day15

December 16, 2020

1 Day 15 - Automata

Elves count and keep track of when they last said something. This feels like a fairly simple cellular automata. We want to set it up with the right rules, set it going and stop when it reaches 2020. I'm sure there's a maths way to solve it, but I don't have that kind of brain, so lets make it simple.

We keep track of the game state, which is a dictionary of numbers spoken, and both the turn they were spoken at most recently, and the turn before.

Each turn, we look at the last number spoken, if it's never been seen, we set next number to 0, and set the last seen for the number to the turn id. If it had been spoken before, then we calculate the difference, set the last seen for the number to be the turn_id, and set the next number to the difference

We don't need to keep track of all the numnbers spoken, we just need to keep a lookup table for each number spoken that records the last time it was spoken. However, I think we're going to need to store the last 2 times, or the logic will get really messy

I don't normally use classes in this kind of python coding, I think classes can create some ugly behaviour, but in this case, I think the memory might make a good class (plus a chance to show off hte encapsulation benefits).

[9]:
```
import ipytest
ipytest.autoconfig()
from collections import defaultdict, deque

class Memory(object):
    def __init__(self, initial):
        self.spoken = {}
        for i,v in enumerate(initial):
            self.add(i+1,v)
    def __repr__(self):
        return repr(self.spoken)
    def add(self, i, v):
        if not v in self.spoken:
            self.spoken[v] = deque([i], 2)
        else:
            self.spoken[v].append(i)
    def age(self,v):
        if v not in self.spoken:
            return 0
        elif len(self.spoken[v]) == 1:
            return 0
        else:
            return self.spoken[v][1] - self.spoken[v][0]

test_mem = Memory([0, 3, 6])
print(test_mem)
assert test_mem.age(6) == 0
test_mem.add(4,0)
assert test_mem.age(0) == 3
test_mem.add(5,3)
assert test_mem.age(3) == 3
test_mem.add(6,3)
assert test_mem.age(3) == 1
test_mem.add(7,1)
assert test_mem.age(1) == 0
test_mem.add(8,0)
```

```
{0: deque([1], maxlen=2), 3: deque([2], maxlen=2), 6: deque([3], maxlen=2)}
```

That class seems to keep track, so let's write a function that takes an initial set of numbers, and a number of iterations, and then returns the last number spoken

[19]:
```
def play_game(initial, rounds):
    mem = Memory(initial)
    last = initial[-1]
    for x in range(len(initial), rounds):
        last = mem.age(last)
        mem.add(x+1,last)
    return last

assert play_game([0,3,6], 4) == 0
assert play_game([0,3,6], 5) == 3
assert play_game([0,3,6], 6) == 3
assert play_game([0,3,6], 7) == 1
assert play_game([0,3,6], 8) == 0

# Other examples
assert play_game([0,3,6], 10) == 0
assert play_game([0,3,6], 2020) == 436

assert play_game([1,3,2], 2020) == 1
assert play_game([2,1,3], 2020) == 10
assert play_game([1,2,3], 2020) == 27
assert play_game([2,3,1], 2020) == 78
assert play_game([3,2,1], 2020) == 438
assert play_game([3,1,2], 2020) == 1836
```

[20]:
```
print(play_game([0,13,1,16,6,17], 2020))
```

```
234
```

1.1 Part 2 Big numbers

Well this is a surprise, I think that the code should just work, we just need to iterate more times… lets try it

[23]:
```
assert play_game([1,3,2], 30000000) == 2578
```

Slow, very slow, but works… lets try the real number

[22]:
```
print(play_game([0,13,1,16,6,17], 30000000))
```

```
8984
```