Egel Solutions to Advent of Code 2024

Marco Devillers

Day 1

```
# Advent of Code (AoC) - day 1, task 2

import "prelude.eg"
using System, OS, List

def input =
    let L = read_line stdin in if eof stdin then {} else {L | input}

def parse =
    do Regex::matches (Regex::compile "[0-9]+") |> map to_int

def tally =
    do map Dict::count |> reduce Dict::inner_join |> Dict::to_list

def main =
    input |> map parse |> transpose |> tally |> map [(X,(Y,Z)) -> X*Y*Z] |> sum
```

Advent of Code day 1.

Day 2

```
# Advent of Code (AoC) - day 2, task 2

import "prelude.eg"

using System, OS, List

def input =
    let L = read_line stdin in if eof stdin then {} else {L | input}

def parse =
    do Regex::matches (Regex::compile "[0-9]+")  |> map to_int

def safe =
    [ XX ->
    [ XX -> or (all (flip elem {-1,-2,-3}) XX) (all (flip elem {1,2,3}) XX) ]
            (zip_with (-) XX (tail XX)) ]
```

```
def dampened =
    [ XX -> zip_with (++) (inits XX) (tail (tails XX)) ]

def main =
    input |> map parse |> map dampened |> map (any safe)
          |> filter id |> length
```

Advent of Code day 2.

Day 3

```
# Advent of Code (AoC) - day 3, task 2

import "prelude.eg"

using System, OS, List

def input =
    let L = read_line stdin in if eof stdin then {} else {L | input}

def parse =
    do Regex::matches (Regex::compile "mul\\([0-9]+,[0-9]+\\)|do(n't)?\\(\\)")

def args =
    do Regex::matches (Regex::compile "[0-9]+") |> map to_int

def calc =
    foldl_state
    [ _ N "don't()" -> (false, N)
    | _ N "do()"    -> (true, N)
    | true N X      -> (true, product (args X) + N)
    | false N X     -> (false, N) ] true 0

def main =
    input |> foldl (+) "" |> parse |> calc
```

Advent of Code day 3.

Day 4

```
# Advent of Code (AoC) - day 4, task 2

import "prelude.eg"
```

```
using System, OS, List, String (to_chars, from_chars)

val star = {(-1, -1), (0,0), (1, 1), (1, -1), (0,0), (-1,1)}

def words =
    [D -> map (flip map star . add) (Dict::keys D) |> map (map (Dict::get_with_default
```

```
def main =
    read_lines stdin |> map to_chars |> Dict::from_lists |> words |> map from_chars
    |> filter (flip elem {"MASMAS", "MASSAM", "SAMMAS", "SAMSAM"}) |> length
```

Advent of Code day 4.

Day 5

```
# Advent of Code (AoC) - day 5, task 2

import "prelude.eg"

using System, OS, List

def parse =
    do Regex::matches (Regex::compile "[0-9]+") |> map to_int

def order =
    foldl [P {X,Y} -> [A B -> (not (and (X == B) (Y == A))) && [_ -> P A B]]] [_ _ -> tr
```

```
def main =
    read_lines stdin |> map parse |> span (do length |> ((==) 2)) |> proj_update 1 tail
    |> [(PP,XX) -> filter (join ((/=) . sort_by (order PP))) XX |> map (sort_by (order P
    |> map (join (nth . flip (/) 2 . length)) |> sum
```

Advent of Code day 5.

Day 6

```
# Advent of Code (AoC) - day 6, task 2

import "prelude.eg"

using System, OS, List, String (to_chars, from_chars), D = Dict

val dirs = {(-1, 0), (0, 1), (1, 0), (0,-1)}
```

```
def start_pos =
    [D -> foldl [P0 P1 -> if D::get D P1 == '^' then P1 else P0] (0,0) (D::keys D)]

def track =
    [D S -> trace_while
        [(P,N) -> D::has D P]
        [(P,N) -> [Q -> if D::has D Q && [_ -> D::get D Q == '#'] then (P,(N+1)%4) else
        (S,0)]

def loops =
    [D S -> let V = D::dict in iter_while
        [(P,N) -> (not (D::has V (P,N))) && [_ -> (D::has D P)]]
        [(P,N) -> D::set V (P,N) 0; [Q -> if D::has D Q && [_ -> D::get D Q == '#'] then
        (S,0) |> D::has V]

def solve =
    [D -> let S = start_pos D in foldl [N P -> if [B -> D::set D P '.';B] (loops (D::set
          (map fst (track D S) |> tail |> unique)]

def main =
    read_lines stdin |> map to_chars |> D::from_lists |> solve
```

Advent of Code day 6.

Day 7

```
# Advent of Code (AoC) - day 7, task 2

import "prelude.eg"

using System, OS, List

def parse =
    do Regex::matches (Regex::compile "[0-9]+") |> map to_int

def conc =
    [X Y -> to_int (to_text X + to_text Y)]

def solutions =
    foldl [{} X -> {X} |XX X -> map ((*) X) XX ++ map ((+) X) XX ++ map (flip conc X) XX

def main =
    read_lines stdin |> map parse |> map [XX -> (head XX, solutions (tail XX))]
    |> filter [(X,XX) -> (filter ((==) X) XX) /= {}] |> map fst |> sum
```

Advent of Code day 7.

Day 8

```
# Advent of Code (AoC) - day 8, task 2

import "prelude.eg"

using System, OS, List, String (to_chars, from_chars)

def antennas =
    do Dict::to_list |> filter [(_,'.') -> false | _ -> true]

def combs =
    do fix [F {} -> {} |F {X|XX} -> map [Y -> (X,Y)] XX ++ F XX]
       |> filter [((P0,A0),(P1,A1)) -> (A0 == A1)]

def cast =
    [D P V -> if Dict::has D P then {P|cast D (add P V) V} else {}]

def antinodes =
    [ D -> do combs
       |> foldl [AA ((P0, A0),(P1,A1)) -> cast D P0 (sub P0 P1) ++ cast D P0 (sub P1 P0)
       |> unique ]

def main =
    read_lines stdin |> map to_chars |> Dict::from_lists
    |> [ D -> antennas D |> antinodes D ]
    |> length
```

Advent of Code day 8.

Day 9

```
# Advent of Code (AoC) - day 9, task 2

import "prelude.eg"

using System, OS, List, String (to_chars, from_chars)

def to_fs =
    do foldl_state [(0, N) XX X -> ((1, N+1), {(X, N)|XX})
                   |(1, N) XX X -> ((0, N), {(X, none)|XX})] (0,0) {}
       |> reverse
```

```
def wipe =
    [I {(J,Y)|XX} -> if I == Y then {(J,none)|XX} else {(J,Y)|wipe I XX}]

def place =
    [(I,X) {(J,none)|XX} -> if I <= J then {(I,X),(J-I,none)|wipe X XX} else {(J,none)|p
    |(I,X) {(J,Y)|XX} -> if X == Y then {(J,Y)|XX} else {(J,Y)|place (I,X) XX}]

def compact =
    [XX -> foldl [XX (_,none) -> XX|XX F -> place F XX] XX (reverse XX)]

def main =
    read_line stdin |> to_chars |> map to_int
    |> to_fs |> compact
    |> foldl_state [N M (L,none) -> (N+L,M)|N M (L,F) -> (N+L, M + F*((N+L)*(N+L- 1)/2-(
```

Advent of Code day 9.

Day 10

```
# Advent of Code (AoC) - day 10, task 2

import "prelude.eg"

using System, OS, List, String (to_chars, from_chars)

def dirs = {(-1,0),(1,0),(0,-1),(0,1)}

def starts = do Dict::to_list |> filter (do snd |> ((==)'0')) |> map fst |> map singletc

def trails =
    [D -> iter 9 (flatmap [PP -> map (add (head PP)) dirs |> map (flip cons PP)
            |> filter [{P,Q|PP} -> Dict::get_safe D P == succ (Dict::get D Q)]])]

def main =
    read_lines stdin |> map to_chars |> Dict::from_lists
    |> [D -> starts D |> trails D ] |> length
```

Advent of Code day 10.

Day 11

```
# Advent of Code (AoC) - day 10, task 2
```

```
import "prelude.eg"

using System, OS, List, String (to_chars, from_chars)

def dirs = {(-1,0),(1,0),(0,-1),(0,1)}

def starts = do Dict::to_list |> filter (do snd |> ((==)'0')) |> map fst |> map singleto

def trails =
    [D -> iter 9 (flatmap [PP -> map (add (head PP)) dirs |> map (flip cons PP)
            |> filter [{P,Q|PP} -> Dict::get_safe D P == succ (Dict::get D Q)]])]

def main =
    read_lines stdin |> map to_chars |> Dict::from_lists
    |> [D -> starts D |> trails D ] |> length
```

Advent of Code day 11.

Day 12

```
# Advent of Code (AoC) - day 12, task 2

import "prelude.eg"

using System, OS, List, String (to_chars), D = Dict

def dirs = {(-1,0),(1,0),(0,-1),(0,1)}

def regions0 =
    [D C PP {} RR -> (PP, RR)
    |D C PP QQ RR ->
        let QQ = flatmap [P -> map (add P) QQ] dirs |> unique |> filter (D::has D) in
        let (PP0, RR) = split [P -> (D::get D P == C) && [_ -> elem P QQ]] RR in
            regions0 D C (PP0++PP) PP0 RR ]

def regions =
    [D {} -> {}
    |D {P|PP} -> [(PP,QQ) -> {PP|regions D QQ}] (regions0 D (D::get D P) {P} {P} PP)]

def perimeter =
    [D PP -> filter (flip not_elem PP) (flatmap [P -> map (add P) dirs] PP)]

def sides0 =
    [PP -> map (flip tuple 0) PP |> D::from_list |> [D -> regions D PP]]
```

```
def sides =
    [PP -> flatmap [P -> map (add P) PP |> filter (flip not_elem PP) |> sides0] dirs]

def main =
    read_lines stdin |> map to_chars |> D::from_lists
    |> [D -> regions D (D::keys D) |> map [PP -> (PP, sides PP)]]
    |> map [(PP0,PP1) -> (length PP0) * (length PP1)] |> sum
```

Advent of Code day 12.

Day 13

```
# Advent of Code (AoC) - day 13, task 2

import "prelude.eg"

using System, OS, List

def parse = do Regex::matches (Regex::compile "[0-9]+") |> map to_int |> list_to_tuple

def solve =
    [O {(AX,AY), (BX,BY), (PX,PY)} ->
        let (PX,PY) = add O (PX,PY) in
        let M = ((PX * BY) - (PY * BX)) / ((AX * BY) - (AY * BX)) in
        let N = (PY - AY * M) / BY in
        if (PX,PY) == add (mul M (AX,AY)) (mul N (BX,BY)) then (M,N) else none]

def main =
    read_lines stdin |> map parse |> split_on tuple
    |> map (solve (10000000000000, 10000000000000))
    |> filter ((/=) none) |> map [(M,N) -> 3 * M + N] |> sum
```

Advent of Code day 13.

Day 14

```
# Advent of Code (AoC) - day 14, task 2

import "prelude.eg"

using System, OS, List

def parse = do Regex::matches (Regex::compile "-?[0-9]+") |> map to_int
            |> chunks 2 |> map list_to_tuple |> list_to_tuple
```

```
def size = (101,103)

def mod = [N M -> ((N%M)+M)%M]

def walk = [N (P,V) -> add P (mul N V) |> [(BX, BY) (PX,PY) -> (mod PX BX, mod PY BY)] s

def quadrants = [(BX,BY) PP ->
        {filter [(PX,PY) -> and (PX < (BX/2)) (PY < (BY/2))] PP,
         filter [(PX,PY) -> and (PX < (BX/2)) (PY > (BY/2))] PP,
         filter [(PX,PY) -> and (PX > (BX/2)) (PY < (BY/2))] PP,
         filter [(PX,PY) -> and (PX > (BX/2)) (PY > (BY/2))] PP} ]

def main =
    read_lines stdin |> map parse
    |> [PP -> map [N -> (map (walk N) PP, N)] (from_to 0 (uncurry (*) size))]
    |> map (proj_update 0 (product . map length . quadrants size))
    |> reduce [(I,N) (J,M) -> if I < J then (I,N) else (J,M)]
    |> snd
```

Advent of Code day 14.

Day 15

```
# Advent of Code (AoC) - day 15, task 2

import "prelude.eg"

using System, OS, List, String (to_chars), D = Dict

def parse = do map to_chars |> split_on {} |> [{XX,YY} -> (XX, reduce (++) YY)]

def dir = ['^' -> (-1,0) |'v' -> (1,0)|'<' -> (0,-1)|'>' -> (0,1)]

def expand = flatmap ['@' -> {'@','.'}|'O' -> {'[',']'}|'.' -> {'.','.'}|'#'->{'#','#'}]

def start = do D::to_list |> filter ((==) '@' . snd) |> head |> fst

def cat = [none F -> none | XX F -> [none -> none| YY -> XX++YY] (F none)]

def region = [D P V -> let Q = add P V in
    [(_,0) '[' -> cat {P, add (0,1) P} [_ -> cat (region D Q V) [_ -> region D (add (0,1
    |(_,0) ']' -> cat {P, add (0,-1) P} [_ -> cat (region D Q V) [_ -> region D (add (0,
    |(0,_) '[' -> cat {P, Q} [_ -> region D (add (0,1) Q) V]
    |(0,_) ']' -> cat {P, Q} [_ -> region D (add (0,-1) Q) V]
```

```
    |_     '@' -> cat {P} [_ -> region D (add P V) V]
    |_     '#' -> none
    |_     _   -> {}] V (D::get D P)]

def shove =
    [D PP V -> foldl [D (P,X) -> D::set D P X] D
            (map [P -> (P,'.')] PP ++ map [P -> (add P V, D::get D P)] PP)]

def step = [D P V -> [none -> (D,P)|PP -> (shove D PP V, add P V)] (region D P V)]

def main =
    read_lines stdin |> parse |> [(XX,VV) ->(D::from_lists (map expand XX), map dir VV)]
    |> [(D,VV) -> foldl [(D,P) V -> step D P V] (D, start D) VV] |> fst
    |> D::to_list |> foldl [N ((X,Y),'[') -> N + 100 * X + Y |N _ -> N] 0
```

Advent of Code day 15.

Day 16

```
# Advent of Code (AoC) - day 16, task 2

import "prelude.eg"

using System, OS, List, String (to_chars, from_chars), D = Dict

def pos = [C -> do D::to_list |> filter ((==) C . snd) |> head |> fst]

def dirs = {(0,1),(1,0),(0,-1),(-1,0)}

def rotate = [(0,Y) -> {(Y,0),(-Y,0)} | (X,0) -> {(0,X),(0,-X)}]

def insort = [P {} -> {P}|P {Q|QQ} -> if proj 0 P <= proj 0 Q then {P,Q|QQ} else {Q|insc

def dijkstra0 =
    [ G {} (D0,D1) -> (D0,D1)
    | G {(N,P)|QQ} (D0,D1) ->
        let ADJ = D::get G P in
        let (D0,D1,QQ) = foldl [(D0,D1,QQ) (M,Q) ->
                        let ALT = N + M in
                        if ALT < D::get_with_default max_int D0 Q then
                        (D::set D0 Q ALT, D::set D1 Q {P}, insort (ALT,Q) QQ)
                        else if ALT == D::get D0 Q then
                        (D::set D0 Q ALT, D::set D1 Q (unique {P|D::get D1 Q}), QQ)
                        else (D0,D1,QQ)] (D0,D1,QQ) ADJ
         in dijkstra0 G QQ (D0,D1)]
```

```
def dijkstra = [G P -> dijkstra0 G {(0,P)} (D::set D::dict P 0, D::set D::dict P {})]

def adj =
    [D (P,V) -> {(1,(add P V,V))} ++ map [V -> (1001, (add P V,V))] (rotate V)
            |> filter ((/=) '#' . D::get D . fst . snd)]

def to_graph =
    [D -> foldl [G (P,'#') -> G
                |G (P,_) -> foldl [G (P,V) -> D::set G (P,V) (adj D (P,V))] G
                                  (map (tuple P) dirs)] D::dict (D::to_list D)]

def nodes =
    [D PP {} -> PP
    |D PP {Q|QQ} -> nodes D {Q|PP} (D::get_with_default {} D Q ++ QQ)]

def main =
    read_lines stdin |> map to_chars |> D::from_lists
    |> [D -> let S = pos 'S' D in let E = pos 'E' D in
        to_graph D |> [G -> dijkstra G (S,(0,1))]
        |> [(D0,D1) ->
               map [P -> (D::get_with_default max_int D0 P,P)] (map (tuple E) dirs)
               |> [PP -> filter ((==) (minimum (map fst PP)) . fst) PP |> map snd]
               |> nodes D1 {} ]
        |> map fst |> unique |> length]
```

Advent of Code day 16.

Day 17

```
# Advent of Code (AoC) - day 17, task 2

import "prelude.eg"

using System, OS, List

def parse = do foldl (+) "" |> (do Regex::matches (Regex::compile "-?[0-9]+") |> map to_
            |> [{A,B,C|P} -> ((A,B,C),P)]

def op = [N RR -> if N < 4 then N else proj (N - 4) RR]

def ins =
    [_ _ {} -> {}| _ _ {X} -> {}
    |PP (A,B,C) {0,N|XX} -> ins PP (A/(Math::pow_int 2 (op N (A,B,C))),B,C) XX
    |PP (A,B,C) {1,N|XX} -> ins PP (A,N^B,C) XX
```

```
    |PP (A,B,C) {2,N|XX} -> ins PP (A,(op N (A,B,C))%8,C) XX
    |PP (0,B,C) {3,N|XX} -> ins PP (0,B,C) XX
    |PP (A,B,C) {3,N|XX} -> ins PP (A,B,C) (drop N PP)
    |PP (A,B,C) {4,N|XX} -> ins PP (A,B^C,C) XX
    |PP (A,B,C) {5,N|XX} -> {(op N (A,B,C))%8| ins PP (A,B,C) XX}
    |PP (A,B,C) {6,N|XX} -> ins PP (A,A/(Math::pow_int 2 (op N (A,B,C))),C) XX
    |PP (A,B,C) {7,N|XX} -> ins PP (A,B,A/(Math::pow_int 2 (op N (A,B,C)))) XX]

def run = [(RR,PP) -> ins PP RR PP]

def iter_with = [0 F X -> X|N F X -> iter_with (N - 1) F (F N X)]

def find = [PP -> iter_with (length PP) [L NN -> flatmap
            [(N,I) -> if run ((8*N+I,0,0),PP) == (drop (L - 1) PP) then {8*N+I} else {}]
            (flatmap [N -> map (tuple N) (from_to 0 7)] NN) ] {0}]

def main = read_lines stdin |> parse |> [(RR,PP) -> find PP] |> minimum
```

Advent of Code day 17.

Day 18

```
# Advent of Code (AoC) - day 18, task 2

import "prelude.eg"

using System, OS, List, D = Dict

def dirs = {(0,1),(1,0),(0,-1),(-1,0)}

def board =
    [(X,Y) PP ->
        let F = [C -> foldl [D P -> D::set D P C]] in
        F '#' (F '.' D::dict (flatmap [X -> map (tuple X) (from_to 0 Y)] (from_to 0 X)))

def adj =
    [D P -> map (add P) dirs |> filter [P -> D::has D P && [_ -> D::get D P /= '#']]]

def graph =
    [D -> foldl [G (P,'#') -> D::set G P {}|G (P,_) -> D::set G P (adj D P)] D::dict (D

def reachable =
    [G V {} -> V
    |G V XX -> reachable G (foldl [V X -> D::set V X 0] V XX)
                           (flatmap (D::get G) XX |> unique |> filter (not . D::has V))]
```

```
def remove =
    [G B -> foldl [G (A,B) -> D::update G [BB -> {B|BB}] A] G
            ((map (add B) dirs) |> filter (D::has G) |> [BB -> (map (tuple B) BB) ++ (ma

def search =
    [G V E {B|BB} ->
        let G = remove G B in let V = reachable G V (map (add B) dirs |> filter (D::has
        if D::has V E then B else search G V E BB]

def main =
    let S = (0,0) in let E = (70,70) in
    read_lines stdin |> map (list_to_tuple . map to_int . Regex::matches (Regex::compile
    |> [BB -> board E BB |> graph |> [G -> search G (reachable G D::dict {S}) E (reverse
```

Advent of Code day 18.

Day 19

```
# Advent of Code (AoC) - day 19, task 2

import "prelude.eg"

using System, OS, List, String (starts_with, remove, count, split_pattern), D = Dict

def match =
    [X Y -> if starts_with X Y then remove 0 (count X) Y else none]

def solve =
    [XX _  D "" -> 1
    |XX {} D Z -> 0
    |XX {Y|YY} D Z -> [none -> solve XX YY D Z |Z0 -> (D::memo D (solve XX XX) Z0) + (so

def main =
    read_lines stdin |> split_on "" |> [{{XX},YY} -> (split_pattern ", " XX, YY)]
    |> [(XX, YY) -> map [Y -> solve XX XX D::dict Y] YY |> sum]
```

Advent of Code day 19.

Day 20

```
# Advent of Code (AoC) - day 20, task 2

import "prelude.eg"
```

```
using System, OS, List, String (to_chars, from_chars), D = Dict

def pos = [C -> do D::to_list |> filter ((==) C . snd) |> head |> fst]

def dirs = {(0,1),(1,0),(0,-1),(-1,0)}

def adj = [D P -> map (add P) dirs |> filter ((/=) '#' . D::get D)]

def to_graph = [D -> foldl [G (P,'#') -> G |G (P,_) -> D::set G P (adj D P)] D::dict (D:

def bfs =
    [G N D {} -> D
    |G N D PP -> bfs G (N+1) (foldl [D P -> D::set D P N] D PP)
                             (flatmap (D::get G) PP |> filter (not . D::has D))]

def manhattan = [P Q -> sub P Q |> [(X,Y) -> abs0 X + abs0 Y]]

def skip = [N {} -> {} |N {X|XX} -> (map (tuple X) XX |> filter [(P, Q) -> manhattan P (

def saved = [D (P,Q) -> abs0 ((D::get D P) - (D::get D Q)) - manhattan P Q]

def main =
    read_lines stdin |> map to_chars |> D::from_lists
    |> [D -> bfs (to_graph D) 0 D::dict {pos 'E' D}]
    |> [D -> map (saved D) (skip 20 (D::keys D)) |> filter (flip (>=) 100) |> length]
```

Advent of Code day 20.

Day 21

```
# Advent of Code (AoC) - day 21, task 2

import "prelude.eg"

using System, OS, List, S = String, D = Dict

def dirs = ['<' -> (0,-1)|'>' -> (0,1)|'^' -> (-1,0)|'v' -> (1,0)|'A' -> (0,0)]

def to_keypad = do D::from_lists |> D::to_list |> map swap |> D::from_list |> flip D::er

val numeric = to_keypad {{'7','8','9'},{'4','5','6'},{'1','2','3'},{' ','0','A'}}
val digital = to_keypad {{' ','^','A'},{'<','v','>'}}

def buttons =
```

```
    [(0,0) -> {}
    |(0,Y) -> if Y < 0 then {'<'|buttons (0,Y+1)} else {'>'|buttons (0,Y - 1)}
    |(X,Y) -> if X < 0 then {'^'|buttons (X+1,Y)} else {'v'|buttons (X - 1,Y)}]

def presses =
    [D (N, T, {A}) -> 0
    |D (N, T, {A,B|BB}) ->
        let K = (if N==T then numeric else digital) in
        if N == 0 then
             let M = length (buttons (sub (D::get K B) (D::get K A))) + 1 in
             M + D::memo D presses (N, T, {B|BB})
        else let PP = permutations (buttons (sub (D::get K B) (D::get K A))) |> unique |
             let PP = filter [BB -> all (flip elem (D::values K)) (scanl add (D::get K A
             let M = map [BB -> D::memo D presses (N - 1, T, {'A'|BB})] PP |> minimum in
             M + D::memo D presses (N, T, {B|BB}) ]

def main =
    read_lines stdin |> map [X -> (to_int X, S::to_chars X)]
    |> (let M = D::dict in map [(N, BB) -> (N, presses M (25, 25, {'A'|BB}))])
    |> map (uncurry (*)) |> sum
```

Advent of Code day 21.

Day 22

```
# Advent of Code (AoC) - day 22, task 2

import "prelude.eg"

using System, OS, List, D = Dict

def step =
    [N -> let F = [G N -> ((G N)^N)%16777216] in F ((*) 64) N |> F ((flip (/) 32)) |> F

def trace = [0 F X -> {} |N F X -> {X|trace (N - 1) F (F X)}]

def group =
    [{(I,A),(J,B),(K,C),(L,D)|XX} -> {((A << 12) + (B << 8) + (C << 4) + D,L)|group {(J,
    |_ -> {}]

def prices =
    do trace 2000 step |> map (flip (%) 10) |> [XX -> zip (tail XX) (zip_with (-) (tail

def count = [D XX ->
    foldl [D XX -> let V = D::dict in
```

```
            foldl [D (T,N) -> if D::has V T then D else D::set V T 0; D::set_with D (+) T
        D XX]

    def main =
        read_lines stdin |> map to_int |> map prices |> map group
        |> count D::dict |> D::values |> maximum
```

Advent of Code day 22.

Day 23

```
    # Advent of Code (AoC) - day 23, task 2

    import "prelude.eg"

    using System, OS, List, D = Dict

    def graph =
        let F = [D V0 V1 -> D::set_with D [XX YY -> unique (XX++YY)] V0 {V1}] in
        foldl [D (V0,V1) -> F (F D V0 V1) V1 V0] D::dict

    def adj = D::get_with_default {}
    def vertices = D::keys

    def bron_kerbosch0 =
        [G (R,{},{}, RR) -> (R,{},{},{R}++RR)
        |G (R, P, X, RR) ->
            foldl
            [(R,P,X,RR) V ->
                let R0 = union {V} R in
                let P0 = intersection P (adj G V) in
                let X0 = intersection X (adj G V) in
                let (_,_,_,RR) = bron_kerbosch0 G (R0,P0,X0,RR) in
                    (R, difference P {V}, union X {V}, RR)] (R,P,X,RR) P]

    def bron_kerbosch = [G -> bron_kerbosch0 G ({},vertices G,{},{}) |> proj 3]

    def main =
        read_lines stdin |> map (Regex::matches (Regex::compile "[a-z]+")) |> map list_to_tu
        |> graph |> bron_kerbosch |> sort_by [XX YY -> length XX > length YY] |> head
        |> sort |> reduce [S0 S1 -> S0 + "," + S1]
```

Advent of Code day 23.

Day 24

```
# Advent of Code (AoC) - day 24, task 2

import "prelude.eg"
using System, OS, List, S = String, D = Dict

def xor = [A B -> and (or A B) (not (and A B))]

def assign =
    foldl [D (X,OP,Y,Z) -> D::set D Z (X, OP, Y)|D (X,Y) -> D::set D X Y|D _ -> D]

def eval0 =
    [D Z -> let F = [A B -> D::set D A B; D] in let G = [A -> D::get D A] in
        [(X,"AND",Y) -> eval0 D X; eval0 D Y;F Z (and (G X) (G Y))
        |(X,"OR",Y) ->  eval0 D X; eval0 D Y;F Z (or (G X) (G Y))
        |(X,"XOR",Y) -> eval0 D X; eval0 D Y;F Z (xor (G X) (G Y))
        |"0" -> F Z false|"1" -> F Z true|_ -> D] (D::get D Z)]

def eval = [D -> foldl [D Z -> eval0 D Z] D (D::keys D)]

def num_in = [N S X -> if N < 0 then {} else {(format "{}{:02}" S N,(X&(1<<N)) /= 0)|num
def num_out = [D N S -> if N < 0 then 0 else
        ([B -> if B then 1<<N else 0] (D::get D (format "{}{:02}" S N))) $ (num_out D (N

def swap = [D A B -> let C = D::get D A in D::set D A (D::get D B);D::set D B C]
def swaps = [D P -> foldl [D (X,Y) -> swap D X Y] (D::copy D) P]

val test_cases = let RNG = [N -> ((N * 1103515245) + 12345) & ((1 << 32) - 1)] in
    map [(X,Y) -> (RNG X, RNG Y)] (zip (from_to 501 600) (from_to 1201 1300))
def test = [D X Y -> let D = eval (assign (assign (Dict::copy D) (num_in 44 "x" X)) (num
            let X = num_out D 44 "x" in
            let Y = num_out D 44 "y" in
            let Z = num_out D 45 "z" in
            printf "{} + {} = {}\n" X Y Z; (X + Y) == Z]
def tests = [D {} -> true |D {(X,Y)|RR} -> if test D X Y then tests D RR else false]
def valid = [D -> tests D test_cases]

def oracle = read_line (open_in "oracle.txt") |> S::split_pattern " " |> chunks 2 |> map

def main =
    read_lines stdin |> map (Regex::matches (Regex::compile "[a-zA-Z0-9]+")) |> map list
    |> assign D::dict |> flip swaps oracle |> valid
```

Advent of Code day 24.

Day 25

```
# Advent of Code (AoC) - day 25, task 1

import "prelude.eg"

using System, OS, List, S = String, D = Dict

def heights =
    do transpose |> map (flip (-) 1 . length . filter ((==) '#'))

def fit =
    [(L,K) -> all (flip (<=) 5) (zip_with (+) L K)]

def main =
    read_lines stdin |> map S::to_chars |> split_on {} |> split [XX -> all ((==) '#') (h
    |> [(XX,YY) -> pairs (map heights XX) (map heights YY)]
    |> filter fit |> length
```

Advent of Code day 25.

Running times

```
day  1 - 141ms | *****
day  2 - 409ms | ********
day  3 - 113ms | *****
day  4 -    2s | ***********
day  5 -   40s | ******************
day  6 - 12min | ************************
day  7 -  2min | ********************
day  8 - 368ms | *******
day  9 -  4min | **********************
day 10 - 767ms | *********
day 11 -    4s | *************
day 12 -  6min | ************************
day 13 - 170ms | ******
day 14 -  2min | ********************
day 15 -    3s | ************
day 16 -   10s | ***************
day 17 - 344ms | *******
day 18 - 499ms | ********
day 19 -   21s | ****************
day 20 -  9min | ************************
day 21 - 368ms | *******
day 22 -  2min | **********************
```

```
day 23 -   53s | ******************
day 24 - 929ms | *********
day 25 -    1s | **********
-- every five stars is ten times bigger
```