day9

December 12, 2020

1 Day 9 - Pairwise sums

Another one where the description of the problem makes the problem sound far more complex than it actually is. In this case, we can ignore most of the text, and simply focus on the problem. Given a sliding window of `n` numbers (25 for real, 5 in test), determine if n+1 is equal to any pairwise sum of the current sliding window.

We can do this the hard way, which is to recalculate the sums each time, or we can try to optimise it a bit, by storing all possible sums for each new number in the window, and invalidating the last and adding the new one as we slide.

That second sounds easier and faster, and might be useful in part 2.

So the plan is: build our sliding window… set our index to n+1 look through the sliding windows set of sums, and see if our n+1 is in the sliding window add the number at n+1 to the sliding window, removing the oldest number Keep going until we find a missing number.

Building the sliding window will be the same, except we won't do the check… which feels like duplicated code…

[1]:
```
import ipytest
ipytest.autoconfig()
```

[2]:
```
import itertools

def calculate_sums(window):
    # This could be more efficient!
    for d in window["data"]:
        d[1].clear()
        for y in window["data"]:
            if d[0] != y[0]:
                d[1].add(d[0]+y[0])


def window_add(window, value):
    if len(window["data"]) == window["size"]:
        window["data"].pop(0)
    window["data"].append((value, set()))
    calculate_sums(window)
```

[3]:
```
window = {
    "size": 3,
    "data": []
}

window_add(window,3)
assert window["data"] == [(3, set())]
window_add(window,5)
assert window["data"] == [(3, {8}), (5, {8})]
window_add(window,7)
assert window["data"] == [(3,{8, 10}), (5, {8, 12}), (7, {10, 12})]
window_add(window,9)
assert window["data"] == [(5, {12, 14}), (7, {12, 16}), (9, {14, 16})]
window_add(window,11)
assert window["data"] == [(7, {16,18}), (9, {16,20}), (11, {18,20})]
```

Right, so our window contains mutating data (and not in the efficient manner I wanted, it recalculates each time at the moment)

Let's write something that inserts the prefix values, and then progresses through the list looking for a value that makes no sense

But first, we need to test whether a number is valid...

[4]:
```
def window_contains(window, i):
    for item in window["data"]:
        if i in item[1]:
            return True
    return False

assert window_contains(window, 16)
assert window_contains(window, 20)
assert not window_contains(window, 10)
```

[5]:
```
testdata = [35,
20,
15,
25,
47,
40,
62,
55,
65,
95,
102,
117,
150,
182,
```

```
127,
219,
299,
277,
309,
576]

def find_invalid(data, prefix):
    window = {
        "size": prefix,
        "data": []
    }
    for i in range(prefix):
        window_add(window, data[i])
    i+=1
    while True:
        if window_contains(window, data[i]):
            window_add(window, data[i])
        else:
            return data[i]
        i += 1

assert find_invalid(testdata, 5) == 127
```

Ok, that seems to work, so lets try on real data

[6]:
```
data = [int(l.strip()) for l in open("day9.txt")]
print(find_invalid(data, 25))
```

```
1930745883
```

1.1 Part 2 - Find arbitrary length sequences

So now we can find a number, we need to find all abitrary length sequences of all the numbers up to that point. This feels like a nice recursive challenge again, becuase we know that sequence `[i..j],k` should be the answer for `[i..j]`+k. We can use the `i,j` as the key for the sum…

After the last time, I saw that functools has a new cached decorator that should create a nice cachable memoisation easily, so we'll try that too.

Let's try that

[7]:
```
import functools

sequence = []
@functools.cache
def sum_sequence(i, j):
    return sum(sequence[i:j])
```

```
sequence = [1, 2, 3, 4, 5, 6]
assert sum_sequence(0,2) == 3
assert sum_sequence(0,6) == 21
assert sum_sequence(2,4) == 7
```

Annoyingly functools.cache needs to memoize all of the arguments, but the list itself is not hashable, so we need to externalise the list. Let's try that as a class or something where we can embed the memoised list

[8]:
```
class SummableSequence:
    def __init__(self, sequence):
        self.sequence = sequence

    @functools.cache
    def sum(self,i,j):
        return sum(self.sequence[i:j])

sequence = SummableSequence([1, 2, 3, 4, 5, 6])
assert sequence.sum(0,2) == 3
assert sequence.sum(0,6) == 21
assert sequence.sum(2,4) == 7
```

Great, so the next step is to find all of the subseqeuences of the sequence and sum them. We can stop at the point where we find a sum that matches the expected total (although it should probably finish properly given a sequence that doesn;t meet the expected total)

[9]:
```
def find_subsequence(seq, target):
    sumseq = SummableSequence(seq)
    for length in range(2, len(seq)): #Lengths from 2 up to the total length of␣
↪the sequence
        for index in range(len(seq)-length): #From the first node, up to the␣
↪last possible start node for that length
            total = sumseq.sum(index,index+length)
            if total == target:
                return seq[index:index+length]
    return False

assert find_subsequence(testdata, 0) == False
assert find_subsequence(testdata, 127) == [15,25,47,40]
result = find_subsequence(testdata, 127)

assert min(result)+max(result) == 62
```

Grand, lets do it on the real data

[10]:
```
result = find_subsequence(data, 1930745883)
print(result)
print(min(result)+max(result))
```

```
[83841781, 74402958, 116169518, 77082886, 102781248, 86284499, 92846520,
105616776, 120072345, 97901172, 144897174, 111088938, 115778483, 118861719,
121721947, 166922616, 194475303]
268878261
```