Advent of Code

Athanasios Tsarapatsanis, Murtaza Haidari, Gediminas Marcinkevičius

January 7, 2026

1 Problem description

1.1 Overview

An underground playground contains junction boxes that the Elves wish to connect using strings of decorative lights. Each junction box occupies a specific position in three-dimensional space, defined by X, Y, and Z coordinates. The electrical system operates under a fundamental principle: when two junction boxes are connected by a string of lights, electrical current can flow between them, effectively creating a pathway (circuit) within the broader network.

1.2 The Connection Process

The Elves employ a methodical approach to minimize the total length of light strings required. They identify pairs of junction boxes based on straight-line (Euclidean) distance, connecting the two closest boxes first, then the next closest pair, and so on. This algorithm continues until a specified number of connections are established.

The standard Euclidean distance formula is used to determine the distance between any two junction boxes. For two boxes at positions (x_1, y_1, z_1) and (x_2, y_2, z_2), the distance is calculated as: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$

1.3 Circuit Formation

As connections are made between junction boxes, electrical circuits begin to form. A circuit represents a group of connected junction boxes where electricity can flow from any box to any other box within that group. Junction boxes that remain unconnected to others constitute single-box circuits.

When the first pair of closest boxes is connected, this creates a simple circuit containing two boxes. As additional connections are made, one of three outcomes occurs:

1. Two previously unconnected boxes form a new two-box circuit

2. A connection links an isolated box to an existing circuit, expanding that circuit by one box

3. A connection bridges two separate circuits, merging them into a single larger circuit

1.3.1 Short Example in 2D Space

Connection	Distance	Order of Connections
Junction 0 ↔ Junction 1	8.5	1st connection
Junction 2 ↔ Junction 3	12.3	2nd connection
Junction 0 ↔ Junction 4	15.7	3rd connection
Junction 1 ↔ Junction 2	18.9	4th connection (merges circuits)

Table 1: Distances between junction boxes.

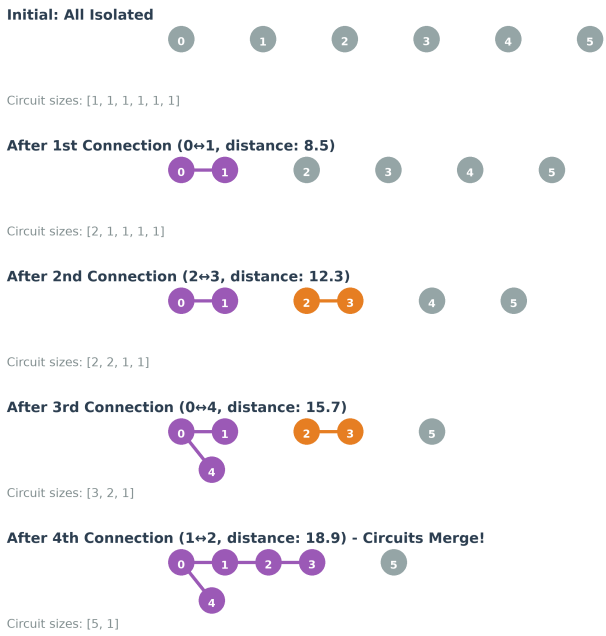

Figure 1: 2D example, where different colours represent different circuits.

1.4 The Challenge

1.4.1 Part 1

Completing one thousand pairwise connections between the closest available junction boxes. After which, the network will consist of multiple distinct circuits of varying sizes. The task then requires identifying the three largest circuits by the number of junction boxes they contain. Finally computing the product of the sizes of these three circuits.

1.4.2 Part 2

Continuing to connect junction boxes beyond the initial one thousand connections until all junction boxes form a single unified circuit. The objective is to identify which specific connection completes this full network integration. That is, which pair of junction boxes, when connected, causes all remaining separate circuits to merge into one. Once this final connection is identified, multiply together the X coordinates of these two junction boxes to produce the answer.

2 Naive approach

2.1 Part I

At the beginning of the investigation, a naive solution approach was implemented to create a baseline for problem solving. This approach was based on native Python without the use of optimized algorithms or external libraries (such as NumPy). However, after implementation, analysis showed that this approach is unsuitable for larger data sets due to serious inefficiencies. The identified weaknesses and the resulting time and memory complexities are outlined below.

2.1.1 Calculating and storing distances

```
#Calculating Distances:
all_pairs = []
FOR i FROM 0 TO N:
    FOR j FROM i+1 TO N:
        distance = SQRT((x1-x2)^2 + ...)

#Storing Distances:
all_pairs.append((distance, i, j))
```

Since the naive algorithm calculates the distance between every possible pair of points, the time complexity is $O(N^2)$. This leads to a quadratic increase in runtime as the number of points (N) increases.

In terms of memory usage, Python's native list data structure was used. Since each calculated pair is stored, the memory requirement also grows with $O(N2)$. This is particularly inefficient because Python lists generate additional overhead (object headers, pointers) for each element, which quickly exceeds the capacity limits of the working memory when N is large.

2.1.2 Sort by distance

After calculating all pairwise distances, the naive approach involves globally sorting the entire list of results in order to then extract the k shortest connections (in this case, k=1000).

```
all_pairs.sort()
top_1000 = all_pairs[:1000]
```

This step represents a significant performance bottleneck. Since the number of pairs to be sorted grows quadratically with the number of points ($M \approx \frac{N^2}{2}$), a complete sort results in a runtime complexity of $\mathcal{O}(M \cdot \log M)$. Substituting M with N^2 results in a complexity of $\mathcal{O}(N^2 \cdot \log(N^2))$.

This is algorithmically inefficient because the problem does not require the completely sorted order of all edges, but only the top k elements. Global sorting means that a large part of the computing power is used to sort irrelevant data (elements k+1 to M).

2.1.3 Finding graphs

After selecting the k shortest edges, the next algorithmic step is to manage the affiliation of the individual points to their respective connected components (clusters). The goal is to identify in which subgraph the nodes u and v of a given connection are located.

The naive approach implements this management using a list of sets, whereby the search for group membership is mapped as follows:

```
group_list = [{0}, {1}, ..., {N}]
FOR EACH connection (u, v):
    FOR EACH group IN group_list:
        IF u IS IN group: save_index_u
        IF v IS IN group: save_index_v
```

Since this iteration must be performed for each of the k selected connections, this results in a runtime complexity of $O(k \cdot N)$. Here, k describes the number of connections to be processed and N describes the number of independent graphs or points. In contrast to efficient tree structures, this linear dependency leads to a significant slowdown in group management as the number of points increases.

2.2 Part II

Similar to the approach taken in the first part of the task, a naive approach was initially chosen to solve the second part. The primary goal was to ensure algorithmic correctness and generate a valid solution to the puzzle ("Golden Star"). After successful validation, a detailed analysis of the runtime and memory behavior was performed, which revealed significant deficiencies and served as the basis for the subsequent optimization.

2.2.1 Generation and Materialization of Edges

The initial approach was based on the assumption of a complete graph in which each point can in principle be connected to every other point.

```
Generate_All_Edges(Points P)
    N = Length(P)
    EdgeList = New List()

    FOR i FROM 0 TO N-1:
        FOR j FROM i+1 TO N-1:
            dx = P[i].x - P[j].x
            dy = P[i].y - P[j].y
            dz = P[i].z - P[j].z

            Distance = SQRT(dx*dx + dy*dy + dz*dz)

            APPEND (Distance, i, j) TO EdgeList
```

A major disadvantage is that all calculated connections are stored explicitly in the working memory (RAM). Since in a complete graph the number of edges M is determined by the formula $M = \frac{N(N-1)}{2}$, the memory requirement grows quadratically with the number of nodes ($\mathcal{O}(N^2)$). With large input quantities, this leads to massive RAM usage, which severely limits the scalability of the algorithm.

In addition to the memory problem, the calculation of edge lengths is inefficient. The square root function (`SQRT`) used in the pseudo-code to determine the exact Euclidean distance is a computationally intensive operation. However, for the pure sorting or comparison of edge lengths, the calculation of the root is not mathematically necessary.

2.3 Global Sorting of the Edge List

```
Global_Sort(EdgeList)
    SORT EdgeList ASCENDING BY Distance
```

After generating all possible connections, the second step is to globally sort the edge list. This process is necessary because the underlying algorithm (Kruskal) follows the greedy principle and must process edges in ascending order of their weight (length).

However, this reveals a serious inefficiency of the naive approach: While the Minimum Spanning Tree (MST) sought for N nodes consists of exactly $N-1$ edges, this approach forces the complete sorting of all $M \approx \frac{N^2}{2}$ generated edges.

This means that the algorithm uses most of its computing power to sort edges that are irrelevant to the final solution because they either close cycles or are simply too long.

3 Solution design

3.1 Part I

Let $p_0, ..., p_{n1} \in R^3$ denote the input points. Our approach constructs a graph whose vertices correspond to points, and whose edges connect pairs of points with small pairwise distance. We then compute the connected components of this graph and use the sizes of the largest components to form the final result.

Pairwise distance computation (condensed form).

We compute all pairwise (squared) Euclidean distances

$$d_{ij} = p_i p_j 2^2,\ 0i < j < n.$$

Using `scipy.spatial.distance.pdist` with metric `sqeuclidean`, these distances are stored in a one-dimensional condensed array of length $M = \binom{n}{2} = \frac{n(n1)}{2}$. The generated vector would look like this:

$$\begin{array}{rc} k=0 & \left(\begin{array}{c} ||p_0 - p_1||_2^2 \\ ||p_0 - p_2||_2^2 \\ \cdots \\ ||p_0 - p_{n-1}||_2^2 \\ ||p_1 - p_2||_2^2 \\ \cdots \end{array} \right) \\ k=1 & \\ \cdots & \\ k=(n-2) & \\ k=(n-1) & \\ \cdots & \end{array}$$

Conceptually, this corresponds to storing the upper triangular part (excluding the diagonal) of an $n \times n$ distance matrix:

$$\begin{pmatrix} ||p_0 - p_1||_2^2 & ||p_0 - p_2||_2^2 & \cdots & ||p_0 - p_{n-1}||_2^2 \\ & ||p_1 - p_2||_2^2 & \cdots & ||p_1 - p_{n-1}||_2^2 \\ & & \ddots & \vdots \\ & & & ||p_{n-2} - p_{n-1}||_2^2 \\ & & & \end{pmatrix}$$

symmetric duplicates $d_{ij} = d_{ji}$and diagonal entries $d_{ii} = 0$ are omitted to reduce memory usage.

Selecting the k smallest distances.

Rather than sorting all M distances, we select the indices of the k smallest entries using `numpy.argpartition`, which performs partial selection without fully sorting the array. This yields a set of condensed indices identifying the shortest candidate edges.

Mapping condensed indices to edges.

Because `pdist` stores distances in condensed form, each selected index must be mapped back to its endpoint pair (i, j). We construct index arrays $(I, J) =$ `np.triu\_indices`$(n, 1)$, which enumerate all upper-triangular pairs in the same order used by `pdist`. For each selected condensed index t, the corresponding edge is $(I[t], J[t])$.

Graph construction and connected components.

From the selected edges we build an undirected, unweighted graph and represent it as a sparse Boolean adjacency matrix (COO format). Finally, we compute connected components using `scipy.sparse.csgraph.connected_components`. We extract the component sizes, select the three largest components, and compute their product as the final output.

3.2 Part II

Although the underlying graph is complete, explicitly constructing all edges or a full distance matrix would require $O(n^2)$ memory. To avoid this, we compute a **Minimum Spanning Tree (MST)** directly using Prim's algorithm. Prim's algorithm requires only:

1. a set T of vertices already in the tree,
2. for each vertex vT, the currently best known connection weight $min_{uT}w(u, v)$, where $w(u, v) = ||p_u - p_v||_2^2$

Thus, edges are generated **implicitly**: whenever a vertex u is added to the tree, the algorithm computes distances from u to all remaining vertices vT and updates their best connection costs. This achieves the effect of operating on the complete graph while keeping memory usage $O(n)$.

3.2.1 Output derived from the MST

During MST construction, we additionally track the heaviest edge inserted into the MST. After completion, we take the endpoints (u^*, v^*) of this maximum-weight MST edge and return the product of their x-coordinates, $x_{u^*}x_{v^*}$ as required by the task.

3.3 Libraries

For Part I we rely on `NumPy` and `SciPy` to implement the computationally intensive steps efficiently. `SciPy`'s `pdist` computes all pairwise distances in optimized compiled code and returns them in a memory-efficient condensed format, while `NumPy`'s vectorized operations (e.g., `argpartition`) enable fast top-k selection without sorting the full array. For graph processing we use `scipy.sparse` and `scipy.sparse.csgraph`, where sparse matrices (`coo_matrix`) allow compact storage of the adjacency structure and `connected_components` provides an optimized implementation for extracting connected components. Additionally, Python's built-in `re` module offers a lightweight and robust way to parse the input coordinates.

3.4 Our Git Workflow

We first set up a central repository on GitHub to handle all our code and keep everything in one place.

3.4.1 Kanban

To organize our work, we used a Kanban board with four columns:

1. *Backlog*: challenges we hadn't started yet.
2. *In Progress*: what we were currently working on.
3. *In Review/Optimization*: solutions awaiting feedback.
4. *Done*: completed challenges.

Each day's challenge was added as a card on the Kanban board, which automatically created a GitHub issue for tracking. When someone was ready to tackle a challenge, they'd move the card to "In Progress" and get assigned to that issue.

3.4.2 Coding Process

For the simpler (early) challenges that could be handled solo:

1. *Create a branch*: the assigned person would create a new branch off main for their solution.

2. *Implement and push*: they'd work on the challenge in that branch and push it to GitHub.

3. *Open a pull request*: once done, they'd create a pull request with "closes #[issue number]" in the description to link it to the issue.

4. *Code review*: the rest of the team would be assigned as reviewers to look over the code and leave comments.

5. *Merge*: after everyone approved, the pull request got merged into main, automatically closing the issue and moving the card to "Done".

This workflow enabled us to catch mistakes through peer review and keep a clear history of who did what and when.

4 Time and Space Complexity

Time and space complexity (Part I). Let n be the number of input points and let $M = \binom{n}{2} = \frac{n(n-1)}{2}$ denote the number of pairwise distances in the complete graph. Computing all squared Euclidean distances using `pdist` takes $O(M)$ time, i.e. $O(n^2)$. Selecting the k smallest distances via `numpy.argpartition` runs in expected $O(M)$ time and avoids the $O(M \log M)$ cost of fully sorting all distances. Mapping condensed indices back to endpoint pairs and constructing the sparse adjacency matrix requires $O(k)$ time. Finally, computing connected components on the resulting sparse graph takes $O(n + k)$ time. Overall, the runtime is dominated by the pairwise distance computation and is therefore $O(n^2)$. In terms of space, the condensed distance array requires $O(M) = O(n^2)$ memory, while the sparse graph representation and auxiliary arrays require $O(n+k)$; hence the overall space complexity is $O(n^2)$, dominated by storing all pairwise distances.

Time and space complexity (Part II). Part II conceptually operates on the complete weighted graph but avoids explicit storage of all $\binom{n}{2}$ edges by constructing a Minimum Spanning Tree using Prim's algorithm. The implementation maintains the arrays `in_tree`, `best_dist`, and `parent`, each of length n, and iteratively selects the next vertex to add by scanning all vertices ($O(n)$) and then updates connection costs by computing distances to all remaining vertices ($O(n)$). Repeating this procedure for n iterations yields a total time complexity of $O(n^2)$. Memory usage is $O(n)$, since the algorithm stores only per-vertex bookkeeping information and computes squared distances on-the-fly, rather than materializing the full distance matrix.