



<2021-12-01 Mi>

Dr. Arne Babenhauserheide / [draketo.de](https://draketo.de)

# Advent of Wisp Code 2021

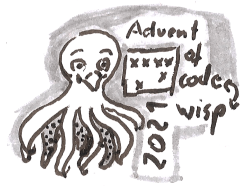
Taking part in the [advent of code](#) to relax as much as I find time to do. I'll use [Wisp](#).  
Check the [RSS-Feed](#) to get informed when I solve puzzles.  
I take them up with delay, because I work during the day and have family.  
Licenses: [cc by-sa](#) for image and text, [AGPLv3](#) or [later](#) for the code.



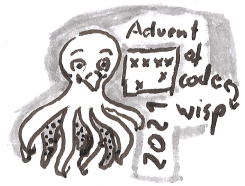


## Contents

1	Day 1, puzzle 1: Sweep the deep	4
2	Day 1, puzzle 2: Sweep the deep averages	5
3	Day 2, Puzzle 1: Pilot the submarine	6
4	Day 2, Puzzle 2: Aim the submarine	7
4.1	Update: simple shell-script . . . . .	8
5	Day 3, Puzzle 1: Diagnose a Dive	9
6	Day 3, Puzzle 2: Diagnose for Life	11
7	Day 4, Puzzle 1: Cheat the Squid	12
8	Day 4, Puzzle 2: Let the squid win	15
9	Day 5, Puzzle 1: Sidestep the vents	16
10	Day 5, Puzzle 2: Sidestep the vents diagonally	18
11	Day 6, Puzzle 1: Model Exponential Fish	19
12	Day 6, Puzzle 2: Model Exponential Fish in Memory	20
13	Day 7, Puzzle 1: Align Fuel Constrained Crab Guns	22
14	Day 7, Puzzle 2: Align Stingy Crab Guns	23
15	Day 8, Puzzle 1: Which numbers are shown?	24
16	Day 8, Puzzle 2: Which numbers are shown?	26
17	Day 9, Puzzle 1: Avoid smoke-sinks	30
18	Day 9, Puzzle 2: Discover Smoke Lakes	32
19	Day 10, Puzzle 1: Pick Wrongly Paired Parends	35
20	Day 10, Puzzle 2: Cleanly close closables	37
21	Day 11, Puzzle 1: Flashing Octopuses	38
22	Day 11, Puzzle 2: Flash together, right now	41
23	Day 12, Puzzle 1: All the exciting trails	42



<b>24 Day 12, Puzzle 2: Accept boredom just once</b>	<b>44</b>
<b>25 Day 13, Puzzle 1: Fold your password</b>	<b>48</b>
<b>26 Day 13, Puzzle 2: Fold your password</b>	<b>51</b>
<b>27 Day 14, Puzzle 1: Polymer-synthesis</b>	<b>52</b>
<b>28 Day 14, Puzzle 2: predict the element disbalance</b>	<b>55</b>
<b>29 Day 15, Puzzle 1: Path planning</b>	<b>58</b>
<b>30 Day 15, Puzzle 2: Larger path planning</b>	<b>62</b>
30.1 Simple Priority Queue . . . . .	65
30.2 Purely Functional Priority Search Queue . . . . .	69



# 1 Day 1, puzzle 1: Sweep the deep

Count how often the depths of the ocean increases.

---

```
import : srfi :1 lists
        srfi :9 records

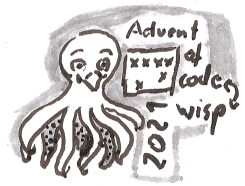
define example-input ' : 199 200 208 210 200 207 240 269 260 263

;; aggregate using a two number window.
define : count-larger current next count
  + count : if {next > current} 1 0

display
  fold count-larger 0
    ;; dropping the first element of the second list
    ;; this shifts the second element in count-larger by 1 => next
    . example-input
  drop example-input 1
```

---

For the real calculation, I plugged in the input via `define input '(...)`.  
Hacky but quick.



## 2 Day 1, puzzle 2: Sweep the deep averages

Count how often the three element moving sum of the depth increases.

---

```
import : srfi :1 lists
        srfi :9 records

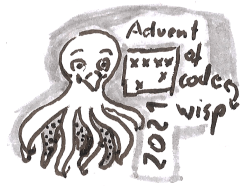
define example-input ' : 199 200 208 210 200 207 240 269 260 263

;; aggregate using a 4 number window.
define : count-larger n0 n1 n2 n3 count
  + count : if {(+ n1 n2 n3) > (+ n0 n1 n2)} 1 0

display
  fold count-larger 0
    . example-input
    drop example-input 1
    drop example-input 2
    drop example-input 3
```

---

I'm not fully happy with this code — it is longer and more complex than I'd like it to be. But it solves the problem. For a quick fix it is OK, and the adaption from puzzle 1 to puzzle 2 was easy, which is a good sign.



### 3 Day 2, Puzzle 1: Pilot the submarine

Read instructions to find the position when following them.

These look like wisp: I'm trying to turn them into code.

The input is now written to a file:

---

```
forward 5
down 5
forward 8
up 3
down 8
forward 2
```

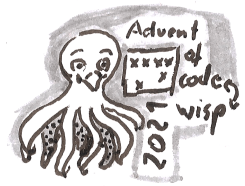
---

```
define horizontal 0
define vertical 0
define-syntax-rule : inc var steps
  set! var {var + steps}
define-syntax-rule : dec var steps
  set! var {var - steps}
define (forward steps) : inc horizontal steps
define (down steps) : inc vertical steps
define (up steps) : dec vertical steps

;; load the input as code
;; load "advent-of-wisp-code-2021-d2p1-real-input.w"
load "advent-of-wisp-code-2021-d2p1-example-input.w"

display {horizontal * vertical}
```

---



## 4 Day 2, Puzzle 2: Aim the submarine

The input is the same, but the code is different.

---

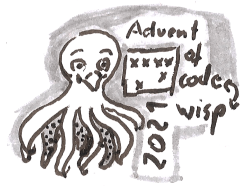
```
define aim 0
define horizontal 0
define vertical 0
define-syntax-rule : inc var steps
  set! var {var + steps}
define-syntax-rule : dec var steps
  set! var {var - steps}
;; the commands and the presence of aim are all that changes:
define (forward steps)
  inc horizontal steps
  inc vertical {aim * steps}
define (down steps) : inc aim steps
define (up steps) : dec aim steps

;; load the input as code
;; load "advent-of-wisp-code-2021-d2p1-real-input.w"
load "advent-of-wisp-code-2021-d2p1-example-input.w"

display {horizontal * vertical}
```

---

I actually like this code quite a bit, and adjusting it from puzzle 1 to puzzle 2 was a breeze. It's still a hack, though ...



## 4.1 Update: simple shell-script

While the previous version is kind of a hack (but one that uses a method I actually use [to write games](#)), it would be an even funnier hack to replace the auto-pilot with a simple shell script.

---

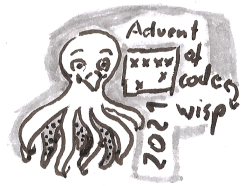
```
export AIM=0
export HORIZONTAL=0
export VERTICAL=0
function inc() {
    export ${1}=$(( ${1} + ${2} ))
}
function dec() {
    export ${1}=$(( ${1} - ${2} ))
}
function forward () {
    inc HORIZONTAL ${1}
    inc VERTICAL=$(( ${AIM} * ${1} ))
}
function down () {
    inc AIM ${1}
}
function up () {
    dec AIM ${1}
}
source "advent-of-wisp-code-2021-d2p1-example-input.w"

echo $(( ${HORIZONTAL} * ${VERTICAL} ))
```

---

Would you bet your life on it? :-)





## 5 Day 3, Puzzle 1: Diagnose a Dive

Calculate the most common bit in each position. The resulting bits give the diagnostic number  $\gamma$ . Using least common bit gives  $\epsilon$ .

Example Input:

---

```
00100
11110
10110
10111
10101
01111
00111
11100
10000
11001
00010
01010
```

---

First define a helper function that was re-used a lot later: `map-over-lines`. This receives a function and a filename and applies the function to every line read from the file.

---

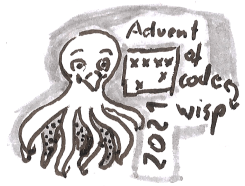
```
;; snippet: {{{map-over-lines}}}
import : only (ice-9 rdelim) read-line
define : map-over-lines fun filename
  let : : port : open-input-file filename
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
        begin
          close port
          reverse! lines
        loop
      cons : fun line
        . lines
      read-line port
```

---

Also for both tasks of day 3, I need base2 tools:

---

```
;; snippet: {{{base2-functions}}}
define : base2->number str
  . "read binary: a base2 number."
```



```
string->number str 2

define : numbers->string list-of-numbers
  string-join
  map number->string list-of-numbers
  . ""

define : numbers->decimal list-of-numbers
  base2->number
  numbers->string list-of-numbers

define : split-line-into-numbers line
  map string->number : map string : string->list line
```

---

Now the actual solution:

---

```
import : only (ice-9 rdelim) read-line

{{{map-over-lines}}}
{{{base2-functions}}}
```

```
define input
  map-over-lines split-line-into-numbers
  . "advent-of-wisp-code-2021-d3p1-example-input.dat"

define len/2 {(length input) / 2}

define most-common
  map :  $\lambda(x)$  : if {x > len/2} #\1 #\0
  apply map + input

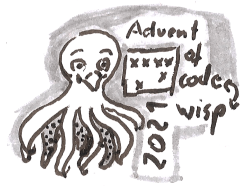
define least-common
  map :  $\lambda(x)$  : if (equal? x #\1) #\0 #\1
  . most-common

define  $\gamma$  : base2->number : apply string most-common
define  $\epsilon$  : base2->number : apply string least-common

display { $\gamma$  *  $\epsilon$ }
```

---

This is more complex than I'd like it to be. The most important missing piece in Scheme to simplify this code is "read all lines".



## 6 Day 3, Puzzle 2: Diagnose for Life

Filter the numbers bit by bit, keeping only those where the bit in the given position is the most common bit. If only one number remains, that's the result.

---

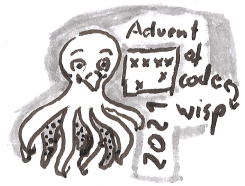
```
import : only (ice-9 rdelim) read-line
        srfi :9 records
        only (srfi :26) cut

;; using map-over-lines again, this will be used many times over
{{{map-over-lines}}}
;; base2->number, numbers->string and numbers->decimal
{{{base2-functions}}}
define input
  map-over-lines split-line-into-numbers
  . "advent-of-wisp-code-2021-d3p1-example-input.dat"
;; most- and least-common as functions to use as aggregator
define : most-common input len/2
  map : λ(x) : if {x >= len/2} 1 0
  apply map + input
define : least-common input len/2
  map : λ(x) : if {x >= len/2} 0 1
  apply map + input
define : filt input aggregator bitindex
  define len/2 {(length input) / 2}
  define aggregated : aggregator input len/2
  define : matches pattern bitindex
    equal? : list-ref pattern bitindex
            list-ref aggregated bitindex
  filter : cut matches <> bitindex
  . input

define : select aggregator
  let loop : (input (filt input aggregator 0)) (next-bitindex 1)
    if : = 1 : length input
    car input
    loop : filt input aggregator next-bitindex
          + next-bitindex 1

define oxygen : select most-common
define co2scrub : select least-common
display
  * : numbers->decimal oxygen
    numbers->decimal co2scrub
```

---



## 7 Day 4, Puzzle 1: Cheat the Squid

A squid attached to the ship. I need to cheat it in Bingo.

Known numbers that will be drawn, and bingo boards:

---

7,4,9,5,11,17,23,2,0,14,21,24,10,16,13,6,15,25,12,22,18,20,8,19,3,26,1

```
22 13 17 11 0
 8  2 23  4 24
21  9 14 16  7
 6 10  3 18  5
 1 12 20 15 19
```

```
 3 15  0  2 22
 9 18 13 17  5
19  8  7 25 23
20 11 10 24  4
14 21 16 12  6
```

```
14 21 17 24  4
10 16 15  9 19
18  8 23 26 20
22 11 13  6  5
 2  0 12  3  7
```

---

Both solutions of this day need to read the bingo board:

---

```
;; snippet: {{{bingo-board}}}
define-record-type <bingo>
  make-bingo numbers boards
  . bingo?
  numbers bingo-numbers bingo-numbers-set!
  boards bingo-boards bingo-boards-set!

define : split-bingo-line line
  if : eof-object? line
    list
    map string->number : delete "" : string-split line #\space

define bingo
  let : : port : open-input-file
    ↪ "advent-of-wisp-code-2021-d4p1-example-input.dat"
```



```
define numbers : map string->number : string-split (read-line
  ↪ port) #\,
;; skip separator line
read-line port
define boards
  let read-board : (boards '())
    if : eof-object? : peek-char port
      reverse boards
      read-board
    cons
      let loop : (board '()) (line (split-bingo-line (read-line
        ↪ port)))
        if : null? line
          reverse board
          loop : cons line board
          split-bingo-line : read-line port
      . boards
  close port
  make-bingo numbers boards

define : play-number number board
  map : λ(x) (map (λ(y) (if (equal? number y) #f y)) x)
  . board

define : board-won? board
  define : row-won? row
    every not row
  if ;; force explicit #t or #f
    or
      member #t : map row-won? board
      member #t : apply map (λ(. x) (row-won? x)) board
  . #t #f
```

---

On day one I want to win:



To cheat the squid, I need to find the sum of all the unmarked fields in the winning board (the first to have one fully marked row or column).

Then multiply it with the winning number to get the result.

---

```
import : only (ice-9 rdelim) read-line
        srfi :9 records
        only (srfi :26) cut
        only (srfi :1) every fold list-index

{{{bingo-board}}}
```

```
display
let loop : (boards (bingo-boards bingo)) (numbers (bingo-numbers
↪ bingo))
define played : map (cut play-number (car numbers) <>) boards
define result : map board-won? played
cond
  : null? numbers
  . #f
  : member #t result
  let :: winner : list-ref played : list-index (λ(x) x) result
    * : car numbers
      apply + : apply map (λ(. x) (apply + (delete #f x)))
        ↪ winner
else
  loop played
  cdr numbers
```

---



## 8 Day 4, Puzzle 2: Let the squid win

A squid attached to the ship. I need to let it win in Bingo. For sure. So I take the board that wins last.

Need to find the sum of all the unmarked fields in the winning board (the first to have one fully marked row or column).

Multiply it with the winning number.

---

```
import : only (ice-9 rdelim) read-line
        srfi :9 records
        only (srfi :26) cut
        only (srfi :1) every fold list-index remove

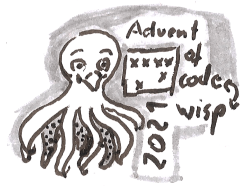
{{{bingo-board}}}
```

```
display
let loop : (boards (bingo-boards bingo)) (numbers (bingo-numbers
↪ bingo))
  define played : map (cut play-number (car numbers) <>) boards
  define result : map board-won? played
  cond
    : null? numbers
      . #f
    : every (cut equal? <> #t) result
      ;; choose the first of the last winners
      let : : winner : list-ref played 0
          * : car numbers
            apply + : apply map (λ(. x) (apply + (delete #f x)))
              ↪ winner
      else
        loop : remove board-won? played
              cdr numbers
```

---

The adjustment worked very well: the only changes are in the final let loop:

- replace loop played by loop : remove board-won? played and
- replace member #t result by every (cut equal? <> #t) result and
- always take the first of the last winners.



## 9 Day 5, Puzzle 1: Sidestep the vents

Draw lines and find meeting points.

---

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

---

```
import : only (ice-9 rdelim) read-line
        only (srfi :26) cut
        only (srfi :1) fold
        ice-9 hash-table

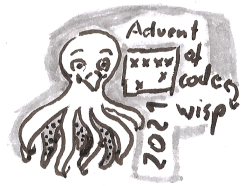
define : pixels-for-line x0 y0 x1 y1
  cond ;; only vertical and orthogonal lines
    {y0 = y1}
      map (cut cons <> y0)
          if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                      iota (+ 1 {x0 - x1}) x1
    {x0 = x1}
      map (cut cons x0 <>)
          if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                      iota (+ 1 {y0 - y1}) y1
    else '()

define : line-coordinates line
  map string->number : string-tokenize line char-set:digit

define : hash-add1 key al
  hash-set! al key : + 1 : hash-ref al key 0
  . al

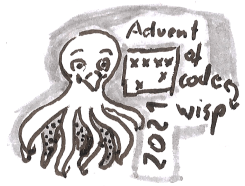
define port : open-input-file
  ↪ "advent-of-wisp-code-2021-d5p1-example-input.dot"
```





```
display
  hash-count : λ(key value) {value >= 2}
  let loop : : coordinates : make-hash-table
  define line : read-line port
  if : eof-object? line
    . coordinates
  loop
    fold hash-add1 coordinates
      apply pixels-for-line : line-coordinates line
```

---



## 10 Day 5, Puzzle 2: Sidestep the vents diagonally

Draw lines and find meeting points.

---

```
import : only (ice-9 rdelim) read-line
        only (srfi :26) cut
        only (srfi :1) fold
        ice-9 hash-table

define : pixels-for-line x0 y0 x1 y1
  cond ;; only vertical and orthogonal lines
    {y0 = y1}
      map (cut cons <> y0)
        if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                    iota (+ 1 {x0 - x1}) x1
    {x0 = x1}
      map (cut cons x0 <>)
        if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                    iota (+ 1 {y0 - y1}) y1
  else
    map cons
      if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                  iota (+ 1 {x0 - x1}) x0 -1
      if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                  iota (+ 1 {y0 - y1}) y0 -1

define : line-coordinates line
  map string->number : string-tokenize line char-set:digit
define : hash-add1 key al
  hash-set! al key : + 1 : hash-ref al key 0
  . al

define port : open-input-file
  ↪ "advent-of-wisp-code-2021-d5p1-example-input.dot"
display
  hash-count : λ(key value) {value >= 2}
  let loop : : coordinates : make-hash-table
    define line : read-line port
    if : eof-object? line
      . coordinates
    loop
      fold hash-add1 coordinates
        apply pixels-for-line : line-coordinates line
```

---



## 11 Day 6, Puzzle 1: Model Exponential Fish

Strange lanternfishes reproduce every 7 days, new fish initially reproduce after 9 days. Model the population growth.

How many will there be *after 80 days*?

Input: The time to reproduce for each fish.

---

3,4,3,1,2

---

For this data format, a simplest possible csv parser is useful. I could use [guile-dsv](#), but I want to avoid libraries here to you can run the code without installing anything but [Guile](#) and [wisp](#). So here is the simplest tool to read commaseparated numbers from a line of text:

---

```
;; snippet: {{{read-numbers-from-csv-line}}}  
define : read-numbers-from-csv-line filename  
  let : : port : open-input-file filename  
    define res : map string->number : string-split (read-line port) #\  
    close port  
    . res
```

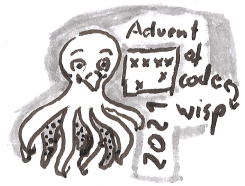
---

For 80 days, I can use a naive approach and simply keep a list of numbers with the reproduction time.

---

```
import : only (srfi :1) fold  
        only (ice-9 rdelim) read-line  
  
{{{read-numbers-from-csv-line}}}  
  
define input  
  read-numbers-from-csv-line  
  . "advent-of-wisp-code-2021-d6p1-example-input.dat"  
  
define : reproduce time-to-reproduce prev  
  if : zero? time-to-reproduce  
    cons 8 : cons 6 prev  
    cons {time-to-reproduce - 1} prev  
  
display  
length  
let rep : (steps 80) (swarm input)  
  if (zero? steps) swarm  
  rep {steps - 1} : fold reproduce '() swarm
```

---



## 12 Day 6, Puzzle 2: Model Exponential Fish in Memory

Now the goal is 256 days. That kills my memory for sure. Need a tighter datastructure. Let's use the keys for the lifetimes. The keys are contiguous integers, so why not a vector?

---

3,4,3,1,2

---

```
import : only (srfi :1) fold
        only (ice-9 rdelim) read-line

{{{read-numbers-from-csv-line}}}
```

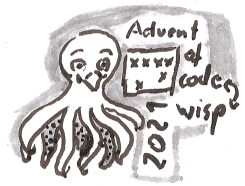
```
define input
  read-numbers-from-csv-line
  . "advent-of-wisp-code-2021-d6p1-example-input.dat"
```

```
define swarm-lifetime-counts
  let : : swarm : make-vector 9 0
      for-each : λ (x) : vector-set! swarm x : + 1 : vector-ref swarm x
        . input
      . swarm
```

```
define : reproduce swarm
  define reproducing : vector-ref swarm 0
  ;; reduce all lifetimes by 1
  for-each
    λ (lifetime)
      vector-set! swarm {lifetime - 1} : vector-ref swarm lifetime
    iota 8 1
  ;; add the reproducing to lifetime 6 and 8
  vector-set! swarm 6 : + reproducing : vector-ref swarm 6
  vector-set! swarm 8 reproducing
  . swarm
```

```
display
  apply +
  vector->list
  let rep : (steps 256) (swarm swarm-lifetime-counts)
      if (zero? steps) swarm
      rep {steps - 1} : reproduce swarm
```

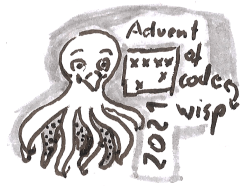
---



Since the fish with the real data are in the trillions, no way I could have done this with the plain list. Each pointer in a linked list needs around 8 byte; just the datastructure would have eaten all my memory many times over. Even a naively optimized tight array with 3-bit-numbers would not have enabled that.

With the new index-counting vector datastructure though, I can easily do 2560 steps. With the example data, the resulting number has 98 digits. 256000 steps take about a second to compute a number with 9687 digits.

Computers are fast.



## 13 Day 7, Puzzle 1: Align Fuel Constrained Crab Guns

Crabs come to blast a path into a cave. You must align them: Find the positions where they need to move the least amount of steps so their guns can interlock into one big gun.

---

16,1,2,0,4,2,7,1,2,14

---

```
import : only (ice-9 rdelim) read-line
        only (srfi :26) cut
        only (srfi :1) list-index list-ref

{{{read-numbers-from-csv-line}}}
```

```
define crabs
  read-numbers-from-csv-line
  . "advent-of-wisp-code-2021-d7p1-example-input.dat"
```

```
define min-position : apply min crabs
define max-position : apply max crabs

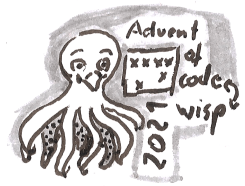
define possible-positions
  iota (+ 1 {max-position - min-position}) min-position

define : fuel-cost target-position crabs
  define : fuel-cost crab
    abs {crab - target-position}
  apply + : map fuel-cost crabs

define costs : map (cut fuel-cost <> crabs) possible-positions
define min-cost : apply min costs
define ideal-position
  list-ref possible-positions
    list-index (cut equal? min-cost <>) costs

display min-cost
```

---



## 14 Day 7, Puzzle 2: Align Stingy Crab Guns

Movement cost now increases by one per step. Step 1 is 1. Step 2 costs 2, so it is 3.  
Formula:  $(\text{step} * (\text{step} + 1)) / 2$

---

16,1,2,0,4,2,7,1,2,14

---

```
import : only (ice-9 rdelim) read-line
        only (srfi :26) cut
        only (srfi :1) list-index list-ref

{{{read-numbers-from-csv-line}}}

define crabs
  read-numbers-from-csv-line
  . "advent-of-wisp-code-2021-d7p1-example-input.dat"

define min-position : apply min crabs
define max-position : apply max crabs

define possible-positions
  iota (+ 1 {max-position - min-position}) min-position

define : fuel-cost target-position crabs
  define : distance crab
    abs {crab - target-position}
  define : cost crab
    define dist : distance crab
    * 1/2 dist {dist + 1}
  apply + : map cost crabs

define costs : map (cut fuel-cost <> crabs) possible-positions
define min-cost : apply min costs
define ideal-position
  list-ref possible-positions
  list-index (cut equal? min-cost <>) costs

display : format #f "position: ~a, cost: ~a" ideal-position min-cost
```

---



## 15 Day 8, Puzzle 1: Which numbers are shown?

I'm late on this, because a brief solution wasn't directly obvious and I didn't have much time.

I have 10 patterns and 4 displays. Four numbers use a unique number of connections:

- 1: 2
- 4: 4
- 7: 3
- 8: 7

So basically I just need to count occurrence of length of strings.

Input:

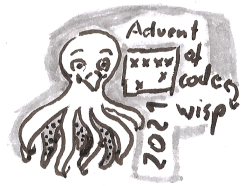
---

```
be cfbegad cbdgef fgaecd cgeb fdcge agebfd fecdb fabcd edb | fdgacbe cefdb
→ cefbgd gcbe
edbfga begcd cbg gc gcadebf fbgde acbfgd abcde gfcbcd gfec | fcgedb cgb
→ dgebacf gc
fgaebd cg bdaec gdafb agbcfd gdcbef bgcad gfac gcb cdgabef | cg cg fdcagb cbg
fbegcd cbd adcefb dageb afcb bc aefdc ecdab fgdeca fcdbega | efabcd cedba
→ gadfec cb
aebfdg fbg gf bafeg dbefa fcge gcbea fcaegb dgceab fcbdga | gecf egdcabf bgf
→ bfgea
fgeab ca afcebg bdacfe gcfdb baec bfadeg bafgc acf | gebdcfa eba ca
→ fadegcb
dbcfg fgd bdegca fgec aegbdf ecdfab fbedc dacgb gdcebf gf | cefg dcbef fcge
→ gbcadfe
bdfegc cbegaf gecbf dfcage bdacg ed bedf ced adcbefg gebcd | ed bcgafe cdgba
→ cbgef
egadfb cdbfeg cegd fecab cgb gbdefca cg fgcdab egfdb bfceg | gbdfcae bgc cg cgb
gcafb gcf dcaebfg ecagb gf abcdeg gaef cafbge fdbac febgdc | fgae cfgab fg
→ bagce
```

---

```
import : only (ice-9 rdelim) read-line
        srfi :9 records
        only (srfi :26) cut
        only (srfi :1) second
define : split-result-into-length line
        map string-length
        string-tokenize
        second : string-split line #\|
        . char-set:letter
```





```
{{map-over-lines}}}
```

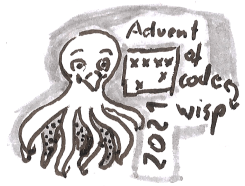
```
define input  
  apply append  
    map-over-lines split-result-into-length  
      . "advent-of-wisp-code-2021-d8p1-example-input.dat"
```

```
define counter : make-vector 8 0
```

```
for-each :  $\lambda(\text{len})$  : vector-set! counter len : + 1 : vector-ref counter  
   $\leftrightarrow$  len  
    . input
```

```
display  
  apply +  
    map (cut vector-ref counter <>)  
      list 2 4 3 7
```

---



## 16 Day 8, Puzzle 2: Which numbers are shown?

Now do the full mapping.

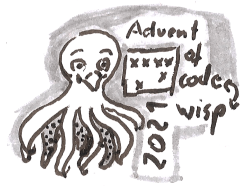
Use the left-hand patterns to recover the configuration.

---

```
import : only (ice-9 rdelim) read-line
        srfi :9 records
        only (srfi :26) cut
        only (srfi :1) first second fold assoc
        only (rnrs lists (6)) find

;;; problem definition
;; the numbers with letters for fields. The fields got scrambled.
;; 0:      1:      2:      3:      4:
;; aaaa    ....    aaaa    aaaa    ....
;; b  c  .  c  .  c  .  c  b  c
;; b  c  .  c  .  c  .  c  b  c
;; .....    ....    dddd    dddd    dddd
;; e  f  .  f  e  .  .  f  .  f
;; e  f  .  f  e  .  .  f  .  f
;; gggg    ....    gggg    gggg    ....
;;
;; 5:      6:      7:      8:      9:
;; aaaa    aaaa    aaaa    aaaa    aaaa
;; b  .  b  .  .  c  b  c  b  c
;; b  .  b  .  .  c  b  c  b  c
;; dddd    dddd    ....    dddd    dddd
;; .  f  e  f  .  f  e  f  .  f
;; .  f  e  f  .  f  e  f  .  f
;; gggg    gggg    ....    gggg    gggg

;; define number-by-length deciders
define : 1? string
  = 2 : string-length string
define : 7? string
  = 3 : string-length string
define : 4? string
  = 4 : string-length string
define : 8? string
  = 7 : string-length string
;; 6 numbers share lengths
define : 2-or-3-or-5? string
  = 5 : string-length string
define : 0-or-6-or-9? string
  = 6 : string-length string
```



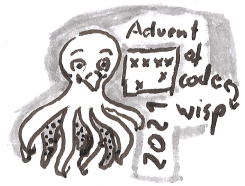
```
;;; get the input
;;; returns (pattern-part result-part)
define : split-into-strings line
  map : cut string-tokenize <> char-set:letter
        string-split line #\|

{{{map-over-lines}}}

define input-strings
  map-over-lines split-into-strings
  . "advent-of-wisp-code-2021-d8p1-example-input.dat"

define input-charsets
  map
  λ(line)
    map : λ(x) : map string->char-set x
          . line
    . input-strings

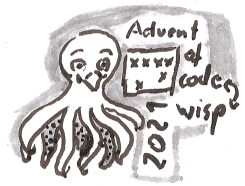
;;; Calculate and apply the de-scrambling and calculation per line
define : process-one-line line-strings line-charsets
  ;; identify the char-sets for digits of unique length
  define pattern-strings
    first line-strings
  define result-charsets
    second line-charsets
  define : find-matching-charsets string-matches? pattern-strings
    fold
      λ(string prev)
        append
          if (string-matches? string) (list (string->char-set
            → string)) '()
          . prev
    . '() pattern-strings
  define one : first : find-matching-charsets 1? pattern-strings
  define four : first : find-matching-charsets 4? pattern-strings
  define seven : first : find-matching-charsets 7? pattern-strings
  define eight : first : find-matching-charsets 8? pattern-strings
  define zero-or-six-or-nine
    find-matching-charsets 0-or-6-or-9? pattern-strings
```



```
define six
  find :  $\lambda(x)$  : char-set<= one x
    . zero-or-six-or-nine
define zero-or-nine
  filter :  $\lambda(x)$  : char-set<= one x
    . zero-or-six-or-nine
define nine
  find :  $\lambda(x)$  : char-set<= four x
    . zero-or-nine
define zero
  find :  $\lambda(x)$  : not : char-set<= four x
    . zero-or-nine
define two-or-three-or-five
  find-matching-charsets 2-or-3-or-5? pattern-strings
define three
  find :  $\lambda(x)$  : char-set<= one x
    . two-or-three-or-five
define five
  find :  $\lambda(x)$  : char-set<= x nine
    delete three two-or-three-or-five
define two
  first : delete five : delete three two-or-three-or-five

define charset-to-number
  list
    cons zero 0
    cons one 1
    cons two 2
    cons three 3
    cons four 4
    cons five 5
    cons six 6
    cons seven 7
    cons eight 8
    cons nine 9

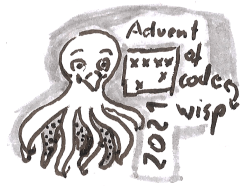
string->number
string-join
  map number->string
  map :  $\lambda(x)$  : cdr : assoc x charset-to-number char-set=
    . result-charsets
  . ""
```



```
write : apply + : map process-one-line input-strings input-charsets
```

---

This one was long, far longer than I would have liked. And with much more logic coming in from me instead of the program. I wonder if micro-/minikanren or Prolog would provide for a nicer solution.



## 17 Day 9, Puzzle 1: Avoid smoke-sinks

Find low points in height-map.

---

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (srfi :1) fold

{{{map-over-lines}}}}

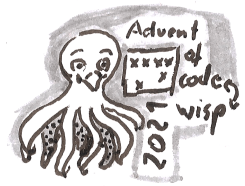
define : string-letters->numbers line
  . "turn every letter in the string into the base10 number it
    ↪ represents"
  map string->number : map string : string->list line

define input
  list->vector
  map-over-lines
  λ (line) : list->vector : string-letters->numbers line
  . "advent-of-wisp-code-2021-d9p1-example-input.dat"

define len-y : 1- : vector-length input
define len-x : 1- : vector-length : vector-ref input 0

define : at vec x y
  vector-ref (vector-ref input y) x

define : low-point? input x y
  define up : and {y > 0} {y - 1}
  define down : and {y < len-y} {y + 1}
  define left : and {x > 0} {x - 1}
  define right : and {x < len-x} {x + 1}
  define val : at input x y
  and : if up (< val (at input x up)) #t
        if down (< val (at input x down)) #t
        if left (< val (at input left y)) #t
        if right (< val (at input right y)) #t
```

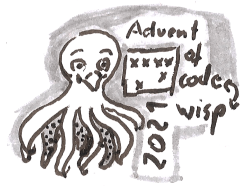


```
define risk-levels
  map
  λ(y)
    map : λ(x) : if (low-point? input x y) (+ 1 (at input x y)) 0
          iota : + 1 len-x
    iota : + 1 len-y

pretty-print
  map : λ(x) : string-join (map number->string x) ""
    . risk-levels

pretty-print : apply + : map (λ(row) (apply + row)) risk-levels
```

---



## 18 Day 9, Puzzle 2: Discover Smoke Lakes

Expand each low-point while the ground gets higher, except if it reaches 9.

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (srfi :1) fold every any lset-difference
        ↪ delete-duplicates take
        only (srfi :26) cut
        ice-9 string-fun

{{{map-over-lines}}}}

define : string-letters->numbers line
  . "turn every letter in the string into the base10 number it
    ↪ represents"
  map string->number : map string : string->list line

define input
  list->vector
  map-over-lines
  λ (line) : list->vector : string-letters->numbers line
  . "advent-of-wisp-code-2021-d9p1-example-input.dat"

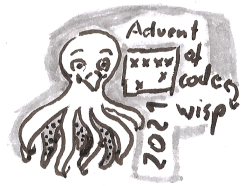
define len-y : 1- : vector-length input
define len-x : 1- : vector-length : vector-ref input 0

define : at vec x y
  vector-ref (vector-ref input y) x

define : around x y
  . "Get all points around the coordinate"
  define up : and {y > 0} {y - 1}
  define down : and {y < len-y} {y + 1}
  define left : and {x > 0} {x - 1}
  define right : and {x < len-x} {x + 1}
  delete #f
  list
    and up : cons x up
    and down : cons x down
    and left : cons left y
    and right : cons right y

define : low-point? input x y known
```





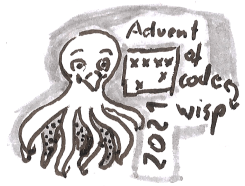
```
define val : at input x y
define : part-of-area? x y
  or : member (cons x y) known
  < val : at input x y
and {val < 9}
  every identity
  map : λ(point) : part-of-area? (car point) (cdr point)
  around x y

define low-points
  filter : λ(x) : low-point? input (car x) (cdr x) '()
  apply append
  map : λ(y) : map (cut cons <> y) : iota : + 1 len-x
  iota : + 1 len-y

define : next x y known
  define : open? point
    and : not : member point known
    . point
  delete #f : map open? : around x y

define : expand-area area
  define : expands-basin? val new
    define newval : at input (car new) (cdr new)
    < val newval 9
  define : expand-point point
    define val : at input (car point) (cdr point)
    cons point
    filter (cut expands-basin? val <>)
    next (car point) (cdr point) area
  delete-duplicates : apply append : map expand-point area

define areas
  let loop : : areas : map list low-points
  define open-points
    lset-difference equal?
    apply append
    map expand-area areas
    apply append areas
  if : null? open-points
    . areas
  loop : map expand-area areas
```



```
define : basin-value x y
  if : any identity : map (cut member (cons x y) <>) areas
    at input x y
    . 0

define area-levels
  map
  λ(y) : map (cut basin-value <> y) : iota : + 1 len-x
  iota : + 1 len-y

pretty-print
  map
  λ(x)
    string-replace-substring
    string-join (map number->string x) ""
    . "0" " " " "
    . area-levels

pretty-print : apply * : take (sort (map length areas) >) 3
```

---

This is much too long for my taste, but I don't see how to make it shorter.



## 19 Day 10, Puzzle 1: Pick Wrongly Paired Prens

Input:

---

```
[({(<(<()) []>[[{[] {<()<>}  
[(<() [;<>]])]({[<{<<[]>>(  
{([(<{ } [;<> []]>{[] { [(<()>  
((({<>}<{<{<>}&{[] { [] { }  
[[<[[[]])<([[{ } [[()]]]  
[{{{({ } } )}({[[{ { } } ) ( []  
{<[[]]>}<{[[{ [ ] { } ( [ [ ]  
[<(<(<(<{ } )>>(<[ [ ( [ ] ( )  
<{([[[[(<>()) { } ]>(<<{ {  
<{([ { } } [ <[[ [ <> { } ] ] ]> [ ] ]
```

---

Goal: Read the code and find a wrongly paired paren.

First parenthesis tools:

---

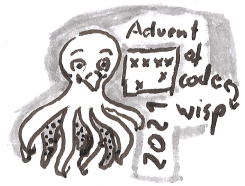
```
;; snippet: {{{paren-tools}}}  
define opening : string->char-set "[{<  
define paired  
  |  
  #\ ( . #\  
  #\ [ . #\  
  #\ { . #\  
  #\ < . #\  
  
define : opening? char  
  char-set-contains? opening char  
define : valid-char? letter-stack char  
  or : opening? char  
      equal? char : car letter-stack  
define : process letter-stack char  
  if : opening? char  
      cons (assoc-ref paired char) letter-stack  
      cdr letter-stack
```

---

Now give the correct error codes:

---

```
import : only (ice-9 rdelim) read-line  
        only (ice-9 pretty-print) pretty-print  
        only (srfi :26) cut
```



```
{{map-over-lines}}}
;; opening paired opening? valid-char? process
{{{paren-tools}}}

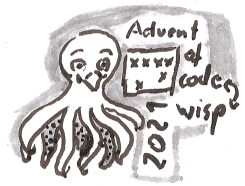
define input
  map-over-lines : λ (x) x ;; unchanged line, identity
  . "advent-of-wisp-code-2021-d10p1-example-input.dat"

define error-values
  ,
  #\ ) . 3
  #\ ] . 57
  #\ } . 1197
  #\ > . 25137

define : find-syntax-error line
  let loop : (letter-stack '()) (open (string->list line))
  cond
    : null? open
    . #f
    : valid-char? letter-stack (car open)
    loop : process letter-stack (car open)
          cdr open
  else
    car open

pretty-print
  apply +
  map (cut assoc-ref error-values <>)
  delete #f
  map find-syntax-error input
```

---



## 20 Day 10, Puzzle 2: Cleanly close closables

Close unclosed parentheses, keep score of the kind of paren closed, multiplying the previous by 5 for each new error.

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (srfi :26) cut
        only (srfi :1) fold

{{{map-over-lines}}}}
;; opening paired opening? valid-char? process
{{{paren-tools}}}}

define input
  map-over-lines : λ (x) x ;; unchanged line, identity
    . "advent-of-wisp-code-2021-d10p1-example-input.dat"

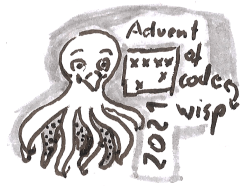
define closing-values
  '
  #\) . 1
  #\] . 2
  #\} . 3
  #\> . 4

define : score numbers
  define : add-number number prev
    + number : * 5 prev
  fold add-number 0 numbers

define : find-syntax-error line
  let loop : (letter-stack '()) (open (string->list line))
    cond
      : null? open
        score : map (cut assoc-ref closing-values <>) letter-stack
      : valid-char? letter-stack (car open)
        loop : process letter-stack (car open)
              cdr open
    else #f

pretty-print
  let : res : delete #f : map find-syntax-error input
    list-ref (sort res <) : floor/ (length res) 2
```

---



## 21 Day 11, Puzzle 1: Flashing Octopuses

Every step each number is increased by 1. If it is higher than 9, it flashes, and the up to 8 surrounding numbers increase by 1, too, also flashing if they become higher than 9.

---

```
5483143223
2745854711
5264556173
6141336146
6357385478
4167524645
2176841721
6882881134
4846848554
5283751526
```

---

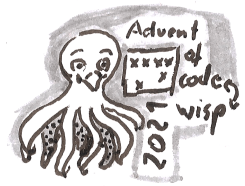
Flashing logic:

---

```
;; snippet: {{{flashing-logic}}}
define input
  map : λ (x) (map string->number (map string x))
  map-over-lines string->list
  ↪ "advent-of-wisp-code-2021-d11p1-example-input.dat"

define : 1++ arr
  . "increase every arr value by 1"
  map : λ (x) : map 1+ x
  . arr

define : flash-indizes arr
  define : flash y
    define L : list-ref arr y
    map : cut cons y <>
      delete #f
      map
        λ(idx)
          let : : num : list-ref L idx
            ;; 99 means "already flashing"
            and {num > 9} {num < 99} idx
          iota : length L
    apply append : map flash : iota : length arr
```



```
define : flash arr
  . "return as values: changed arr and count of flashing"
  let reflash : : count 0
    define indizes : flash-indizes arr
    define : apply-flash index
      define y : car index
      define x : cdr index
      define line : list-ref arr y
      define len-line-1 : 1- : length line
      define len-arr-1 : 1- : length arr
      ;; all indizes around the current index
      ;; but only inside the limits
      define around
        delete #f
        list ;; #f if outside the limits
          and {x > 0}
            cons {x - 1} y
          and {x < len-line-1}
            cons {x + 1} y
          and {y < len-arr-1}
            cons x {y + 1}
          and (and {y < len-arr-1} {x > 0})
            cons {x - 1} {y + 1}
          and (and {y < len-arr-1} {x < len-line-1})
            cons {x + 1} {y + 1}
          and {y > 0}
            cons x {y - 1}
          and (and {y > 0} {x > 0})
            cons {x - 1} {y - 1}
          and (and {y > 0} {x < len-line-1})
            cons {x + 1} {y - 1}
      for-each
        λ : x y
          let : : line : list-ref arr y
            list-set! line x : 1+ : list-ref line x
            list-set! arr y line
          map car around
          map cdr around
          ;; 99 means "already flashing"
          list-set! line x 99
      if : null? indizes
        ;; use multiple values return as a side-channel
        ;; to report the number of flashes (as count)
        values
```



```
map : λ (line) : map (λ (num) (if {num >= 99} 0 num)) line
      . arr
      . count
begin
  for-each apply-flash indizes
  reflash : + count : length indizes

define : step arr
  flash : 1++ arr



---



import : only (ice-9 rdelim) read-line
         only (ice-9 pretty-print) pretty-print
         only (srfi :26) cut
         only (srfi :1) fold first second
         only (srfi :11) let-values

{{{map-over-lines}}}
;; input, step
{{{flashing-logic}}}

define flash-counter 0

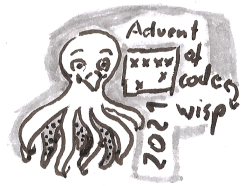
display
  string-join
  map : λ (line) : string-join (map number->string line) ""
  fold
    λ (num prev)
      let-values : : (arr count) (step prev)
        set! flash-counter {flash-counter + count}
      . arr
      . input
      iota 100
  . "\n"
newline
display flash-counter



---


```





## 22 Day 11, Puzzle 2: Flash together, right now

Find the step where all flash together.

Use a local return to stop the fold when I find the flashing.

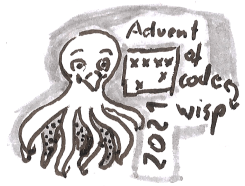
---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (ice-9 control) call/ec ; non-local exit -> return
        only (srfi :26) cut
        only (srfi :1) fold first second
        only (srfi :11) let-values

{{{map-over-lines}}}
;; input, step
{{{flashing-logic}}}

display
string-join
  map : λ (line) : string-join (map number->string line) ""
      ;; introduce a return statement locally
      call/ec
      λ : return
          fold
            λ (num prev)
              let-values : : (arr count) (step prev)
                when : = 0 : apply + : map (λ(x) (apply + x)) arr
                return : append arr `((,+ 1 num))
              . arr
            . input
          iota 1999
  . "\n"
```

---



## 23 Day 12, Puzzle 1: All the exciting trails

Find all paths through the cave that visit small caves only once.

I enter at `start`, I exit at `end`, I'm only allowed to enter uppercase rooms more than once. These are the edges (the connections) between rooms that give 10 paths:

---

```
start-A
start-b
A-c
A-b
b-d
A-end
b-end
```

---

And a larger input with 226 paths:

---

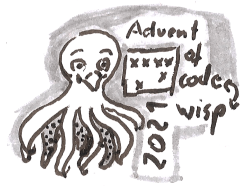
```
fs-end
he-DX
fs-he
start-DX
pj-DX
end-zg
zg-sl
zg-pj
pj-he
RW-he
fs-DX
pj-RW
zg-RW
start-pj
he-WI
zg-he
pj-fs
start-RW
```

---

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (srfi :26) cut
        only (srfi :1) first second append-map remove

{{{map-over-lines}}}
```



```
define input
  map-over-lines
  cut string-split <> #\ -
  . "advent-of-wisp-code-2021-d12p1-example-input-larger.dat"

define : undirected edges
  append-map : λ (edge) : list edge (reverse edge)
  . edges

define : all-paths edges
  let loop : (path '("start")) (edges (undirected edges))
  define matching-edges
    ;; keep the edges that match the first element of the path
    filter : λ (edge) : equal? (first edge) (first path)
    . edges
  define remaining-edges
    ;; remove edges that match the first element of the path
    ;; if we're in a lowercase field, otherwise keep all
    if : string-every char-set:upper-case (first path)
    . edges
    remove : λ (edge) : equal? (first edge) (first path)
    . edges
  define extended-paths-for-matching
    ;; create one extended path for each matching edge
    map : λ (edge) : cons (second edge) path
    . matching-edges
  define : process-one extended-path
  loop extended-path remaining-edges
  cond
    : equal? "end" : first path
    list : string-join (reverse path) ","
    : null? edges
    list ;; empty, because we did not reach the end
  else
    append-map process-one extended-paths-for-matching

pretty-print : length : all-paths input
```

---



## 24 Day 12, Puzzle 2: Accept boredom just once

Find all paths through the cave that visit small caves only once; except that you may visit one of them twice.

I enter at **start**, I exit at **end**, I'm allowed to enter uppercase rooms more than once. These are the edges (the connections) between rooms that give 36 paths:

---

```
start-A
start-b
A-c
A-b
b-d
A-end
b-end
```

---

And a larger input with 3509 paths:

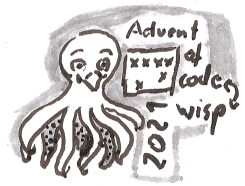
---

```
fs-end
he-DX
fs-he
start-DX
pj-DX
end-zg
zg-sl
zg-pj
pj-he
RW-he
fs-DX
pj-RW
zg-RW
start-pj
he-WI
zg-he
pj-fs
start-RW
```

---

This looks harmless, but it originally pushed my non-optimized code to its limits and got my CPU to suffer. It could benefit a lot from a functional dictionary datastructure instead of a linear-update alist. But still, it's nice to my memory and already did the job.

After finishing it, I optimized it to avoid doing work twice, so this now has reasonable speed.



```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (srfi :26) cut
        only (srfi :1) first second append-map remove

{{{map-over-lines}}}
```

```
define input
  map-over-lines
  cut string-split <> #\-
  . "advent-of-wisp-code-2021-d12p1-example-input.dat"
```

```
define : undirected edges
  append-map : λ (edge) : list edge (reverse edge)
  . edges
```

```
define : lower? str
  string-every char-set:lower-case str
```

```
define : all-paths edges
  let loop : (path ("start")) (edges (undirected edges)) (bored? #f)
  define : twice-in-path?
    if : member (first path) (cdr path)
    . #t #f
  define path-head : first path
  define start? : equal? path-head "start"
  define end? : equal? path-head "end"
  define lowercase? : lower? path-head
  define boring? : and lowercase? : twice-in-path?
  define matching-edges
    ;; keep the edges that match the first element of the path
    filter : λ (edge) : equal? path-head : first edge
    . edges
  define remaining-edges
    ;; remove edges that match the first element of the path
    ;; if we're in a lowercase field, otherwise keep all
    cond
      boring? ;; remove the current edge and all lowercase path
      ↪ elements
      let :: lowercase-path-elements : filter lower? path
      remove : λ (edge) : member (first edge)
      ↪ lowercase-path-elements
      . edges
```



```
      : or start? : and bored? lowercase?
        remove : λ (edge) : equal? path-head : first edge
          . edges
      else edges ;; keep all
define extended-paths-for-matching
  ;; create one extended path for each matching edge
  map : λ (edge) : cons (second edge) path
    . matching-edges
define : process-one extended-path
  loop extended-path remaining-edges : or bored? boring?
cond
end?
  list : string-join (reverse path) ","
: null? edges
  list ;; empty, because we did not reach the end
: and start? : not : null? : cdr path
  list ;; empty, because revisiting start is forbidden
else
  append-map process-one extended-paths-for-matching

pretty-print : length : all-paths input
```

---

Profiling this, gives the expected results: `remove`, `string-every` and `lower?` are the most expensive actions, since they are the inner loops. To profile it, just copy it into a wisp shell and then run:

```
,profile pretty-print : length : all-paths input .
```

---

The output with profile looks like this:

```
3509
%      cumulative      self
time  seconds         seconds  procedure
26.47   0.16          0.16  string-every-c-code
14.71   0.16          0.09  remove
14.71   0.11          0.09  lower?
11.76   0.07          0.07  <current input>:50:39
 8.82   0.62          0.05  <current input>:31:38:loop
 8.82   0.07          0.05  <current input>:71:0
 2.94   0.38          0.02  filter
 2.94   0.02          0.02  car
 2.94   0.02          0.02  %after-gc-thunk
 2.94   0.02          0.02  string-join
```



```
2.94      0.02      0.02  procedure?
0.00     15.13      0.00  srfi/srfi-1.scm:584:5:map1
0.00      5.55      0.00  srfi/srfi-1.scm:672:0:append-map
0.00      0.62      0.00  anon #x1752678
0.00      0.02      0.00  anon #xe3d160
0.00      0.02      0.00  ice-9/boot-9.scm:340:2:string-every
---
Sample count: 34
Total time: 0.621091946 seconds (0.093273494 seconds in GC)
```

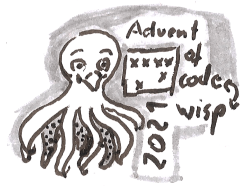
Since `string-every`, `remove` and `lower?` already have 0.34s — more than half the runtime — I won't optimize further. A better datastructure could get rid of most of the cost of `remove`, and the `lower?` could be cached to save another 10% of the runtime, for example like this:

---

```
define lower?
  let :: cache '()
      lambda (str)
        let :: cached : assoc str cache
            if cached
              cdr cached
              let :: res : string-every char-set:lower-case str
                  set! cache : cons (cons str res) cache
                  . res
```

---

See, now I did optimize further, but I'll stop here :-)  
Have fun!



## 25 Day 13, Puzzle 1: Fold your password

Mirror points over a given axis.

---

```
6,10
0,14
9,10
0,3
10,4
4,11
6,0
6,12
4,1
0,13
10,12
3,4
3,0
8,4
1,10
2,14
8,10
9,0
```

```
fold along y=7
fold along x=5
```

---

Here I need to read two fields. I realize that with a modification of `map-over-lines`:

---

```
;; snippet: {{{map-over-lines-port}}}: an alternate map-over-lines
;; that takes a port and a terminator, so multiple fields can be read
import : only (ice-9 rdelim) read-line
        only (srfi :26) cut
        only (ice-9 optargs) define*
define* : map-over-lines/port fun port #:key (terminator eof-object?)
  define terminator?
    if : or (procedure? terminator) (macro? terminator)
      . terminator
      cut equal? terminator <>
  let loop : (lines '()) (line (read-line port))
    if : terminator? line
      begin : reverse! lines
      loop : cons (fun line) lines
              read-line port
```

---





Also I need to draw coordinates on the commandline:

---

```
;; snippet: {{{draw-coordinates}}}
define : draw coordinates
  define xs : map first coordinates
  define ys : map second coordinates
  define len-x
    + 1 : apply max xs
  define len-y
    + 1 : apply max ys
  define pane
    let loop : (res '()) (rest-y len-y)
      if : zero? rest-y
        . res
        loop : cons (make-list len-x #\.) res
          - rest-y 1
  for-each : λ (x y) : list-set! (list-ref pane y) x #\#
    . xs ys
  string-join
    map : λ (sublist) : apply string sublist
    . pane
    . "\n"
```

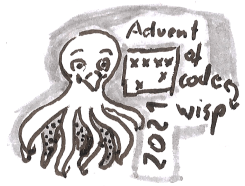
---

And I need to read the coordinates *and* instructions and apply instructions:

---

```
;; snippet: {{{coordinates-and-instructions}}}
define-values : coordinates instructions
  let : : port : open-input-file
    ↪ "advent-of-wisp-code-2021-d13p1-example-input.dat"
    define coordinates
      map-over-lines/port
      λ (line) : map string->number : string-split line #\,
        . port
        . #:terminator "" ;; split by empty line
    define instructions
      map-over-lines/port
      cut string-split <> #\=
      . port
  values coordinates instructions

define : apply-instruction coordinates instruction
  define instruction-value : string->number : second instruction
  define is-y : equal? "fold along y" : first instruction
```



```
define : process-one coordinate
  define val : if is-y (second coordinate) (first coordinate)
  define new-val
    if {val < instruction-value} val
      - {instruction-value * 2} val
  if is-y
    list (first coordinate) new-val
    list new-val (second coordinate)
map process-one coordinates
```

---

Finally: actually apply one instruction, and draw the coordinates:

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (ice-9 optargs) define*
        only (srfi :26) cut
        only (srfi :1) first second append-map remove

{{{map-over-lines-port}}}
{{{draw-coordinates}}}
{{{coordinates-and-instructions}}}

display
  string-count : draw : apply-instruction coordinates : car
  ↪ instructions
  . #\#
```

---

I did too much when solving this: I directly implemented folding to the end, because I did not read the final paragraph carefully enough.

I took that additional part out again for the code above. It's in the code for part 2.



## 26 Day 13, Puzzle 2: Fold your password

Complete folding, then use the letters as result (in the example: O). The only changes are creating folded by let-recursion over the instructions and printing the drawing instead of the count.

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (ice-9 optargs) define*
        only (srfi :26) cut
        only (srfi :1) first second append-map remove

{{{map-over-lines-port}}}
{{{draw-coordinates}}}
{{{coordinates-and-instructions}}}

define folded
  let loop : (coords coordinates) (instrs instructions)
    if : null? instrs
      . coords
      loop : apply-instruction coords : car instrs
            cdr instrs

display : draw folded
```

---

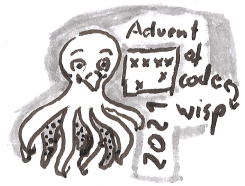
And since it just calls for it, let's follow Scheme tradition and implement `map-over-lines` on top of `map-over-lines/port`:

---

```
{{{map-over-lines-port}}}
define : map-over-lines fun filename
  define port : open-input-file filename
  define lines : map-over-lines/port fun port
  close port
  . lines
```

---

This closes the port explicitly. Without that, closing is delayed until garbage-collection which could exhaust file descriptors if I open many files in a very short time.



## 27 Day 14, Puzzle 1: Polymer-synthesis

Insert characters inside matching pairs.

The first line is the initial sequence. A second block gives a lookup table: insert the character in the middle of matching pairs.

---

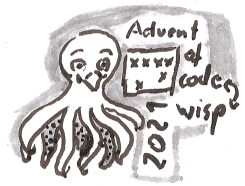
NNCB

```
CH -> B
HH -> N
CB -> H
NH -> C
HB -> C
HC -> B
HN -> C
NN -> C
BH -> H
NC -> B
NB -> B
BN -> B
BB -> N
BC -> B
CC -> N
CN -> C
```

---

```
import : only (srfi :1) take first second fold
        only (ice-9 string-fun) string-replace-substring
{{{map-over-lines-port}}}

define input
  let : : port : open-input-file
      ↪ "advent-of-wisp-code-2021-d14p1-example-input.dat"
      define init
        first : map-over-lines/port string->list port #:terminator ""
      define : split-line line
        define key-value
          string-split : string-replace-substring line " -> " ","
            . #\;
        cons : string->list : first key-value
              first : string->list : second key-value
      define rules
        map-over-lines/port split-line port
      list init rules
```



```
define : apply-rule left right prev rules
  define matching : assoc (list left right) rules
  define prev-with-match
    if matching : cons (cdr matching) prev
    . prev
  cons right prev-with-match

define : apply-rules letters rules
  reverse!
  fold (cut apply-rule <> <> <> rules)
    take letters 1 ;; initial value: first letter
    . letters ;; left letters in pairs
    cdr letters ;; shifted => right right letters

define : apply-rules-n-times N letters rules
  let loop : (N N) (letters letters)
    if : zero? N
    . letters
    loop {N - 1}
    apply-rules letters rules

define result-string
  apply string
  apply-rules-n-times 10 (first input) (second input)

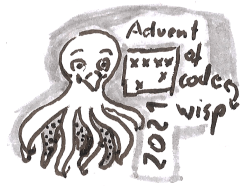
define all-possible-letters
  let : : with-duplicates : append (first input) : map cdr (second
  ↪ input)
    ;; hack: use char-set conversion to remove duplicates
  char-set->list
  list->char-set with-duplicates

define occurrences
  map (cut string-count result-string <>) all-possible-letters

let
  : maxOcc : apply max occurrences
  : minOcc : apply min occurrences
  display {maxOcc - minOcc}
```

---

This is pretty slow. At 20 steps it takes two seconds to calculate 3 million elements and at 22 steps it's already at 6 seconds for 12 million elements. The second part needs 40 steps. I must change the approach.



Also I'm still kind of annoyed that reading the input usually takes a too large fraction of the total code. I have the feeling that some primitives are too low level in Scheme — need to fix that.

Consequence: I just wrote `string-split-substring` and if/once the tests pass, I'll submit it to Guile to ease the pain and use it in the next step.



## 28 Day 14, Puzzle 2: predict the element disbalance

The answer actually only needs the counts of letters, so why should I actually synthesize the polymer-string? Just having letter-bigrams with their counts should avoid the algorithmic explosion.

But first let's simplify the string input:

---

```
;; snippet: {{{string-split-substring}}}
define : string-split-substring str substring
  if : equal? substring ""
    map string : string->list str ;; split each letter
    let : : sublen : string-length substring
      let lp : (start 0) (res '())
        cond
          (string-contains str substring start) =>
            λ : end
              lp (+ end sublen) (cons (substring/shared str start
                ↪ end) res)
          else
            reverse! : cons (substring/shared str start) res
```

---

This allows to simplify reading the input a bit:

---

```
;; before
define key-value
  string-split : string-replace-substring line " -> " ";"
  . #\;
;; after
define key-value
  string-split-substring line " -> "
```

---

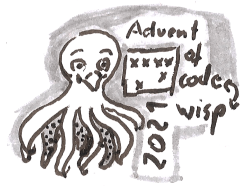
To fix the algorithmic explosion, I'll just not generate the polymer: since nature already does it, why should I do it myself when all I need are the resulting statistics? :-)

The simplest option is to use hash-maps and global mutation.

---

```
import : only (srfi :1) take first second third fold drop-right
        only (ice-9 string-fun) string-replace-substring
        only (srfi :26) cut

{{{map-over-lines-port}}}
{{{string-split-substring}}}
```



```
define letters : make-hash-table
define : letters-inc! key value
  hash-set! letters key : + value : or (hash-ref letters key) 0

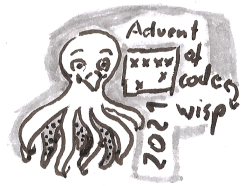
define pairs : make-hash-table
define : pairs-inc! key value
  hash-set! pairs key : + value : or (hash-ref pairs key) 0
define : pairs-dec! key value
  pairs-inc! key : - value

define input
  let : : port : open-input-file
  ↪ "advent-of-wisp-code-2021-d14p1-example-input.dat"
  ;; get the letters as before
  define init
    first : map-over-lines/port string->list port #:terminator ""
  ;; get the rules as before
  define : split-line line
    define key-value
      string-split-substring line " -> "
    cons : string->list : first key-value
          first : string->list : second key-value
  define rules
    map-over-lines/port split-line port

  ;; split the letters into pairs
  define init-pairs
    map cons
      drop-right init 1
      cdr init
  ;; track letters and pairs in the global hash-maps
  for-each (cut letters-inc! <> 1) init
  for-each (cut pairs-inc! <> 1) init-pairs
  . rules

define : apply-rule left right weight rules
define pair : cons left right
define matching : assoc (list left right) rules
when matching
  let : : middle : cdr matching
    pairs-dec! pair weight
    pairs-inc! (cons left middle) weight
    pairs-inc! (cons middle right) weight
    letters-inc! middle weight
```





```
define : apply-rules rules
  ;; get the items to fold over (avoid mutation while folding)
  define letters-and-weights
    hash-map->list : lambda (key value) : list (car key) (cdr key) value
    . pairs
  for-each (cut apply-rule <> <> <> rules)
    map first letters-and-weights ;; first
    map second letters-and-weights ;; second
    map third letters-and-weights ;; weight

define : apply-rules-n-times N rules
  ;; simplified: no need for return values
  for-each : lambda(x) : apply-rules rules
  iota N

apply-rules-n-times 40 input

define occurrences
  hash-map->list : lambda(key value) value
  . letters
let
  : maxOcc : apply max occurrences
  : minOcc : apply min occurrences
  display {maxOcc - minOcc}
```

---

This now solves my speed problems: It takes 7 seconds for 10k steps. The version from part 1 could only do 22 steps in that time.



## 29 Day 15, Puzzle 1: Path planning

Now we're getting serious. Find the path with the lowest aggregated cost of fields to enter. I need the globally best path, so the obvious choice is [Dijkstra's algorithm](#).

---

```
1163751742
1381373672
2136511328
3694931569
7463417111
1319128137
1359912421
3125421639
1293138521
2311944581
```

---

For Dijkstra I need a set of unvisited nodes and a set of visited nodes. For simplicity I'll start with plain lists of lists, the most direct translation of the task, though that will not scale, so I will likely have to change to a better datastructure in part 2.

The datastructure is a simple list of lists, defined by its accessors `xy-ref` and `xy-set!`:

---

```
;;; snippet: {{{p15-xy-ref-and-set}}}
define : xy-ref arr x y
  list-ref : list-ref arr y
    . x
define : xy-set! arr x y val
  list-set! : list-ref arr y
    . x val
```

---

In this naive approach, calculating the neighbors just looks around in the datastructure.

---

```
;;; snippet: {{{d15-neighbors}}}
define : neighbors x y
  define dpos
    ' (-1 0) (0 -1) (+1 0) (0 +1)
  delete #f
  map
    λ : dx dy
      let : (xx {x + dx}) (yy {y + dy})
        and {xx >= 0} {xx < len-x} {yy >= 0} {yy < len-y}
          pos {x + dx} {y + dy}
  map first dpos
  map second dpos
```

---



Positions are tracked as records with x and y and distances use a simple xy-structure:

---

```
;; snippet: {{{d15-position-handling}}}
define-record-type <pos>
  pos x y
  . pos?
  x pos-x
  y pos-y
;; Distances check x and y in a distances map:
define : distance node
  xy-ref distances (pos-x node) (pos-y node)
define : distance-<? A B
  <
    distance A
    distance B
define : calculate-distance node current-distance
  define X : pos-x node
  define Y : pos-y node
  define path-cost : xy-ref input X Y
  define known-distance : xy-ref distances X Y
  min known-distance {current-distance + path-cost}
```

---

Finding the closest node just iterates over all positions and keeps the unvisited one with the shortest distance.

---

```
;; snippet: {{{d15-closest-unvisited-simple}}}
define : find-closest-unvisited-node
  define len-x-1 {len-x - 1}
  define len-y-1 {len-y - 1}
  let loop : (x 0) (y 0) (closest-x 0) (closest-y 0) (closest-dist
  ↪ (inf))
    define dist : distance : pos x y
    define unvisited : not : xy-ref visited x y
    if : and unvisited {dist < closest-dist}
      loop x y x y dist
    cond
      : and {len-x-1 <= x} {len-y-1 <= y}
        if unvisited
          pos closest-x closest-y
          . #f
      {len-x-1 <= x}
        loop 0 {y + 1} closest-x closest-y closest-dist
    else : loop {x + 1} y closest-x closest-y closest-dist
```

---



And processing one node just means calculating and setting the distances of all its not yet visited neighbors:

---

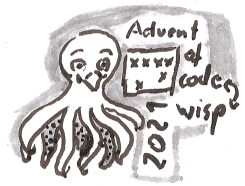
```
;; snippet: {{{d15-visit-current-node-simple}}}  
define : visit-current-node  
  define neigh ;; all unvisited neighbors  
    remove : λ (node) : xy-ref visited (pos-x node) (pos-y node)  
      neighbors (pos-x current-node) (pos-y current-node)  
  define current-distance  
    xy-ref distances (pos-x current-node) (pos-y current-node)  
  for-each  
    λ : node  
      . #f  
      xy-set! distances  
        pos-x node  
        pos-y node  
        calculate-distance node current-distance  
      . neigh  
  xy-set! visited (pos-x current-node) (pos-y current-node) #t  
  and=> (find-closest-unvisited-node) : cut set! current-node <>
```

---

To put it all together:

---

```
import : only (ice-9 rdelim) read-line  
        only (ice-9 pretty-print) pretty-print  
        only (ice-9 optargs) define*  
        only (srfi :26) cut  
        only (srfi :9) define-record-type  
        only (srfi :1) first second append-map remove  
  
{{{map-over-lines}}}  
{{{string-split-substring}}}  
  
;; use the new string-split-substring function  
define : line->numbers line  
  map string->number : string-split-substring line ""  
  
define input  
  map-over-lines line->numbers  
  ↪ "advent-of-wisp-code-2021-d15p1-example-input.dat"  
  
define len-y : length input  
define len-x : length (list-ref input 0)
```



```
{{p15-xy-ref-and-set}}
```

```
define visited
  map : λ(y) : map (λ(x) #f) : iota len-x
      iota len-y
define distances
  map : λ(y) : map (λ(x) (inf)) : iota len-x
      iota len-y
;; init the risk of the first node as 0
xy-set! distances 0 0 0
```

```
{{d15-position-handling}}
{{d15-neighbors}}
```

```
define initial-node : pos 0 0
define target-node : pos {len-x - 1} {len-y - 1}
define current-node initial-node
```

```
{{d15-closest-unvisited-simple}}
```

```
{{d15-visit-current-node-simple}}
```

```
while : visit-current-node
  . #f
```

```
;; Now the cost of all shortest paths to all nodes is known.
;; The lowest total risk is just the distance to the target
pretty-print : distance target-node
```

---

That finished with the real input in less than one minute despite the sub-par data-structures used here.



## 30 Day 15, Puzzle 2: Larger path planning

As expected, the second task has a larger map. More exactly: a 25x larger map.

I need better datastructures. But the first step is profiling:

---

```
;; add here all the code before calling visit-current-node with the real  
↪ code  
,profile while (visit-current-node) #f
```

---

% time	cumulative seconds	self seconds	procedure
71.06	27.98	27.98	list-ref
13.18	39.19	5.19	find-closest-unvisited-node
6.16	2.43	2.43	distance
5.95	15.86	2.34	xy-ref
3.38	1.33	1.33	%after-gc-thunk
0.09	39.35	0.03	visit-current-node
0.04	39.38	0.02	anon #x18f2be8
0.04	0.02	0.02	<current input>:82:0
0.04	0.02	0.02	xy-set!
0.04	0.02	0.02	ice-9/boot-9.scm:812:0:and=>
0.00	1.33	0.00	anon #xf88160
0.00	0.07	0.00	ice-9/boot-9.scm:253:2:for-each
0.00	0.07	0.00	<current input>:112:0
0.00	0.02	0.00	remove
0.00	0.02	0.00	neighbors
0.00	0.02	0.00	ice-9/boot-9.scm:230:5:map2

The culprits are obvious: `list-ref` and `find-closest-unvisited-node`.

The reason is clear: `list-ref` on the linked lists scales linearly:  $O(N)$ . Once it is taken out, the next target for optimization is `find-closest-unvisited-node`: it currently looks at all nodes, so it also scales at best in the number of `list-access`:  $O(N) * \text{list-ref}$ . Since it is needed once per node in Dijkstra, the total algorithmic cost of this naive implementation is at least cubic:

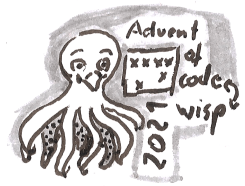
$$O(N) * \text{find-closest-unvisited-node} * \text{list-ref} \sim O(N^3)$$

But first: let's check whether just switching to a vector is enough. With 25x the nodes, quadratic scaling would mean 625x the runtime. Vectors already reduce the runtime for the unchanged input to 17s, so this could finish in 3 hours.

Change: Use 25x the nodes:

---

```
;;; snippet: {{{d15p2-5x-input}}}  
define : inc number
```



```
let : : num {number + 1}
  if {num > 9} 1 num
define : inc-list L
  map inc L

define : 5x-line line
  define nextline line
  let loop : (n 4) (line line)
    set! nextline : map inc nextline
    if : zero? n
      . line
      loop {n - 1} : append line nextline
define : 5x-arr arr
  define nextarr arr
  let loop : (n 4) (arr arr)
    set! nextarr : map inc-list nextarr
    if : zero? n
      . arr
      loop {n - 1} : append arr nextarr
;; 5x each line
set! input : map 5x-line input
;; 5x the input, in lazy mode
set! input : 5x-arr input
```

---

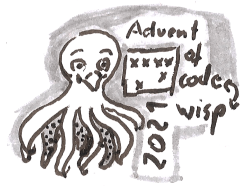
Change: Use a vector. Change the data to vectors and adjust xy-ref and xy-set!.

---

```
;; snippet: {{{d15p2-move-to-vector}}}
;; move to a vector
set! input
  list->vector : map list->vector input

define : xy-ref arr x y
  vector-ref : vector-ref arr y
  . x
define : xy-set! arr x y val
  vector-set! : vector-ref arr y
  . x val

;; move to a vector
define visited
  list->vector
  map :  $\lambda(y)$  : list->vector : map ( $\lambda(x)$  #f) : iota len-x
  iota len-y
```



```
define distances
  list->vector
  map :  $\lambda(y) : \text{list} \rightarrow \text{vector} : \text{map} (\lambda(x) (\text{inf})) : \text{iota len-x}$ 
      iota len-y
```

---

That's all the changes: the other snippets stay the same. Putting it all together (re-using snippets from part 1):

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (ice-9 optargs) define*
        only (srfi :26) cut
        only (srfi :9) define-record-type
        only (srfi :1) first second append-map remove

{{{map-over-lines}}}
{{{string-split-substring}}}

define : line->numbers line
  map string->number : string-split-substring line ""
define input
  map-over-lines line->numbers
  ↪ "advent-of-wisp-code-2021-d15p1-example-input.dat"

;; change: use 5x the input
{{{d15p2-5x-input}}}

define len-y : length input
define len-x : length (list-ref input 0)

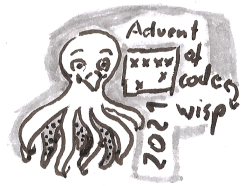
;; change: use a vector instead of a list
{{{d15p2-move-to-vector}}}

;; init the risk of the first node as 0
xy-set! distances 0 0 0

{{{d15-position-handling}}}
{{{d15-neighbors}}}

define initial-node : pos 0 0
define target-node : pos {len-x - 1} {len-y - 1}
define current-node initial-node
```





```
{{d15-closest-unvisited-simple}}  
  
{{d15-visit-current-node-simple}}  
  
while : visit-current-node  
  . #f  
  
;; Now the cost of all shortest paths to all nodes is known.  
;; The lowest total risk is just the distance to the target  
pretty-print : distance target-node
```

---

Yes, it works. Slow, but fast enough to finish. That's the power of just switching out the basic datastructure for one that's better suited for the task. Don't use a linked list, if you want to access elements at arbitrary positions by index.

With this, the task is done, but not yet done *well*.

## 30.1 Simple Priority Queue

The next step is changing `find-closest-unvisited-node` to use a priority queue.

I'll have to implement a Fibonacci heap — or one of the [other priority queues with sufficient scaling](#).

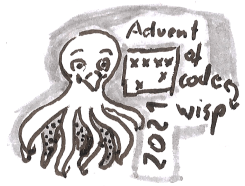
Likely I should try a Strict Fibonacci heap for the best scaling (Brodal, Gerth Stølting; Lagogiannis, George; Tarjan, Robert E., 2012), or one of the queues with best empirical results (Daniel H. Larkin, Siddhartha Sen, Robert E. Tarjan, 2014).

But that requires thinking in trees, so let's make the simplest priority queue instead. The scaling will not suffice for hard challenges, but it should suffice for this Dijkstra — and keep it simple. The data holder is a plain list for starters and ordering is done by simply sorting after every insert and searching the list linearly when decreasing, because there a value has to be moved.

The algorithm uses a slowly moving front of open nodes of roughly  $O(\sqrt{N})$  size, and it kind of follows the natural ordering of the elements, so the scaling of the priority queue for the task at hand may actually be  $O(\sqrt{N})$ .

---

```
;; snippet {{priority-queue}}  
import : only (ice-9 pretty-print) pretty-print  
        only (srfi :9) define-record-type  
        only (srfi :1) take  
  
define-record-type <queue-item>  
  queue-item priority value  
  . queue-item?  
  priority queue-item-priority queue-item-priority-set!  
  value queue-item-value
```



```
define : make-priority-queue
  . '() ;; plain linked list
define : pq-find-min q
  if : null? q
    . #f
    queue-item-value : car q
define pq-delete-min cdr
define : pq-sort q
  sort q : λ (a b) : < (queue-item-priority a) (queue-item-priority b)
define : pq-insert q q-item
  and=> (cons q-item q) pq-sort
define : pq-decrease q priority q-item-value
  . "This has linear time: O(N).
```

```
For a proper priority queue it should have  $O(\log n)$  or  $O(1)$ ."
let loop : (item (car q)) (before '()) (after '()) (rest (cdr q))
  cond
    : equal? q-item-value : queue-item-value item
      ;; use mutating functions (!) for efficiency
      append!
      reverse! before
      cons : queue-item priority q-item-value
      reverse! after
      . rest
    : null? rest
      error "item not found in q:" q-item-value : take q 10
      ;; the <= is required to have stable sorting.
      { (queue-item-priority item) <= priority }
      loop : car rest
      cons item before
      . after
      cdr rest
  else
    loop : car rest
    . before
    cons item after
    cdr rest
```

---

Now I use the priority queue to track the unvisited nodes and get the closest one:

```
;; snippet {{{priority-queue-usage-unvisited}}}
;; priority-queue: unvisited (this should decrease the cost)
```



```
define unvisited
  append-map
    λ (x)
      map
        λ (y) : queue-item (inf) : pos x y
          iota len-y
      iota len-x

;; With the priority queue, this is down to a single line.
define : find-closest-unvisited-node
  pq-find-min unvisited
```

---

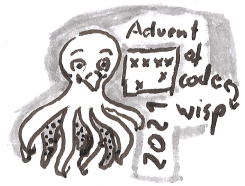
Processing the current node not only sets the by position, but also changes the distance inside the priority queue and takes the next node to process from it:

```
;; snippet {{{priority-queue-usage-visit-current-node}}}
define : visit-current-node
  define neigh ;; all unvisited neighbors
    remove : λ (node) : xy-ref visited (pos-x node) (pos-y node)
      neighbors (pos-x current-node) (pos-y current-node)
  define current-distance
    xy-ref distances (pos-x current-node) (pos-y current-node)
  for-each
    λ : node
      let : : d : calculate-distance node current-distance
        set! unvisited : pq-decrease unvisited d node
        xy-set! distances
          pos-x node
          pos-y node
          . d
      . neigh
  xy-set! visited (pos-x current-node) (pos-y current-node) #t
  ;; delete the current-node
  set! unvisited : pq-delete-min unvisited
  and=> (find-closest-unvisited-node) : cut set! current-node <>
```

---

Putting the full Dijkstra together again, now with the priority queue:

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
```



```
only (ice-9 optargs) define*
only (srfi :26) cut
only (srfi :9) define-record-type
only (srfi :1) first second append-map remove take

{{{map-over-lines}}}
{{{string-split-substring}}}
{{{priority-queue}}}

define : line->numbers line
  map string->number : string-split-substring line ""

define input
  map-over-lines line->numbers
  ↪ "advent-of-wisp-code-2021-d15p1-example-input.dat"

{{{d15p2-5x-input}}}

define len-y : length input
define len-x : length (list-ref input 0)

{{{d15p2-move-to-vector}}}

;; init the risk of the first node as 0
xy-set! distances 0 0 0

{{{d15-position-handling}}}

{{{d15-neighbors}}}

define initial-node : pos 0 0
define target-node : pos {len-x - 1} {len-y - 1}
define current-node initial-node

{{{priority-queue-usage-unvisited}}}
set! unvisited : pq-decrease unvisited 0 : pos 0 0

{{{priority-queue-usage-visit-current-node}}}

define : cost-to-target
  while : visit-current-node
    . #f
  distance target-node
```



```
;; Now the cost of all shortest paths to all nodes is known.  
;; The lowest total risk is just the distance to the target  
pretty-print : cost-to-target
```

---

Using this trivial priority queue, we're down from 2h with the naive search on the raw map-data to 11 minutes. The algorithmic cost is dominated by pq-decrease, so using a better priority-queue could decrease the cost a lot:

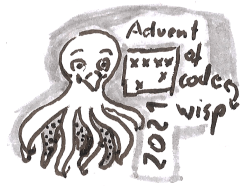
```
,profile cost-to-target .  
%      cumulative    self  
time  seconds      seconds  procedure  
81.49   883.98    734.70  pq-decrease  
  6.49    58.50    58.50  %after-gc-thunk  
  6.45    58.11    58.11  reverse!  
  3.63    32.70    32.70  equal?  
  1.73    15.58    15.56  append!  
  0.03   900.08     0.30  <current input>:185:0  
  0.03     0.28     0.28  <current input>:203:8  
  0.03     0.24     0.24  xy-ref  
  0.02   901.51     0.19  visit-current-node  
  0.02     0.15     0.15  <current input>:160:0  
  0.01     0.13     0.13  xy-set!  
  0.01     0.56     0.11  neighbors  
  0.01     0.54     0.11  ice-9/boot-9.scm:230:5:map2  
  0.01     0.09     0.09  min  
  0.01   900.25     0.06  ice-9/boot-9.scm:253:2:for-each  
  0.01     0.06     0.06  cadr  
  0.00     0.26     0.04  ice-9/boot-9.scm:220:5:map1  
  0.00     0.04     0.04  pq-delete-min
```

## 30.2 Purely Functional Priority Search Queue

Redefining the priority queue with the Priority Search Queue from the [Purely Functional Data Structures](#) should give us much better performance. Let's try that:

---

```
;; snippet {{{priority-queue-pfds}}}  
import : only (ice-9 pretty-print) pretty-print  
        only (srfi :26) cut  
        only (srfi :9) define-record-type  
        only (srfi :1) take  
        pfds psqs ;; Priority Search Queue  
  
define-record-type <queue-item>
```



```
queue-item priority value
. queue-item?
priority queue-item-priority queue-item-priority-set!
value queue-item-value
;; the position comparator defines the traversal order
define : pos-<? p0 p1
  define x0 : pos-x p0
  define y0 : pos-y p0
  define x1 : pos-x p1
  define y1 : pos-y p1
  if {x0 = x1} {y0 < y1} {x0 < x1}
define : make-priority-queue
  make-psq pos-<? <
define pq-find-min psq-min
define pq-delete-min psq-delete-min
define pq-delete psq-delete
define pq-insert psq-set
define : pq-decrease q priority q-item-value
  psq-update q q-item-value
  λ (prio) : min priority prio
  . priority
```

---

We'll have to adjust the creation of the unvisited node queue:

```
;; snippet {{{priority-queue-usage-unvisited-pfds}}}
;; priority-queue: unvisited (this should decrease the cost)
define unvisited
  fold : λ (next previous) : pq-insert previous next : inf
  make-priority-queue
  append-map
  λ (x)
    map
    λ (y) : pos x y
    iota len-y
  iota len-x
;; With the priority queue, this is down to a single line.
define : find-closest-unvisited-node
  and
  not : psq-empty? unvisited
  pq-find-min unvisited
```

---



And delete the minimum node before adjusting other nodes to avoid an infinite loop:

---

```
;; snippet {{{priority-queue-usage-visit-current-node-pfds}}}
define : visit-current-node
  define neigh ;; all unvisited neighbors
    remove : λ (node) : xy-ref visited (pos-x node) (pos-y node)
      neighbors (pos-x current-node) (pos-y current-node)

  define current-distance
    xy-ref distances (pos-x current-node) (pos-y current-node)
  ;; delete the current-node from unvisited
  set! unvisited : pq-delete-min unvisited

  for-each
    λ : node
      let : : d : calculate-distance node current-distance
        set! unvisited : pq-decrease unvisited d node
        xy-set! distances
          pos-x node
          pos-y node
          . d
      . neigh

  xy-set! visited (pos-x current-node) (pos-y current-node) #t
  and=> (find-closest-unvisited-node) : cut set! current-node <>
```

---

But overall the changes are pretty minor:

---

```
import : only (ice-9 rdelim) read-line
        only (ice-9 pretty-print) pretty-print
        only (ice-9 optargs) define*
        only (srfi :26) cut
        only (srfi :9) define-record-type
        only (srfi :1) first second append-map remove take fold
        pfds psqs ;; Priority Search Queue

{{{map-over-lines}}}
{{{string-split-substring}}}

define : line->numbers line
  map string->number : string-split-substring line ""
```



```
define input
  map-over-lines line->numbers
  ↪ "advent-of-wisp-code-2021-d15p1-example-input.dat"

{{{d15p2-5x-input}}}}

define len-y : length input
define len-x : length (list-ref input 0)

{{{d15p2-move-to-vector}}}}

;; init the risk of the first node as 0
xy-set! distances 0 0 0

{{{d15-position-handling}}}}

{{{priority-queue-pfds}}}}
{{{priority-queue-usage-unvisited-pfds}}}}
{{{d15-neighbors}}}}

define initial-node : pos 0 0
define target-node : pos {len-x - 1} {len-y - 1}
define current-node initial-node

set! unvisited : pq-decrease unvisited 0 : pos 0 0

{{{priority-queue-usage-visit-current-node-pfds}}}}

define : cost-to-target
  while : visit-current-node
    . #t
    distance target-node

;; Now the cost of all shortest paths to all nodes is known.
;; The lowest total risk is just the distance to the target
pretty-print : cost-to-target
```

---

This code now takes only **10 seconds** for completion instead of **8 minutes**.

*The difference of the previous version compared to the earlier results are due to faster hardware.*

```
wisp@(guile-user)> ,profile cost-to-target
...
```





...

%	cumulative	self	
time	seconds	seconds	procedure
29.36	4.10	4.10	struct-ref
15.65	4.08	2.19	ice-9/boot-9.scm:1211:8
14.46	6.33	2.02	rnrs/records/procedural.scm:141:6
5.37	1.89	0.75	ice-9/boot-9.scm:978:0:record-type-parents
4.32	0.75	0.60	ice-9/boot-9.scm:1169:19
4.17	0.71	0.58	pfds/psqs.sls:235:0:update
3.28	7.37	0.46	pfds/psqs.sls:149:0:play-match
2.68	2.69	0.37	pfds/psqs.sls:359:0:balance
2.53	0.35	0.35	<
2.24	0.31	0.31	%after-gc-thunk

...

---

Sample count: 671

Total time: 13.97066953 seconds (4.1994665 seconds in GC)

I cannot fix `struct-ref`, so I finally consider this task done. Though it is now 30 months my first try.

## List of Links

draketo.de: <a href="https://www.draketo.de">https://www.draketo.de</a> . . . . .	1
advent of code: <a href="https://adventofcode.com/2021/">https://adventofcode.com/2021/</a> . . . . .	1
Wisp: <a href="https://www.draketo.de/software/wisp.html">https://www.draketo.de/software/wisp.html</a> . . . . .	1
RSS-Feed: <a href="https://www.draketo.de/software/advent-of-wisp-code-2021.xml">https://www.draketo.de/software/advent-of-wisp-code-2021.xml</a> . . . . .	1
cc by-sa: <a href="https://creativecommons.org/licenses/by-sa/4.0/">https://creativecommons.org/licenses/by-sa/4.0/</a> . . . . .	1
AGPLv3 or later: <a href="https://www.gnu.org/licenses/agpl">https://www.gnu.org/licenses/agpl</a> . . . . .	1
to write games: <a href="https://hg.sr.ht/~arnebab/dryads-wake/">https://hg.sr.ht/~arnebab/dryads-wake/</a> . . . . .	8
guile-dsv: <a href="https://github.com/artiom-poptsov/guile-dsv/">https://github.com/artiom-poptsov/guile-dsv/</a> . . . . .	19
Guile: <a href="https://www.gnu.org/software/guile/">https://www.gnu.org/software/guile/</a> . . . . .	19
wisp: <a href="https://www.draketo.de/software/wisp.html">https://www.draketo.de/software/wisp.html</a> . . . . .	19
Dijkstra's algorithm: <a href="https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm">https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm</a> . . . . .	58
other priority queues with sufficient scaling: <a href="https://en.wikipedia.org/wiki/Fibonacci_heap#Summary_of_running_times">https://en.wikipedia.org/wiki/Fibonacci_heap#Summary_of_running_times</a> . . . . .	65
Brodal, Gerth Stølting; Lagogiannis, George; Tarjan, Robert E., 2012: <a href="https://www.cs.au.dk/~gerth/papers/stoc12.pdf">https://www.cs.au.dk/~gerth/papers/stoc12.pdf</a> . . . . .	65
Daniel H. Larkin, Siddhartha Sen, Robert E. Tarjan, 2014: <a href="https://arxiv.org/abs/1403.0252">https://arxiv.org/abs/1403.0252</a> . . . . .	65
Purely Functional Data Structures: <a href="https://github.com/ijp/pfds">https://github.com/ijp/pfds</a> . . . . .	69