advent_of_code_day 08

December 14, 2021

1 Advent of Code

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

1.0.1 Day 8: Seven Segment Search

You barely reach the safety of the cave when the whale smashes into the cave mouth, collapsing it. Sensors indicate another exit to this cave at a much greater depth, so you have no choice but to press on.

As your submarine slowly makes its way through the cave system, you notice that the four-digit (seven-segment displays)[https://en.wikipedia.org/wiki/Seven-segment_display] in your submarine are malfunctioning; they must have been damaged during the escape. You'll be in a lot of trouble without them, so you'd better figure out what's wrong.

Each digit of a seven-segment display is rendered by turning on or off any of seven segments named a through g:

```
  0:      1:      2:      3:      4:
 aaaa    ....    aaaa    aaaa    ....
b    c  .    c  .    c  .    c  b    c
b    c  .    c  .    c  .    c  b    c
 ....    ....    dddd    dddd    dddd
e    f  .    f  e    .  .    f  .    f
e    f  .    f  e    .  .    f  .    f
 gggg    ....    gggg    gggg    ....

  5:      6:      7:      8:      9:
 aaaa    aaaa    aaaa    aaaa    aaaa
b    .  b    .  .    c  b    c  b    c
b    .  b    .  .    c  b    c  b    c
 dddd    dddd    ....    dddd    dddd
.    f  e    f  .    f  e    f  .    f
.    f  e    f  .    f  e    f  .    f
 gggg    gggg    ....    gggg    gggg
```

So, to render a 1, only segments c and f would be turned on; the rest would be off. To render a 7, only segments a, c, and f would be turned on.

The problem is that the signals which control the segments have been mixed up on each display. The submarine is still trying to display numbers by producing output on signal wires a through g, but those wires are connected to segments **randomly.** Worse, the wire/segment connections are mixed up separately for each four-digit display! (All of the digits **within** a display use the same connections, though.)

So, you might know that only signal wires b and g are turned on, but that doesn't mean **segments** b and g are turned on: the only digit that uses two segments is 1, so it must mean segments c and f are meant to be on. With just that information, you still can't tell which wire (b/g) goes to which segment (c/f). For that, you'll need to collect more information.

For each display, you watch the changing signals for a while, make a note of **all ten unique signal patterns** you see, and then write down a single **four digit output** value (your puzzle input). Using the signal patterns, you should be able to work out which pattern corresponds to which digit.

For example, here is what you might see in a single entry in your notes:

```
acedgfb cdfbe gcdfa fbcad dab cefabd cdfgeb eafb cagedb ab |
cdfeb fcadb cdfeb cdbaf
```

(The entry is wrapped here to two lines so it fits; in your notes, it will all be on a single line.)

Each entry consists of ten **unique signal patterns**, a | delimiter, and finally the **four digit output value.** Within an entry, the same wire/segment connections are used (but you don't know what the connections actually are). The unique signal patterns correspond to the ten different ways the submarine tries to render a digit using the current wire/segment connections. Because 7 is the only digit that uses three segments, dab in the above example means that to render a 7, signal lines d, a, and b are on. Because 4 is the only digit that uses four segments, eafb means that to render a 4, signal lines e, a, f, and b are on.

Using this information, you should be able to work out which combination of signal wires corresponds to each of the ten digits. Then, you can decode the four digit output value. Unfortunately, in the above example, all of the digits in the output value (cdfeb fcadb cdfeb cdbaf) use five segments and are more difficult to deduce.

For now, **focus on the easy digits.** Consider this larger example:

be cfbegad cbdgef fgaecd cgeb fdcge agebfd fecdb fabcd edb |
fdgacbe cefdb cefbgd **gcbe**
edbfga begcd cbg gc gcadebf fbgde acbgfd abcde gfcbed gfec |
fcgedb **cgb dgebacf gc**
fgaebd cg bdaec gdafb agbcfd gdcbef bgcad gfac gcb cdgabef |
cg cg fdcagb **cbg**
fbegcd cbd adcefb dageb afcb bc aefdc ecdab fgdeca fcdbega |
efabcd cedba gadfec **cb**
aecbfdg fbg gf bafeg dbefa fcge gcbea fcaegb dgceab fcbdga |
gecf egdcabf bgf bfgea
fgeab ca afcebg bdacfeg cfaedg gcfdb baec bfadeg bafgc acf |
gebdcfa ecba ca fadegcb
dbcfg fgd bdegcaf fgec aegbdf ecdfab fbedc dacgb gdcebf gf |
cefg dcbef **fcge gbcadfe**

bdfegc cbegaf gecbf dfcage bdacg ed bedf ced adcbefg gebcd |
ed bcgafe cdgba cbgef
egadfb cdbfeg cegd fecab cgb gbdefca cg fgcdab egfdb bfceg |
gbdfcae bgc cg cgb
gcafb gcf dcaebfg ecagb gf abcdeg gaef cafbge fdbac fegbdc |
fgae cfgab **fg** bagce

Because the digits 1, 4, 7, and 8 each use a unique number of segments, you should be able to tell which combinations of signals correspond to those digits. Counting **only digits in the output values** (the part after | on each line), in the above example, there are **26** instances of digits that use a unique number of segments (highlighted above).

In the output values, how many times do digits 1, 4, 7, or 8 appear?

[ ]:
```
test_data_strings = np.loadtxt("data/seven_segment_test.dat", dtype=np.str0,␣
↪delimiter=' | ')
problem_data_strings = np.loadtxt("data/seven_segment.dat", dtype=np.str0,␣
↪delimiter=' | ')
display_segments = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]   # number of lit display␣
↪segments for the digits 0 - 9

# determine the segment count frequency then isolate the unique ones
segment_frequency = np.asarray(np.unique(display_segments, return_counts=True)).
↪T
unique_segments = segment_frequency[np.in1d(segment_frequency[:, 1], 1)][:,0]

# load the strings to check and strip out the output data
data_strings = problem_data_strings.copy()
output_string = data_strings[:,1]

number_of_required_digits = 0
word_list = np.char.split(output_string)    # create an array of word list from␣
↪the output data
for words in word_list:
    word_array = np.array(words)            # convert each word list to an␣
↪array or words
    for word in word_array:
        if np.char.str_len(word) in unique_segments:
            number_of_required_digits += 1

print(f'The digits 1, 4, 7, or 8 appear {number_of_required_digits} times')
```

```
The digits 1, 4, 7, or 8 appear 294 times
```

1.0.2 Part Two

Through a little deduction, you should now be able to determine the remaining digits. Consider again the first example above:

```
acedgfb cdfbe gcdfa fbcad dab cefabd cdfgeb eafb cagedb ab |
```

```
cdfeb fcadb cdfeb cdbaf
```

After some careful analysis, the mapping between signal wires and segments only make sense in the following configuration:

```
 dddd
e    a
e    a
 ffff
g    b
g    b
 cccc
```

So, the unique signal patterns would correspond to the following digits:

```
acedgfb: 8
cdfbe: 5
gcdfa: 2
fbcad: 3
dab: 7
cefabd: 9
cdfgeb: 6
eafb: 4
cagedb: 0
ab: 1
```

Then, the four digits of the output value can be decoded:

```
cdfeb: 5
fcadb: 3
cdfeb: 5
cdbaf: 3
```

Therefore, the output value for this entry is 5353.

Following this same process for each entry in the second, larger example above, the output value of each entry can be determined:

```
fdgacbe cefdb cefbgd gcbe: 8394
fcgedb cgb dgebacf gc: 9781
cg cg fdcagb cbg: 1197
efabcd cedba gadfec cb: 9361
gecf egdcabf bgf bfgea: 4873
gebdcfa ecba ca fadegcb: 8418
cefg dcbef fcge gbcadfe: 4548
ed bcgafe cdgba cbgef: 1625
gbdfcae bgc cg cgb: 8717
fgae cfgab fg bagce: 4315
```

Adding all of the output values in this larger example produces 61229.

For each entry, determine all of the wire/segment connections and decode the four-digit output values. What do you get if you add up all of the output values?

1.0.3 Analysis

If we number the segments for the digits as follows:

```
  0:      1:      2:      3:      4:      5:      6:      7:      8:      9:
.0000.  ......  .0000.  .0000.  ......  .0000.  .0000.  .0000.  .0000.  .0000.
1....2  .....2  .....2  .....2  1....2  1.....  1.....  .....2  1....2  1....2
1....2  .....2  .....2  .....2  1....2  1.....  1.....  .....2  1....2  1....2
......  ......  .3333.  .3333.  .3333.  .3333.  .3333.  ......  .3333.  .3333.
4....5  .....5  4.....  .....5  .....5  .....5  4....5  .....5  4....5  .....5
4....5  .....5  4.....  .....5  .....5  .....5  4....5  .....5  4....5  .....5
.6666.  ......  .6666.  .6666.  ......  .6666.  .6666.  ......  .6666.  .6666.
```

We can then determine which digits each segment is in

digit	segments	no. of lit segments
0	0, 1, 2, 4, 5, 6	6
1	2, 5	2
2	0, 2, 3, 4, 6	5
3	0, 2, 3, 5, 6	5
4	1, 2, 3, 5,	4
5	0, 1, 3, 5, 6	5
6	0, 1, 3, 4, 5, 6	6
7	0, 2, 5	3
8	0, 1, 2, 3, 4, 5, 6	7
9	0, 1, 2, 3, 5, 6	6

For the example `acedgfb cdfbe gcdfa fbcad dab cefabd cdfgeb eafb cagedb ab | cdfeb fcadb cdfeb cdbaf`:
We have a 2 segment digit (1 -> a, b) number so segments a and b equate to either segments 2 or 5
We have a 3 segment digit (7 -> d, a, b) as a and b are either 2 or 5 then segment d must equate to segment 0
We have a 4 segment digit (4 -> e, a, f, b), eliminating a, b gives e, f equates to either 1 or 3
For the 6 digits numbers:
digit 9 has all the segments of 4 whilst digit 6 is missing segment 2 and digit 0 is missing segment 4 so we look for `eafb` in the patterns and this gives 9 -> `ceafabd`
digit 0 has all the elements of 1 whilst digit 6 doesn't. We look for `ab` in the patterns and this gives 0 -> `cagedb` leaving 6 -> `cdfgeb`
For the 5 digit numbers:
digit 3 is the only one that has all the segments of digit 1 so 3 -> `fbcad` finally, digit 2 contains 2 of the segments of digit 4 whilst 5 contains 3 segments. So 5 -> `cdfbe` and 2 -> `gcdfa`

we can now fill the pattern table

patern	digit
acedgfb	8
cdfbe	5

patern	digit
gcdfa	2
fbcad	3
dab	7
cefabd	9
cdfgeb	6
eafb	4
cagedb	9
ab	1

This gives `cdfeb fcadb cdfeb cdbaf` as 5353

[ ]:
```
test_data_strings = np.loadtxt("data/seven_segment_test.dat", dtype=np.str0,␣
 ↪delimiter=' | ')
problem_data_strings = np.loadtxt("data/seven_segment.dat", dtype=np.str0,␣
 ↪delimiter=' | ')
display_segments = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]   # number of lit display␣
 ↪segments for the digits 0 - 9

# load the test input data and get it into a numpy array of strings
pattern_strings = problem_data_strings[:,0].copy()
pattern_lists = np.char.split(pattern_strings)
pattern_array = np.ndarray(0)
for patterns in pattern_lists:
    for pattern in patterns:
        pattern_array = np.append(pattern_array, pattern)
test_outputs = pattern_array.reshape(int(len(pattern_array)/10), 10)

# lload the output display data and get it into a numpy array of strings
display_strings = problem_data_strings[:,1].copy()
display_lists = np.char.split(display_strings)
display_array = np.ndarray(0)
for displays in display_lists:
    for display in displays:
        display_array = np.append(display_array, display)
display_outputs = display_array.reshape(int(len(display_array)/4), 4)
```

[ ]:
```
def analyse_test(test) -> dict:
    """
    given a set of test strings for the seven segment display
    returns a dictionay with the digit as the key and the sorted
    seven segment display the value
    """

    test_result = {x: "" for x in range(7)}
```

```
    #  Find the unique patterns
    remaining_pos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    for i, pattern in enumerate(test):
        pattern_len = len(pattern)
        if pattern_len == 2:
            test_result[1] = "".join(sorted(pattern))
            remaining_pos.remove(i)
        if pattern_len == 4:
            test_result[4] = "".join(sorted(pattern))
            remaining_pos.remove(i)
        if pattern_len == 3:
            test_result[7] = "".join(sorted(pattern))
            remaining_pos.remove(i)
        if pattern_len == 7:
            test_result[8] = "".join(sorted(pattern))
            remaining_pos.remove(i)

    # print(f'Removed 1, 4, 7 and 8 leaving: {test}')
    remaining_tests = test[remaining_pos]

    # now find digit 9:
    search_chars = test_result[4]
    for i, pattern in enumerate(remaining_tests):
        if len(pattern) != 6:
            continue
        if search_chars[0] in pattern and search_chars[1] in pattern \
            and search_chars[2] in pattern and search_chars[3] in pattern:
                test_result[9] = "".join(sorted(pattern))
                remaining_tests = np.delete(remaining_tests, i).copy()
    # print(f'Removed 9 (1st 6 char string) leaving: {test}')

    # now find digit 0
    search_chars = test_result[1]
    for i, pattern in enumerate(remaining_tests):
        if len(pattern) != 6:
            continue
        if search_chars[0] in pattern and search_chars[1] in pattern:
            test_result[0] = "".join(sorted(pattern))
            remaining_tests = np.delete(remaining_tests, i).copy()
    # print(f'Removed 0 (2nd 6 char string) leaving: {test}')

    # last remaining 6 char test pattern is digit 6
    for i, pattern in enumerate(remaining_tests):
        if len(pattern) == 6:
            test_result[6] = "".join(sorted(pattern))
            remaining_tests = np.delete(remaining_tests, i).copy()
    # print(f'Removed 6 (1st 6 char string) leaving: {test}')
```

```
     # now find digit 3
    search_chars = test_result[1]
    for i, pattern in enumerate(remaining_tests):
        if search_chars[0] in pattern and search_chars[1] in pattern:
            test_result[3] = "".join(sorted(pattern))
            remaining_tests = np.delete(remaining_tests, i).copy()
    # print(f'Removed 3 (1st 5 char string) leaving: {test}')
    # print(remaining_tests)

    # determine if the first of the remaining patterns has 2 or 3 of digit 4's␣
 ↪segments
    char_count = 0
    for segment in test_result[4]:
        if segment in remaining_tests[0]:
            char_count += 1
    if char_count == 2:
        test_result[2] = "".join(sorted(remaining_tests[0]))
        test_result[5] = "".join(sorted(remaining_tests[1]))
    else:
        test_result[2] = "".join(sorted(remaining_tests[1]))
        test_result[5] = "".join(sorted(remaining_tests[0]) )

    return test_result

def match_digits(segment_pattern: dict, output_pattern) -> int:
    """
    Given a dictionary of sorted digit/strings for the seven segment
    displays returns the number that represented by the
    output_pattern
    """

    value = ""
    for i, x in enumerate(output_pattern):
        x_sorted = "".join(sorted(x))
        for key, pattern in segment_pattern.items():
            if pattern == x_sorted:
                value += str(key)
    return int(value)

total_display_count = 0
for i, test_output in enumerate(test_outputs):
    test_digits = analyse_test(test_output)
    total_display_count +=  match_digits(test_digits, display_outputs[i])

print(f'Total of all output values: {total_display_count}')
```

```
Total of all output values: 973292
```