advent_of_code_day 09

December 15, 2021

1 Advent of Code

[ ]:
```
# set up the environment
import numpy as np
```

1.0.1 Day 9: Smoke Basin

These caves seem to be lava tubes. Parts are even still volcanically active; small hydrothermal vents release smoke into the caves that slowly settles like rain.

If you can model how the smoke flows through the caves, you might be able to avoid it and be that much safer. The submarine generates a heightmap of the floor of the nearby caves for you (your puzzle input).

Smoke flows to the lowest point of the area it's in. For example, consider the following heightmap:

2199943210
3987894921
9856789892
8767896789
9899965678

Each number corresponds to the height of a particular location, where 9 is the highest and 0 is the lowest a location can be.

Your first goal is to find the **low points** - the locations that are lower than any of its adjacent locations. Most locations have four adjacent locations (up, down, left, and right); locations on the edge or corner of the map have three or two adjacent locations, respectively. (Diagonal locations do not count as adjacent.)

In the above example, there are *four low points*, all highlighted: two are in the first row (a 1 and a 0), one is in the third row (a 5), and one is in the bottom row (also a 5). All other locations on the heightmap have some lower adjacent location, and so are not low points.

The **risk level** of a low point is **1 plus its height**. In the above example, the risk levels of the low points are 2, 1, 6, and 6. The sum of the risk levels of all low points in the heightmap is therefore **15.**

Find all of the low points on your heightmap. **What is the sum of the risk levels of all low points on your heightmap?**

[ ]:
```
# Set up the data arrays for both the test data and the puzzle data
cave_data_raw = np.loadtxt('data/cave_heighmap.dat', dtype=str)
cave_data = np.empty((len(cave_data_raw), len(cave_data_raw[0])), dtype=np.int0)
for x, row in enumerate(cave_data_raw):
    for y, height in enumerate(cave_data_raw[x]):
        cave_data[x,y] = height
test_data = np.array([[2, 1, 9, 9, 9, 4, 3, 2, 1, 0],
                      [3, 9, 8, 7, 8, 9, 4, 9, 2, 1],
                      [9, 8, 5, 6, 7, 8, 9, 8, 9, 2],
                      [8, 7, 6, 7, 8, 9, 6, 7, 8, 9],
                      [9, 8, 9, 9, 9, 6, 5, 6, 7, 8]])
```

[ ]:
```
def array_low_points(a, fill_val: int) -> np.ndarray:
    """
    Returns an array where non low points are replaced by the specified value
    a: the array to check
    fill_value: replacement value for non-points
    """
    low_points = np.full_like(a, fill_val)
    for x, line in enumerate(a):
        if line[0] < line[1]:
            low_points[x, 0] = line[0]
        for y in range(1, len(line)-1):
            if line[y] < line[y-1] and line[y] < line[y+1]:
                low_points[x,y] = line[y]
        if line[-1] < line[-2]:
            low_points[x,-1] = line[-1]
    return low_points
```

[ ]:
```
# find the risk factor
data = cave_data.copy()

line_low_points = array_low_points(data, 10) # array of row low points with 10␣
 ↪as non low points
col_low_points = array_low_points(data.T, 11).T # array of column low points␣
 ↪with 11 as the fill value
low_point_matrix = col_low_points == line_low_points # maxtrix where the row␣
 ↪and column low points coincide.\

low_points = data[low_point_matrix]


risk = np.sum(low_points) + np.size(low_points)
print(f'Risk level of low points: {risk}')
```

```
Risk level of low points: 524
```

1.0.2 Part Two

Next, you need to find the largest basins so you know what areas are most important to avoid.

A **basin** is all locations that eventually flow downward to a single low point. Therefore, every low point has a basin, although some basins are very small. Locations of height 9 do not count as being in any basin, and all other locations will always be part of exactly one basin.

The **size** of a basin is the number of locations within the basin, including the low point. The example above has four basins.

The top-left basin, size 3:

2199943210
3987894921
9856789892
8767896789
9899965678

The top-right basin, size 9:

21999**43210**
3987894**921**
98567898**9**2
8767896789
9899965678

The middle basin, size 14:

2199943210
398**78**94921
9**85678**9892
8767896789
9899965678

The bottom-right basin, size 9:

2199943210
3987894921
9856789**8**92
8767896**78**9
98999**65678**

Find the three largest basins and multiply their sizes together. In the above example, this is 9 * 14 * 9 = 1134.

What do you get if you multiply together the sizes of the three largest basins?

1.0.3 Strategy

As the basins are all surrounded by high points we can traverse across each line marking each non high point with a number. Every time we cross a high point we use a new unique allocation number. At the start of a new line we also increase the allocation number by one. On all but the first line we check each points preceding row value - if it is not a high point and a different allocated number

we change all the current cells marked with the current allocation number to the preceding rows value and continue with the preceding value.

After all the cells have been traversed we count the number of each remaining allocated values and calculate the producet of the three highest counts.

[ ]:
```
basin_data = cave_data.copy()
basin_map = np.zeros_like(basin_data)

next_allocation_number = 0
current_allocation_number = next_allocation_number
for x, row in enumerate(basin_data):
    next_allocation_number += 1
    current_allocation_number = next_allocation_number
    for y, cell in enumerate(row):
        if cell == 9:
            next_allocation_number += 1
            current_allocation_number = next_allocation_number
        else:
            basin_map[x, y] = current_allocation_number
            if not x == 0:
                preceding_value = basin_map[x-1, y]
                if preceding_value != current_allocation_number and␣
 ↪preceding_value != 0:
                    basin_map = np.where(basin_map ==␣
 ↪current_allocation_number, preceding_value, basin_map)
                    current_allocation_number = preceding_value


basin_numbers = np.flip(np.sort(np.unique(basin_map, return_counts=True)[1][1:
 ↪]))

product_of_largest_basins = basin_numbers[0] * basin_numbers[1] *␣
 ↪basin_numbers[2]

product_of_largest_basins
```

[ ]:
```
1235430
```