Analysis: Word Search

Word Search: Analysis

Small dataset

One surprisingly viable approach for the Small dataset is to create random grids of random sizes and count the numbers of `I/O`s in them until we happen to find a grid with the desired number. Even if we only try square grids with a random size and random contents, this is easily fast enough to pass. Other optimizations are possible — we can vary the frequencies of `I`, `/`, and `O`, or try to edit grids that are close — but not necessary.

Another possibility is to directly construct a grid with the desired number of `I/O`s. Better yet, we can construct one grid with at least 287 `I/O`s, and then use it in every test case, eliminating `I/O`s individually until we have the desired number. To ensure that `I/O`s do not interfere with each other, let's space them out in a field of `O`s like this:

```
I/OOI/OOI/OO...
OOOOOOOOOOO...
I/OOI/OOI/OO...
OOOOOOOOOOO...
I/OOI/OOI/OO...
...
```

If we extend this pattern to fill a 50 by 50 grid, we will have 25 rows with 12 `I/O`s each; 25 × 12 = 300, which is more than enough. We can eliminate an individual `I/O` by changing its `/` into another `O`. With this grid setup, that change cannot possibly affect other existing `I/O`s or create new ones. Other similar strategies are possible.

Large dataset

The particular construction strategy above cannot produce enough `I/O`s when each of our grid dimensions is capped at 15. Nor will random generation help; we would be lucky to get a random 15 x 15 grid for **N** = 80, let alone 287. We need to pack as many `I/O`s into the grid as possible. Let's start with a single row; we can fit many `I/O`s in, overlapping forward ones with backward ones:

```
I/O/I/O/I...
```

If we stack these rows on top of each other, we will also form many diagonal `I/O`s:

```
I/O/I/O/I...
I/O/I/O/I...
I/O/I/O/I...
...
```

How many `I/O`s are in the 15 by 15 version of this grid? Let's count by looking only at the `/`es. Any `/` in the top or bottom row is part of exactly one `I/O`, running horizontally. Any other `I/O` is part of exactly three `I/O`s: one horizontal and two diagonal. There are 7 `/`es in each row; the 14 in the top and bottom row contribute 14 `I/O`s, and the 13 × 7 = 91 in the other rows contribute

91 × 3 = 273. That's a total of 287, which is conveniently exactly the maximum number of `I/O`s we could be asked to produce. Note that we do not need to prove that this arrangement packs in as many `I/O`s as possible; we only had to find a way to fit at least 287 in.

Now, in each test case, we only need to whittle the number of `I/O`s down from 287 to **N**. As we do this, we must be careful not to create any new `I/O`s or destroy more `I/O`s than we want to. One safe strategy is to change `/`es into `O`s, as in the Small strategy above. Changing a `/` in the top or bottom row eliminates one `I/O`, and changing any other `/` eliminates three. We can reach any value of **N** by first removing as many "threes" as possible (without going below **N**) and then removing as many "ones" as possible (without going below **N**). For example, for **N** = 280, we remove two "threes" and one "one". For **N** = 4, we remove everything except for four of the "ones". Since there are only 288 possible test cases, we can even precompute answers to all of them before downloading the dataset (but this is not necessary).