Contents

#+: AOC 2025 Day 8 - Playground

1 — Day 8: Playground —

1.1 Started: Thu Dec 11 15:51:26 2025

2 Package Setup

```
[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,break
;;;; Day08.lisp ;;;; 2025 AOC Day 8 solution ;;;; Common Lisp solutions
by Leo Laporte (with lots of help) ;;;; Started:  Thu Dec 11 15:51:26
2025 ;;;; Finished:  Fri 19 Dec 2025 08:55:40 PM PST
   (defpackage :aoc.2025.day08 (:use :cl :alexandria :iterate) ; no prefix
for these libraries (:local-nicknames ; short prefixes for these (:re
:cl-ppcre) ; regex (:5a :fiveam) ; test framework (:sr :serapeum) ; CL
extensions (:tr :trivia))) ; pattern matching
   (in-package :aoc.2025.day08)
   (setf 5a:*run-test-when-defined* t) ; test as we go (setf 5a:*verbose-failures*
t) ; show failing expression (sr:toggle-pretty-print-hash-table) ; automatic
pretty print for hashes (declaim (optimize (debug 3))) ; max debugging
info ;; (declaim (optimize (speed 3)) ; max speed if needed
   (defparameter *data-file* " /cl/AOC/2025/Day08/input.txt" "Downloaded
from the AoC problem set")
```

3 Part One

3.1 Notes for Part One

These numbers represent "junction boxes" with X Y Z coordinates in a 3D space. We (well the elves) want to connect the junction boxes that are closest together in a straight line (i.e. Euclidean distance). So

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}$$

Or in Lisp: (dist p q) = (sqrt (+ (expt (- xp xq) 2) (expt (- yp yq) 2) (expt (- zp zq) 2)

Once we connect the two closest junction boxes to form a circuit, we look for the next two. If any are already in a circuit the new junction points are added to that circuit. And continue until we've connected all the points up to the limit (10 for the example, 1000 for the input).

Then multiply the largest 3 circuits (by the number of junction boxes in the circuit) to get the answer. In the example it's (5x4x2) or 40.

The problem asks us to "connect together the 1000 pairs of junction boxes which are closest together. Afterward, what do you get if you multiply

together the sizes of the three largest circuits?"

3.2 Example Data

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,break
```
(defparameter *example* (list "162,817,812" "57,618,57" "906,360,560"
"592,479,940" "352,342,300" "466,668,158" "542,29,236" "431,825,988" "739,650,466"
"52,470,668" "216,146,977" "819,987,18" "117,168,530" "805,96,715" "346,949,466"
"970,615,88" "941,993,340" "862,61,35" "984,92,344" "425,690,689"))
   (defparameter *example2* (list "1,1,2" "2,14,4" "2,5,6" "3,4,11" "5,8,10"
"1,2,7" "12,4,12" "5,2,9" "3,8,11" "16,2,9" "13,4,15" "5,16,7" "18,9,10"
"100,100,100") "Another perverse example from the fertile mind of Paul
Holder Part 1 is the three circuit sizes 7, 2, 2 so the answer is 28,
and part 2 is 18, 100 so the answer is 1800")
```

As usual, the first task is to think about how to represent the data. I think I'll make a junction box struct with x y and z. Nothing in this first part seems to want CLOS so I don't need to use defclass but I'll keep that in mind for part 2. I use short conc-names (j-x j-y and j-z for conciseness). I'm also using an abbreviated make function name: make-junc.

3.3 defstruct junction

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,break
```
(defstruct (junction (:conc-name j-) ; short name is j-x, j-y.  j-z (:constructor
make-junc (x y z))) (x 0 :type integer) (y 0 :type integer) (z 0 :type
integer))
   (format t "Print representation:   a

Print representation: #S(JUNCTION :X 1 :Y 2 :Z 3)
```

3.4 parse-input()

And now let's turn the provided list of strings into a list(?). If it's a list I can represent circuits as lists within the list. Never mind. Let's make it a vector for speed. The sorting will occur later, to the distances, anyway..

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br
```
(sr:-> parse-input (list) vector) (defun parse-input (input) (iter (for
box in input) (collect (apply 'make-junc (mapcar 'parse-integer (sr:words
box))) result-type vector)))
   (format t "


#(#S(JUNCTION :X 162 :Y 817 :Z 812) #S(JUNCTION :X 57 :Y 618 :Z 57)
```

```
  #S(JUNCTION :X 906 :Y 360 :Z 560) #S(JUNCTION :X 592 :Y 479 :Z 940)
  #S(JUNCTION :X 352 :Y 342 :Z 300) #S(JUNCTION :X 466 :Y 668 :Z 158)
  #S(JUNCTION :X 542 :Y 29 :Z 236) #S(JUNCTION :X 431 :Y 825 :Z 988)
  #S(JUNCTION :X 739 :Y 650 :Z 466) #S(JUNCTION :X 52 :Y 470 :Z 668)
  #S(JUNCTION :X 216 :Y 146 :Z 977) #S(JUNCTION :X 819 :Y 987 :Z 18)
  #S(JUNCTION :X 117 :Y 168 :Z 530) #S(JUNCTION :X 805 :Y 96 :Z 715)
  #S(JUNCTION :X 346 :Y 949 :Z 466) #S(JUNCTION :X 970 :Y 615 :Z 88)
  #S(JUNCTION :X 941 :Y 993 :Z 340) #S(JUNCTION :X 862 :Y 61 :Z 35)
  #S(JUNCTION :X 984 :Y 92 :Z 344) #S(JUNCTION :X 425 :Y 690 :Z 689))
```

3.5 dist()

So the next thing I need to do is write a Euclidean distance function. Except the last part of the Euclidean formula, the square root, adds considerable calculation time. Using ISQRT - integer square root - is fast, if offering lower resolution.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(sr:-> dist (junction junction) fixnum) (defun dist (j1 j2) "given two
JUNCTIONS j1 and j2 return the euclidean distance rounded to the nearest
integer between the two" (isqrt (+ (expt (- (j-x j1) (j-x j2)) 2) (expt
(- (j-y j1) (j-y j2)) 2) (expt (- (j-z j1) (j-z j2)) 2))))
   (5a:test dist-test (5a:is (= (dist (make-junc 1 2 3) (make-junc 4 5
6)) 5)))


Running test DIST-TEST .
 Did 1 check.
    Pass: 1 (100%)
    Skip: 0 ( 0%)
    Fail: 0 ( 0%)
```

3.6 get-dists()

Now I need to think about how I'll keep track of circuits. I'll worry about the circuit connections in a bit. My biggest concern is whether I have to calculate the distances between all points. That's a million calculations for the 1000 junction boxes in the input so I'll use a design pattern I see a lot: a loop that matches the first element in the list with the succeeding elements, then he next with the rest and so on. This eliminates two cases we don't care about, the identity case (dist j1 j1) and the mirror case (dist j1 j2) (dist j2 j1).

I'll make a hash-table of the results - the key will be the distance, the value will be the two junction boxes that make that distance. This will make it easy to create a sorted list of junction box pair distances in the next step.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(sr:-> get-dists (vector) hash-table) (defun get-dists (junctions) "given
a vector of JUNCTION structs, return a hash-table of the distances between
any two pairs of junctions.  The key will be the distance, the value the
pair of junctions that are that far apart" (let* ((len (length junctions))
(distances (make-hash-table :size len)))
   (iter (for i below len) (for j1 = (aref junctions i)) (iter (for j
from (1+ i) below len) (for j2 = (aref junctions j))
   (let ((d (dist j1 j2))) (setf (gethash d distances) (append (list j1
j2) (gethash d distances))))))
   distances))
```

I better profile this to make sure it doesn't take too long. It's a lot of points!

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(let ((*trace-output* *standard-output*)) (time (get-dists (parse-input
(uiop:read-file-lines *data-file*)))))

Evaluation took:
  0.064 seconds of real time
  0.064680 seconds of total run time (0.062707 user, 0.001973 system)
  [ Run times consist of 0.012 seconds GC time, and 0.053 seconds non-GC time. ]
  101.56% CPU
  213,143,951 processor cycles
  42,127,504 bytes consed
```

Now I can sort the hash-table keys (the distances) to get the shortest distances.

This is what the 10 closest pairs of junction boxes look like...

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(let* ((junctions (parse-input *example*)) (distances (get-dists junctions))
(keys (sort (hash-table-keys distances) '<))) (iter (for i below 10) (format
t "  a ->  a" (gethash (nth i keys) distances) (nth i keys))))

(#S(JUNCTION :X 162 :Y 817 :Z 812) #S(JUNCTION :X 425 :Y 690 :Z 689)) -> 316
(#S(JUNCTION :X 162 :Y 817 :Z 812) #S(JUNCTION :X 431 :Y 825 :Z 988)) -> 321
(#S(JUNCTION :X 906 :Y 360 :Z 560) #S(JUNCTION :X 805 :Y 96 :Z 715)) -> 322
(#S(JUNCTION :X 431 :Y 825 :Z 988) #S(JUNCTION :X 425 :Y 690 :Z 689)) -> 328
```

```
(#S(JUNCTION :X 862 :Y 61 :Z 35) #S(JUNCTION :X 984 :Y 92 :Z 344)) -> 333
(#S(JUNCTION :X 52 :Y 470 :Z 668) #S(JUNCTION :X 117 :Y 168 :Z 530)) -> 338
(#S(JUNCTION :X 819 :Y 987 :Z 18) #S(JUNCTION :X 941 :Y 993 :Z 340)) -> 344
(#S(JUNCTION :X 906 :Y 360 :Z 560) #S(JUNCTION :X 739 :Y 650 :Z 466)) -> 347
(#S(JUNCTION :X 346 :Y 949 :Z 466) #S(JUNCTION :X 425 :Y 690 :Z 689)) -> 350
(#S(JUNCTION :X 906 :Y 360 :Z 560) #S(JUNCTION :X 984 :Y 92 :Z 344)) -> 352
```

3.7 sort-pairs-by-distance()

I'm going to do this a lot, so let's make a function that takes the input and returns a list of junction box pairs sorted ascending by distance. This is really an extension of the parse function. I could put this in PARSE-INPUT but I'll factor it out for clarity.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(sr:-> sort-pairs-by-distance (list) list) (defun sort-pairs-by-distance
(input) "given a list of strings representing a list of JUNCTIONS, return
a list of JUNCTION pairs sorted ascending by the distance between the
pair" (let* ((distances (get-dists (parse-input input))) ; hash of dists->juncs
(keys (sort (hash-table-keys distances) '<))) ; list of sorted dists
   (iter (for k in keys) (collect (gethash k distances)))))
```

And just to reassure me that this doesn't take too long with the full problem set

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(let ((*trace-output* *standard-output*)) (time (sort-pairs-by-distance
(uiop:read-file-lines *data-file*))))

Evaluation took:
  0.117 seconds of real time
  0.117798 seconds of total run time (0.115833 user, 0.001965 system)
  [ Run times consist of 0.057 seconds GC time, and 0.061 seconds non-GC time. ]
  100.85% CPU
  389,100,629 processor cycles
  45,609,296 bytes consed
```

OK fast enough. But does it work? Let's compare the four closest junction boxes to the example provided in AoC:

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
   (5a:test sort-pairs-by-distance-test (let ((closest (subseq (sort-pairs-by-distance
*example*) 0 4))) (5a:is (equalp (first closest) (list (make-junc 162
817 812) (make-junc 425 690 689)))) (5a:is (equalp (second closest) (list
```

```
(make-junc 162 817 812) (make-junc 431 825 988)))) (5a:is (equalp (third
closest) (list (make-junc 906 360 560) (make-junc 805 96 715)))) (5a:is
(equalp (fourth closest) (list (make-junc 431 825 988) (make-junc 425
690 689))))))


Running test SORT-PAIRS-BY-DISTANCE-TEST ....
 Did 4 checks.
    Pass: 4 (100%)
    Skip: 0 ( 0%)
    Fail: 0 ( 0%)
```

3.8 Pause to study up on trees

I think I have all the pieces I need. Now I can write the function that builds the paths according to the rules in part 1. But wait - I think I'm building a "minimum spanning tree" that is, the shortest path from start to end.

OK I didn't know that until I got some hints from Reddit but I'm glad I did. I went down the tree rabbit hole (to mix metaphors), which is something I've been wanting to do for some time. There are two things that Lispers agree are important for solving a wide variety of problems: **recursion** which I've got a pretty good handle on, and **tree** data structures and the wide variety of ways they can be used. **This** is why I do AoC. I'm not trying to be the fastest, I know I'm a duffer programmer. I'm in it for the learnings (and it's a really fun way to learn). I didn't study computer science in school. These problems provide a fun way for me to learn algorithms, and, incidentally, Common Lisp, Emacs, org-mode, and Literate Programming. So...

Excuse me while I read up on these in Steve Skiena's Algorithm Design Manual and Domkin's Programming Algorithms in Lisp - which I've owned for some time. And I watched some videos on YouTube, in particular MIT's Introduction to Algorithms course, CS 6.006 videos 6 and 7. I'll be back in a minute.

3.9 Kruskal's Algorithm to the rescue

Let me paraphrase the instructions. This helps me understand what I need to do. For part 1 I need to start with the two junction boxes that are closest together. Put them in a circuit. Check the next closest pair. If the pair shares a junction box with the first circuit, add it to the circuit, otherwise create a new circuit. Take the next closest pair. Check to see if any of its

points fit into an existing circuit without creating a loop. (It's a loop if *both* points in the junction box pair already exist in the circuit.) Only add a pair to an existing circuit if exactly *one* of the two points exists in the circuit. Proceed until all the junction boxes are in a circuit, even if it's only a circuit of one. Multiply the sizes of the three largest circuits together to get the answer.

From now on I'll call a junction box a **node** and the distance between two nodes an **edge**. The circuit I'll call a **path**. So this is a tree. And a path is really just a **set** of nodes, so I can add a node to a path using UNION to avoid multiple points. And I can check if a node is already in a circuit using MEMBER.

This is the fundamental process for the whole problem. It turns out that this is an example of Kruskal's Algorithm (or Prim's algorithm which is similar). It delivers the "minimum spanning tree of an undirected edge weight graph." In other words, the shortest string of lights that can connect all the junction boxes.

In this case the weights are the distances between points, and the minimum tree is the shortest length of Christmas lights. So this describes the problem exactly.

We already have the first two steps done (from Wikipedia):

3.9.1 Create a forest (a set of trees) initially consisting of a separate single-vertex tree for each vertex in the input graph. (DONE)

3.9.2 Sort the graph edges by weight. (DONE)

Now...

3.9.3 Loop through the edges of the graph, in ascending sorted order by their weight.

1. For each edge:
   (a) Test whether adding the edge to the current forest would create a cycle.
   (b) If not, add the edge to the forest, combining two trees into a single tree.

   At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest

has a single component and forms a minimum spanning tree. Ta da!

So I need a function that checks whether adding the next pair to the current forest would create a cycle and then, if there are shared points in nodes, returns a new joined circuit.

3.10 A Fresh Start with Union-Find

OK here's where things get interesting. My first attempt at this problem (the ADD-PAIR-TO-PATH? and BUILD-PATHS functions I wrote earlier, now abandoned) tried to track circuits using lists and set operations. It worked... sort of. But it had a fatal flaw: when I tried to merge two circuits together, I wasn't doing it correctly. The order of connections mattered, and I was losing track of which nodes belonged to which circuit.

After banging my head against this for a while, I went back to the algorithm textbooks. It turns out there's a classic data structure designed *exactly* for this problem: **Union-Find** (also called "Disjoint Set Union" or DSU). It's one of those beautiful computer science inventions that makes a hard problem easy.

The idea is simple: imagine each junction box starts in its own little circuit (a "component"). When we connect two junction boxes with a wire, their circuits merge. Union-Find lets us do two things very fast:

- **FIND**: "Which circuit does this junction box belong to?" (Find the "root" or representative of its component)
- **UNION**: "Merge these two circuits into one." (Connect two components)

The magic is in two optimizations:

1. **Path compression**: When we look up which circuit a node belongs to, we update its pointer to go directly to the root. This flattens the tree over time.
2. **Union by rank**: When merging circuits, we attach the shorter tree under the taller one, keeping things balanced.

With both optimizations, these operations run in nearly O(1) time - actually O((n)) where is the inverse Ackermann function, which is essentially constant for any practical input size. Pretty neat!

I'm using hash tables here instead of arrays because my junction boxes are structs, not simple integers. The `:test 'equalp` ensures struct equality works.

3.10.1 Union-Find Implementation

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,break

```
;; Union-Find data structure using hash tables (for struct keys) (defun
make-union-find () "Create a new union-find structure.  Returns (parent
.  rank) hash tables." (cons (make-hash-table :test 'equalp) ; parent
(make-hash-table :test 'equalp))) ; rank
   (defun uf-find (uf node) "Find the root of NODE with path compression."
(let ((parent (car uf))) ;; Initialize node if not seen (unless (gethash
node parent) (setf (gethash node parent) node)) ;; Find root with path
compression (if (equalp (gethash node parent) node) node (setf (gethash
node parent) (uf-find uf (gethash node parent))))))
   (defun uf-union (uf node1 node2) "Union the sets containing NODE1 and
NODE2.  Returns T if they were separate." (let* ((parent (car uf)) (rank
(cdr uf)) (root1 (uf-find uf node1)) (root2 (uf-find uf node2))) (unless
(gethash root1 rank) (setf (gethash root1 rank) 0)) (unless (gethash root2
rank) (setf (gethash root2 rank) 0)) (cond ((equalp root1 root2) nil)
; already in same set ((< (gethash root1 rank) (gethash root2 rank)) (setf
(gethash root1 parent) root2) t) ((> (gethash root1 rank) (gethash root2
rank)) (setf (gethash root2 parent) root1) t) (t (setf (gethash root2
parent) root1) (incf (gethash root1 rank)) t))))
   (defun uf-component-sizes (uf nodes) "Return a list of component sizes
for all NODES in union-find UF." (let ((size-map (make-hash-table :test
'equalp))) (iter (for node in nodes) (let ((root (uf-find uf node))) (incf
(gethash root size-map 0)))) (hash-table-values size-map)))
   (5a:test union-find-test (let ((uf (make-union-find))) ;; Initially
separate (5a:is (uf-union uf 'a 'b)) ; returns T, now connected (5a:is-false
(uf-union uf 'a 'b)) ; returns NIL, already connected (5a:is (uf-union
uf 'c 'd)) (5a:is (uf-union uf 'b 'c)) ; connects a,b with c,d (5a:is-false
(uf-union uf 'a 'd)) ; all in same component now (5a:is (equal (list 4)
(uf-component-sizes uf '(a b c d))))))


Running test UNION-FIND-TEST ......
 Did 6 checks.
    Pass: 6 (100%)
    Skip: 0 ( 0%)
    Fail: 0 ( 0%)
```

3.10.2 get-all-edges()

Now I need to rethink how I'm generating the edges. My earlier GET-DISTS function stored edges in a hash table keyed by distance, but that's awkward for Kruskal's algorithm which wants a simple sorted list of edges.

So I'm rewriting this to return a list of `(distance j1 j2)` tuples, sorted by distance. This is the format Kruskal wants: "give me all the edges from shortest to longest, and I'll process them in order."

The nested loop pattern is the same as before - for each junction box, I pair it with every junction box that comes *after* it in the vector. This avoids both the identity case (distance from a point to itself) and duplicates (we don't need both A→B and B→A).

One nice thing about having the distance as the first element of each tuple: I can sort the whole list with `#'< :key #'first` and it just works.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br
```
(sr:-> get-all-edges (vector) list) (defun get-all-edges (junctions) "Return
a list of (distance j1 j2) for all pairs, sorted by distance." (let* ((len
(length junctions)) (edges nil)) (iter (for i below len) (for j1 = (aref
junctions i)) (iter (for j from (1+ i) below len) (for j2 = (aref junctions
j)) (push (list (dist j1 j2) j1 j2) edges))) (sort edges '< :key 'first)))
```

3.10.3 kruskal()

Now for the main event! This is Kruskal's algorithm, and it's surprisingly simple once you have Union-Find doing the heavy lifting.

The algorithm works like this:

1. Start with each junction box in its own component (this happens automatically when we first call UF-FIND on a node)
2. Process edges in order from shortest to longest
3. For each edge, try to union the two junction boxes
4. If they were already in the same component, UF-UNION returns NIL (we'd create a cycle, so skip this edge)
5. If they were in different components, UF-UNION returns T (we just merged them!)
6. Stop after processing MAX-EDGES edges

For Part 1, we process exactly 1000 edges (or 10 for the example), then count how many junction boxes ended up in each component. The answer is the product of the three largest component sizes.

I'm also tracking the last edge processed, which will be useful for Part 2. And I keep a list of all nodes we've seen so I can calculate component sizes at the end.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br
```
(defun kruskal (edges max-edges) "Process up to MAX-EDGES edges using
Kruskal's algorithm.  Returns (component-sizes last-j1 last-j2)." (let
((uf (make-union-find)) (nodes nil) (last-j1 nil) (last-j2 nil) (edges-processed
0)) (iter (for (dist j1 j2) in edges) (while (< edges-processed max-edges))
;; Track all nodes we've seen (pushnew j1 nodes :test 'equalp) (pushnew
j2 nodes :test 'equalp) ;; Union the nodes (uf-union uf j1 j2) (setf last-j1
j1 last-j2 j2) (incf edges-processed)) (values (uf-component-sizes uf
nodes) last-j1 last-j2)))
```

3.10.4 Solution

With all the pieces in place, DAY08-1 becomes almost trivial. Parse the input, generate all edges sorted by distance, run Kruskal for the specified number of edges, sort the resulting component sizes in descending order, take the top three, and multiply them together.

I'm using Serapeum's threading macro `~>` here because I find it reads nicely: "take sizes, sort descending, take first 3, multiply them." The underscore `_` is where the result of each step gets plugged into the next.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br
```
(defun day08-1 (input max-edges) "Given a list of junction boxes, INPUT,
return the result of multiplying together the three largest possible circuits
that can be built by connecting the MAX-EDGES closest pairs.  Uses Kruskal's
algorithm with Union-Find." (let* ((junctions (parse-input input)) (edges
(get-all-edges junctions)) (sizes (kruskal edges max-edges))) (sr: > sizes
(sort '>)(subseq_0(min3(length_)))(apply'*_))))
```

3.10.5 Test

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,break
```
(5a:test day08-1-test (5a:is (= 40 (day08-1 *example* 10))))


Running test DAY08-1-TEST .
 Did 1 check.
```

```
Pass: 1 (100%)
Skip: 0 ( 0%)
Fail: 0 ( 0%)
```

3.10.6 Part one finished: Sat Dec 13 16:00:55 2025

4 Part Two

4.1 Notes

Part 2 threw me for a loop at first. The problem says: find the pair of junction boxes that, when connected, completes a single circuit containing *all* the junction boxes. Then multiply their X coordinates.

My first instinct was wrong. I thought "just keep connecting until everything is connected and remember the last pair." But my old BUILD-PATHS function was merging circuits in a haphazard way - it wasn't properly tracking which edge actually *caused* the final merge.

Here's the key insight that made it click: we're building a **Minimum Spanning Tree**. To connect n nodes into a single tree, we need exactly n-1 edges. Each edge we add (that actually merges two components) reduces our component count by one. We start with n components (each junction box alone) and we need to get down to 1 component (everything connected).

So the algorithm is:

1. Start with n components (one per junction box)
2. Process edges shortest-to-longest (same as Part 1)
3. Each time UF-UNION returns T, we successfully merged two components - decrement our component counter and remember this edge
4. Stop when components = 1 (everything is connected)
5. The last edge that returned T is our answer!

The beauty of Union-Find is that UF-UNION tells us *exactly* what we need to know: did this edge actually connect two separate components (T) or were they already connected (NIL)?

4.2 Solution

The code is almost the same as KRUSKAL, but instead of processing a fixed number of edges, we process until there's only one component left. We track

the component count explicitly and decrement it each time we successfully merge.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(defun day08-2 (input) "Find the pair that connects all junction boxes
into a single circuit.  Return the product of the X coordinates of that
pair." (let* ((junctions (parse-input input)) (edges (get-all-edges junctions))
(uf (make-union-find)) (n (length junctions)) (components n) ; start with
n separate components (last-j1 nil) (last-j2 nil))
   ;; Initialize all nodes in union-find (iter (for j in-vector junctions)
(uf-find uf j))
   ;; Process edges until we have 1 component (all connected) (iter (for
(dist j1 j2) in edges) (while (> components 1)) (when (uf-union uf j1
j2) ;; This edge merged two separate components (decf components) (setf
last-j1 j1 last-j2 j2)))
   (* (j-x last-j1) (j-x last-j2))))
```

4.3 Test

Let me verify this works with the example. For the 20 junction boxes in `*example*`, we need 19 edges to connect them all. The last edge that merges two components into one connects the junction boxes at (216, 146, 977) and (117, 168, 530). Their X coordinates are 216 and 117, so the answer is 216 × 117 =

1.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
(5a:test day08-2-test (5a:is (= 25272 (day08-2 *example*))))


Running test DAY08-2-TEST .
 Did 1 check.
    Pass: 1 (100%)
    Skip: 0 ( 0%)
    Fail: 0 ( 0%)
```

4.4 Part two finished: Fri 19 Dec 2025 08:55:40 PM PST

4.5 Reflections

This was a great learning experience! I went into this problem thinking "I'll just track circuits with lists" and came out understanding why Union-Find is

such a fundamental data structure. It's now in my toolbox for any problem involving connected components, cycle detection, or graph connectivity.

The key lessons:

- When you need to track "which group does this thing belong to?" and merge groups, reach for Union-Find
- Kruskal's algorithm + Union-Find is a powerful combination for MST problems
- Sometimes the "simple" approach (lists and set operations) isn't enough - you need the right data structure
- Reading the algorithm textbooks pays off!

5 Run Solutions and Timings

Timings with SBCL on a Thinkpad X1 Carbon with Intel Core Ultra 7 and 32GB RAM running pop-os 24.04. Literate document created in Emacs 29.3 with org-babel. Solution created with help from Claude Code running Opus 4.5.

[basicstyle=,keywordstyle=blue,commentstyle=gray,stringstyle=green!50!black,frame=single,br

```
;; now solve the puzzle!
   (let ((*trace-output* *standard-output*)) (time (format t "The answer
to AOC 2025 Day 8 Part 1 is  a" (day08-1 (uiop:read-file-lines *data-file*)
1000)))
   (time (format t "The answer to AOC 2025 Day 8 Part 2 is  a" (day08-2
(uiop:read-file-lines *data-file*)))))

The answer to AOC 2025 Day 8 Part 1 is 105952
Evaluation took:
  0.201 seconds of real time
  0.202040 seconds of total run time (0.179928 user, 0.022112 system)
  [ Run times consist of 0.043 seconds GC time, and 0.160 seconds non-GC time. ]
  100.50% CPU
  667,296,888 processor cycles
  32,654,560 bytes consed

The answer to AOC 2025 Day 8 Part 2 is 975931446
Evaluation took:
  0.207 seconds of real time
  0.208359 seconds of total run time (0.188300 user, 0.020059 system)
```

```
[ Run times consist of 0.066 seconds GC time, and 0.143 seconds non-GC time. ]
100.48% CPU
686,194,986 processor cycles
32,614,704 bytes consed
```