2.14 Condensing Sentences	33
2.15 Garland Words	34
2.16 Basic Graph Statistics: Node Degrees	35
2.17 Jolly Jumper	40
2.18 Kaprekar Numbers	41
2.19 L33tspeak Translator	42
2.20 Letters in Alphabetical Order	44
2.21 In what year were most presidents alive?	45
2.22 Making numbers palindromic	50
2.23 Pandigital Roman Numbers	51
2.24 Pronouncing Hexadecimal	51
2.25 Reverse Factorial	53
2.26 Roller Coaster Words	54
2.27 Ruth-Aaron Pairs	54
2.28 Calculating Shannon Entropy of a String	57

2.29 Shuffling a List	58
2.30 Spelling with Chemistry	60
2.31 Playing the Stock Market	63
2.32 Thue-Morse Sequence Generator	65
2.33 Vampire Numbers	66
2.34 XOR Multiplication	68
2.35 Finding Amicable Numbers	70
2.36 Making Imgur-style Links	71
2.37 Capitalize The First Letter of Every Word	72
2.38 Closest String	73
2.39 Collatz Conjecture	76
2.40 Double (or More) Knots	77
2.41 Emirp Numbers	78
2.42 Evil Numbers	79
2.43 Extravagant Numbers	80
2.44 Finding the Freshest Eggs	82
2.45 Harshad Number	83
2.46 Integer Sequence Search Part 1	84
2.47 Isomorphic Words	85
2.48 Kolakoski Sequence Generator	87
2.49 Lipogram Detector	88
2.50 Longest Palindromic Substring	90
2.51 Longest Repeated Substring	91
2.52 Narcissistic numbers	92
2.53 Regular Paperfold Sequence Generator	94
2.54 Pell Numbers	95
2.55 Perfect Numbers	96
2.56 Primes in Grids	98
2.57 Summing Across the Rudin-Shapiro Sequence	99
2.58 Safe Prime Numbers	100
2.59 Typo Maker	102
2.60 Wedderburn-Etherington Sequence	103
3 Intermediate	105
3.1 Introduction	105
3.2 Anagram Maker	105
3.3 ASCII85 Encoding and Decoding	106
3.4 ASCII Histogram Maker: Part 1 - The Simple Bar Chart	107
3.5 Where Should Grandma's House Go?	108
3.6 Connect Four	112
3.7 Calculating De Bruijn sequences	114
3.8 Detecting Four Sided Figures	115
3.9 Bioinformatics 2: DNA Restriction Enzymes	116
3.10 Elggob - Make a Boggle Layout	119
3.11 Balancing My Spending	120
3.12 Maximizing Crop Irrigation	124

3.13 Graph Radius and Diameter	127
3.14 Gray Code	131
3.15 IPv4 Subnet Calculator	133
3.16 Comparing Rotated Words	134
3.17 Linear Feedback Shift Register	135
3.18 Training for Summiting Everest	137
3.19 Generating Text with Markov Processes	138
3.20 Packing a Sentence in a Box	141
3.21 Packing Stacks of Boxes	142
3.22 Parsing Postal Addresses	144
3.23 Generating Polyominoes	145
3.24 Punch Card Creator	149
3.25 Scrabble in Reverse	151
3.26 Finding Legal Reversi Moves	153
3.27 Set Game Solver	155
3.28 Write a Web Client	157
3.29 Listening for Incoming Aircraft	159
3.30 Red Squiggles	161
3.31 Simple Stream Cipher	163
3.32 Use a Web Service to Find Bitcoin Prices	165
3.33 Word Squares Part 1	166
3.34 Worm Wars 1 - Basic Epidemiology	167
3.35 Zeckendorf Representations of Positive Integers	168
3.36 Finding Ancestors	170
3.37 Encoding with the Beale Cipher	171
3.38 Bifid Cipher	174
3.39 Sturdy Brick Walls	175
3.40 Calkin-Wilf Tree	177
3.41 Change Ringing	179
3.42 Counting Maximal Length Palindromes	180
3.43 Fencing In the Birds	181
3.44 Constructing Cyclic Numbers	182
3.45 Cyclic Words	183
3.46 Sorting Dienes Tiles	184
3.47 Tallest Tower from a List of Digits	186
3.48 The Ducci Sequence	188
3.49 The Ehrenfeucht-Mycielski sequence (0,1-version)	190
3.50 Math Snake	191
3.51 Calculating the Farey Sequence	191
3.52 Fibonacci Word Fractals	192
3.53 Friedman numbers	193
3.54 Goldbach's Weak Conjecture	194
3.55 Hitori Solver	195
3.56 Laser in a Box	196
3.57 Longest Alternating Subsequence	197
3.58 Needles in Haystacks	199

3.59 Magic Squares	200
3.60 Mathagrams	201
3.61 Picture Spot	202
3.62 Finding Numbers with Manners	203
3.63 Polydivisible Numbers	204
3.64 Primes in Several Bases	205
3.65 Singles	206
3.66 Slitherlink	207
3.67 Spinning Gears	210
3.68 Spoonerism	211
3.69 English Word Syllbalizer	212
3.70 Tile Shuffling	213
3.71 Trees in the Park	217
3.72 Two for One	218
3.73 Calculating Ulam Numbers	220
3.74 Unique County Names	221
3.75 Unwrap Some Text	222
3.76 Vaki Puzzle Solver	224
3.77 Well, Well, Well	225
3.78 Longest Word in a Box	226
3.79 XOR Decoding	228
3.80 Zombies Invaded my Village	229
4 Hard	233
4.1 Introduction	233
4.2 8 Puzzle	233
4.3 ASCII Histogram Maker: Part 2 - The Proper Histogram	234
4.4 Coiled sentence	236
4.5 Calculating Costas Arrays	237
4.6 Customer Unit Delivery Scheduling	238
4.7 DNA Shotgun Sequencing	239
4.8 Elevator Scheduling	241
4.9 Finding Friends in the Social Graph	250
4.10 Generate Strings to Match a Regular Expression	256
4.11 Golomb Rulers	257
4.12 Gophers and Robot Dogs	259
4.13 The Guards and the Mansion	260
4.14 Guarding the Coast	261
4.15 Hue Drops Puzzle	263
4.16 Kakuro Solver	264
4.17 Kanoodle Solver	265
4.18 KenKen Solver	267
4.19 Severing the Power Grid	269
4.20 Museum Cameras	273
4.21 Nonogram Solver	274
4.22 Finding Point Nemo	277

4.23 Procedural Dungeon Generation	278
4.24 Bioinformatics 3: Predicting Protein Secondary Structures	280
4.25 Rush Hour Solver	281
4.26 DNA and Protein Sequence Alignment	283
4.27 Redistricting Voting Blocks	284
4.28 Snake in a Box	286
4.29 Eight Husbands for Eight Sisters	287
4.30 Static HTTP Server	288
4.31 New York Street Sweeper Paths	290
4.32 Subset Sum Automata	291
4.33 Implementing a Templating Engine	294
4.34 Text Summarizer	296
4.35 Longest Uncrossed Knight's Path	298
4.36 Van der Waerden numbers	299
4.37 Write a Web Crawler	300
4.38 Word Squares Part 2	301
4.39 Worm Wars 2 - Network Epidemiology	303
4.40 Mission Impossible: Fooling the Anomaly Detector and Exfiltrating Data	305
4.41 Chess Solitaire	306
4.42 Calculate Graph Chromatic Number	308
4.43 Congruent Numbers	313
4.44 Glass Cutting Optimization	314
4.45 Drainage Ditches	315
4.46 Comparing Graphs	316
4.47 Buying Groceries	317
4.48 Integer Sequence Search Part 2	318
4.49 Number Grid Puzzles	320
4.50 Nurikabe Puzzle Solver	321
4.51 Implement Basic RSA Public Key Encryption	322

Chapter 1

Introduction

1.1 Introduction

I wrote this book originally inspired by the idea of a comparative languages book, comparing the approaches of multiple programming paradigms: functional, object oriented, imperative, logic (e.g. Prolog), and the like. What I wound up with, instead, was a collection of programming challenges I put together over a couple of years for the Reddit [/r/dailyprogrammer](#) community.

I wrote each of these challenges in this book, although I certainly had inspiration from elsewhere in some cases. On the one hand I own the copyright for these challenges outright. On the other hand you wind up with a slightly narrow diversity in challenges.

As you'll see, I tend to favor things like graph theory, number theory (especially the generation of integer sequences), and puzzles. One of the other moderators introduced me to computational geometry and so I later threw a few of those in there. Other moderators got me thinking about computer science and algorithm fundamentals, so there are some challenges in that vein but they're not all that common. I threw in some challenges with ciphers and encryption algorithms to help illustrate the history of the field and also how approachable encryption can be.

But overall you'll find a bunch of challenges that remain variants on a theme: dynamic programming, constraint satisfaction, and such.

1.1.1 How to Use This Book

This book isn't meant to be read end to end, but instead picked from here and there. While challenges get harder through the book - progressing from easy to intermediate to hard - they don't follow a strict set of requirements for categorizing. Don't be discouraged, sometimes you can come back to a harder one after some thought and practice.

A good thing to keep in mind is that you should your time to design a solution first, then implement it. As you progress you'll learn how to exploit the features of your language of choice and improve your coding.

1.1.2 A Note on Solutions

While working on this book, and through the Reddit community, I used these challenges to work on a few languages. As such, I wrote some code here and there to solve various challenges, although not all. Remember, I had originally planned a book comparing different programming paradigms.

Solutions to some of these challenges appear peppered throughout the book. Here are some of the languages I used while coding and a brief bit about their style and such.

Python - a popular scripting language supporting object oriented and imperative paradigms. Python is a popular language for beginners and experts alike. I tend to write small prototypes in Python. Python has a lot of built-in features in many places.

FSharp - a mixed paradigm language from Microsoft that runs on the DotNet runtime. It's essentially OCaml for DotNet, mixing imperative, objects, and functional programming paradigms. I tend to favor the functional programming aspects of the language. FSharp has become my go-to functional programming language.

Go - an imperative language from Google, written in part by some of the original C and UNIX developers.

Scala - a mix-model programming language that runs on the JVM, mixing functional and object oriented aspects. While a lot of people tend to write more compact OOP in Scala, I tend to try and use it for functional programming aspects.

Challenge Input

Our much-loved enable1.txt.

Try some other dictionaries in other languages to see if you can find the highest-degree garland word in any human language.

Bonus Challenge

Find all garland words that have at least two different degrees.

Notes

This challenge was suggested by user /u/skeeto. If you have your own idea for a challenge, submit it to /r/DailyProgrammer_Ideas, and there's a good chance we'll post it.

2.16 Basic Graph Statistics: Node Degrees

Description

In graph theory, the *degree* of a node is the number of edges coming into it or going out of it - how connected it is. For this challenge you'll be calculating the degree of every node.

Input Description

First you'll be given an integer, N , on one line showing you how many nodes to account for. Next you'll be given an undirected graph as a series of number pairs, a and b , showing that those two nodes are connected - an edge. Example:

```
3
1 2
1 3
```

Output Description

Your program should emit the degree for each node. Example:

```
Node 1 has a degree of 2
Node 2 has a degree of 1
Node 3 has a degree of 1
```

Challenge Input

This data set is an social network of tribes of the Gahuku-Gama alliance structure of the Eastern Central Highlands of New Guinea, from Kenneth Read (1954). The dataset contains a list of all of links, where a link represents signed friendships between tribes. It was downloaded from the network repository⁸.

```
16
1 2
1 3
2 3
1 4
3 4
1 5
2 5
1 6
2 6
3 6
3 7
5 7
6 7
3 8
4 8
6 8
7 8
2 9
5 9
6 9
2 10
9 10
6 11
7 11
8 11
9 11
10 11
1 12
6 12
7 12
8 12
11 12
6 13
7 13
9 13
10 13
11 13
5 14
8 14
12 14
13 14
```

⁸http://networkrepository.com/soc_tribes.php

```

1 15
2 15
5 15
9 15
10 15
11 15
12 15
13 15
1 16
2 16
5 16
6 16
11 16
12 16
13 16
14 16
15 16

```

Challenge Output

```

Node 1 has a degree of 8
Node 2 has a degree of 8
Node 3 has a degree of 6
Node 4 has a degree of 3
Node 5 has a degree of 7
Node 6 has a degree of 10
Node 7 has a degree of 7
Node 8 has a degree of 7
Node 9 has a degree of 7
Node 10 has a degree of 5
Node 11 has a degree of 9
Node 12 has a degree of 8
Node 13 has a degree of 8
Node 14 has a degree of 5
Node 15 has a degree of 9
Node 16 has a degree of 9

```

Bonus: Adjacency Matrix

Another tool used in graph theory is an *adjacency matrix*, which is an N by N matrix where each (i,j) cell is filled out with the degree of connection between nodes i and j . For our example graph above the adjacency matrix would look like this:

```

0 1 1
1 0 0
1 0 0

```

Indicating that node 1 is connected to nodes 2 and 3, but nodes 2 and 3 do not connect. For a bonus, create the adjacency matrix for the challenge graph.

Scala Solution

```

1  def degree(edges:String) =
2      edges.
3          split("\n").
4          map(_ .split(" ").filter(_ .length>0)).
5          toSet.
6          toList.
7          flatten.
8          groupBy(_ .toString).
9          mapValues(_ .size)
10
11  def adj_matrix(edges:String, n:Int):String = {
12      val m = Array.ofDim[Int](n,n)
13      val es = edges.
14          split("\n").
15          map(_ .split(" ").filter(_ .length>0)).
16          map(_ .map(_ .toInt))
17      for (e <- es) { m(e(0)-1)(e(1)-1) = 1; m(e(1)-1)(e(0)-1) = 1 }
18      m.map(_ .mkString(" ")).mkString("\n")
19  }
20
21  def challenge(edges:String) =
22      degree(edges).foreach { kv => println(kv._1 + " has a degree of " +
23      ↪ kv._2) }
24
25  def bonus(edges:String, n:Int) = {
26      challenge(edges)
27      println(adj_matrix(edges, n))
28  }

```

Go Solution

```

1  package main
2
3  import (
4      "fmt"
5      "io/ioutil"
6      "os"
7      "strconv"
8      "strings"
9  )
10
11  func check(e error) {
12      if e != nil {
13          panic(e)

```

```

14     }
15 }
16
17 func main() {
18     bdata, err := ioutil.ReadFile(os.Args[1])
19     check(err)
20
21     data := string(bdata)
22     var nodes map[string]int
23     nodes = make(map[string]int)
24
25     // calculate node degree
26     lines := strings.Split(data, "\n")
27     for _, line := range lines {
28         vals := strings.Split(line, " ")
29         if len(vals) == 2 {
30             nodes[vals[0]] = nodes[vals[0]] + 1
31             nodes[vals[1]] = nodes[vals[1]] + 1
32         }
33     }
34     i := 0
35     for k, v := range nodes {
36         fmt.Printf("Node %s has a degree of %d\n", k, v)
37         i = i + 1
38     }
39
40     // bonus adjacency matrix
41     adjm := make([][]string, i)
42     for n := range adjm {
43         adjm[n] = make([]string, i)
44         for m := range adjm[n] {
45             adjm[n][m] = "0"
46         }
47     }
48     for _, line := range lines {
49         vals := strings.Split(line, " ")
50         if len(vals) == 2 {
51             x, err := strconv.ParseUint(vals[0], 10, 32)
52             check(err)
53             y, err := strconv.ParseUint(vals[1], 10, 32)
54             check(err)
55             adjm[x-1][y-1] = "1"
56             adjm[y-1][x-1] = "1"
57             adjm[x-1][x-1] = "1"
58         }
59     }
60 }

```

```
61     for n := 0; n < i; n++ {
62         fmt.Printf("%q\n", strings.Join(adjm[n], " "))
63     }
64 }
```

2.17 Jolly Jumper

Description

A sequence of $n > 0$ integers is called a jolly jumper if the absolute values of the differences between successive elements take on all possible values through $n - 1$ (which may include negative numbers). For instance,

1 4 2 3

is a jolly jumper, because the absolute differences are 3, 2, and 1, respectively. The definition implies that any sequence of a single integer is a jolly jumper. Write a program to determine whether each of a number of sequences is a jolly jumper.

Input Description

You'll be given a row of numbers. The first number tells you the number of integers to calculate over, N , followed by N integers to calculate the differences. Example:

4 1 4 2 3
8 1 6 -1 8 9 5 2 7

Output Description

Your program should emit some indication if the sequence is a jolly jumper or not. Example:

4 1 4 2 3 JOLLY
8 1 6 -1 8 9 5 2 7 NOT JOLLY

Challenge Input

4 1 4 2 3
5 1 4 2 -1 6
4 19 22 24 21
4 19 22 24 25
4 2 -1 0 2

Challenge Output

- A “decrypt” function (or method) that takes a key and the ciphertext and returns the plaintext.

Python Solution

```

1  import sys
2
3  # def xor(b, s): return "".join(map(lambda x: chr(x^b), map(lambda x: ord(x),
4  ↪      s)))
5  def xor(b, s): return list(map(lambda x: x^b, map(lambda x: ord(x), s)))
6
7  M = sys.maxsize
8  M = 128
9
10 def lcg(m, a, c, x): return (a*x + c) % m
11
12 def enc(msg, seed):
13     res = []
14     for ch in msg:
15         res.extend(xor(lcg(M, 1664525, 1013904223, seed), ch))
16         seed = lcg(M, 1664525, 1013904223, seed)
17     return res
18
19 def dec(msg, seed):
20     res = []
21     for ch in msg:
22         res.append(lcg(M, 1664525, 1013904223, seed)^ch)
23         seed = lcg(M, 1664525, 1013904223, seed)
24     return ''.join(map(chr, res))

```

Scala Solution

```

1  def lcg(m: Int, a: Int, c: Int, x: Int) = (a*x + c) % m
2
3  def enc(s: String, key: Int): List[Int] =
4      (0 to s.length).toList.foldLeft[List[Int]](List())((acc, x) => if
5      ↪      (acc.isEmpty) {List(lcg(128, 664525, 1013904223, key))} else
6      ↪      {lcg(128, 664525, 1013904223, acc.head)::acc}).zip(s.toCharArray).map(x =>
7      ↪      x._1^x._2)
8
9  def dec(msg: List[Int], key: Int): String =
10     (0 to msg.length).toList.foldLeft[List[Int]](List())((acc, x) => if
11     ↪     (acc.isEmpty) {List(lcg(128, 664525, 1013904223, key))} else
12     ↪     {lcg(128, 664525, 1013904223, acc.head)::acc}).zip(msg).map(x =>
13     ↪     x._1^x._2).map(_._2).mkString

```

3.32 Use a Web Service to Find Bitcoin Prices

Description

Modern web services are the core of the net. One website can leverage 1 or more other sites for rich data and mashups. Some notable examples include the Google maps API which has been layered with crime data, bus schedule apps, and more.

For this challenge, you'll be asked to implement a call to a simple RESTful web API for Bitcoin pricing. This API was chosen because it's freely available and doesn't require any signup or an API key. Other APIs work in much the same way but often require API keys for use.

The Bitcoin API we're using is documented³⁰ Specifically we're interested in the `/v1/trades.csv` endpoint.

Your native code API (e.g. the code you write and run locally) should take the following parameters:

- The short name of the bitcoin market. Legitimate values are:
bitfinex bitstamp btce itbit anxhk hitbtc kraken bitkonan bitbay rock cbx cotr vcx
- The short name of the currency you wish to see the price for Bitcoin in. Legitimate values are:
KRW NMC IDR RON ARS AUD BGN BRL BTC CAD CHF CLP CNY CZK DKK EUR GAU GBP HKD HUF ILS INR JPY LTC MXN NOK NZD PEN PLN RUB SAR SEK SGD SLL THB UAH USD XRP ZAR

The API call you make to the `bitcoincharts.com` site will yield a plain text response of the most recent trades, formatted as CSV with the following fields: UNIX timestamp, price in that currency, and amount of the trade. For example:

```
1438015468,349.250000000000,0.001356620000
```

Your API should return the current value of Bitcoin according to that exchange in that currency. For example, your API might look like this (in F# notation to show types and args):

```
val getCurrentBitcoinPrice : exchange:string -> currency:string -> float
```

Which basically says take two string args to describe the exchange by name and the currency I want the price in and return the latest price as a floating point value. In the above example my code would return 349.25.

Part of today's challenge is in understanding the API documentation, such as the format of the URL and what endpoint to contact.

Note

Many thanks to /u/adrian17 for finding this API for this challenge.

³⁰<http://bitcoincharts.com/about/markets-api/>

Challenge Input

```

A, b, d, g, h, c, j, a, f, i, e
B, f, b, i, g, a, j, h, e, c, d
C, b, i, j, g, h, d, e, f, c, a
D, f, a, e, i, c, j, b, g, d, h
E, f, d, a, e, i, b, c, g, j, h
F, d, f, a, c, j, e, i, b, g, h
G, e, g, c, b, f, d, a, i, j, h
H, f, i, b, c, e, a, h, g, d, j
I, i, a, j, f, c, e, b, g, h, d
J, h, f, c, e, b, a, j, g, d, i
a, J, C, E, I, B, F, D, G, A, H
b, I, H, J, C, D, A, E, B, G, F
c, C, B, I, F, H, A, D, J, G, E
d, F, G, J, D, C, E, I, H, B, A
e, D, G, J, C, A, H, I, E, B, F
f, E, H, C, J, B, F, D, A, G, I
g, J, F, G, E, I, A, H, B, D, C
h, E, C, B, H, I, A, G, D, F, J
i, J, A, F, G, E, D, H, B, I, C
j, E, A, B, C, J, I, G, D, H, F

```

Challenge Output

```

(A; h)
(B; j)
(C; b)
(D; e)
(F; d)
(G; g)
(H; i)
(I; a)

```

4.30 Static HTTP Server

Description

I'm willing to bet most of you are familiar with HTTP, you're using it right now to read this content. If you've ever done any web programming you probably interacted with someone else's HTTP server stack - Flask, Apache, Nginx, Rack, etc.

For today's challenge, the task is to implement your own HTTP server. No borrowing your language's built in server (e.g. no, you can't just use Python's SimpleHTTPServer). The rules, requirements, and constraints:

- Your program will implement the bare basics of HTTP 1.0: GET requests required, any other methods (POST, HEAD, etc) are optional (see the bonus below).

- You have to write your own network listening code (e.g. `socket()`) and handle listening on a TCP port. Most languages support this, you have to start this low. Yep, learn some socket programming. `socket()` ... `bind()` ... `listen()` ... `accept()` ... and the like.
- Your server should handle static content only (e.g. static HTML pages or images), no need to support dynamic pages or even cgi-bin executables.
- Your server should support a document root which contains pages (and paths) served by the web server.
- Your server should correctly serve content it finds and can read, and yield the appropriate errors when it can't: 500 for a server error, 404 for a resource not found, and 403 for permission denied (e.g. exists but it can't read it).
- For it to display properly in a browser, you'll need to set the correct content type header in the response.
- You'll have to test this in a browser and verify it works as expected: content displays right (e.g. HTML as HTML, text as text, images as images), errors get handled properly, etc.

A basic, bare bones HTTP/1.0 request³⁹ looks like this;

```
GET /index.html HTTP/1.0
```

That's it, no Host header required etc., and all other headers like user-agent and such are optional. (HTTP/1.1 requires a host header, in contrast.)

A basic, bare bones HTTP/1.0 response⁴⁰ looks like this:

```
HTTP/1.0 200 OK
Content-type: text/html
```

```
<H1>Success!</H1>
```

The first line indicates the protocol (HTTP/1.0), the resulting status code (200 in this case means "you got it"), and the text of the status. The next line sets the content type for the browser to know how to display the content. Then a blank line, then the actual content. Date, server, etc headers are all optional.

Here's some basics on HTTP/1.0: <http://tecfa.unige.ch/moo/book2/node93.html>

Once you have this in your stash, you'll not only understand what more fully-featured servers like Apache or Nginx are doing, you'll have one you can customize. For example, I'm looking at extending my solution in C with an embedded Lua⁴¹ interpreter.

³⁹<https://www.w3.org/Protocols/HTTP/1.0/spec.html#Request>

⁴⁰<https://www.w3.org/Protocols/HTTP/1.0/spec.html#Response>

⁴¹<https://www.lua.org/>

Bonus

Support threading for multiple connections at once.

Support HEAD⁴² requests.

Support POST⁴³ requests.

C Solution

Years ago I wrote some C code to do this, which you can see in this gist⁴⁴. It doesn't implement error handling well, at all.

4.31 New York Street Sweeper Paths

Description

In graph theory, various walks and cycles occupy a number of theorems, lemmas, and papers. They have practical value, it turns out, in a wide variety of applications: computer networking and street sweepers among them.

For this challenge you're the commissioner of NYC street sweeping. You have to keep costs down and keep the streets clean, so you'll minimize the number of streets swept twice while respecting traffic directionality. The goal of this challenge is to visit all edges at least one while minimizing the number of streets swept to the bare minimum.

Can you find a route to give your drivers?

Input Description

Your program will be given two integers h and w on one line which tell you how tall and wide the street map is. On the next line will be a single uppercase letter n telling you where to begin. Then the ASCII map will begin using the dimensions you were given $h \times w$). Your tour should end the day at the starting point (n).

You'll be given an ASCII art graph. Intersections will be named as uppercase letters A-Z. Streets will connect them. The streets may be bi-directional ($-$ or $|$) or one-way (one of $\wedge$ for up only, $\vee$ for down only, $<$ for left only, and $>$ for right only) and you may not violate the rules of the road as the commissioner by driving down a one way street the wrong way. Bi-directional streets ($-$ or $|$) need only be visited in one direction, not both. You don't need to return to the starting point.

Output Description

Your program should emit the intersections visited in order and the number of street segments you swept.

⁴²<https://www.w3.org/Protocols/HTTP/1.0/spec.html#HEAD>

⁴³<https://www.w3.org/Protocols/HTTP/1.0/spec.html#POST>

⁴⁴<https://gist.github.com/paralax/6f57e457b5edd7b11aae2988ae8564a0>