# day7

December 12, 2020

## 0.1 Day 7 Finding bags in bags

Oh my, what a challange. I can see 2 immediate challanges. Firstly, parsing out the bags is going to tricky. It looks a lot like parsing english, which is very hard. Luckily there is some structure to the words, which makes it a little easier to extract them out. We can probably hardcode something based on the number of spaces.

Secondly is the data structures required to store the information.

The first thought is to build some kind of tree or lookup structure, so turning this into a dictionary or something. Given that in part 1, we don't need the quantities (that'll come up in part 2), we can probably do something in parsing that results in something like this `{ "light red": ["bright white", "muted yellow"], "dark orange": ["bright white", "muted yellow"], "dark red": ["light red", "dark orange"] }`

Given that this uses just strings, we don't need to understand the strucutre or parse in a certain order. However, that structure doesn't necessarily make it that simple to answer our question. We'd have to write a search algorithm that can find all of the instances of say "muted yellow", add their parents to a search list, and then keep popping the first parent from the search list and repeat.

Let's give that a go, first, let's parse

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

lineregex = re.compile("\d+ (\w+ \w+) bags?")

def parse_lines_to_tree(lines):
    result = {}
    for line in lines:
        key = " ".join(line.split()[:2])
        result[key] = [l.group(1) for l in lineregex.finditer(line)]
    return result


testlines = [
    "light red bags contain 1 bright white bag, 2 muted yellow bags, 4 light␣
 ↪puce bags.",
```

```
"dark orange bags contain 3 bright white bags, 4 muted yellow bags."
]

assert parse_lines_to_tree(testlines) == {
    "light red": ["bright white", "muted yellow", "light puce"],
    "dark orange": ["bright white", "muted yellow"],
}
```

Ok, let's try that on the example dataset

[2]:
```
import util

testlines = """light red bags contain 1 bright white bag, 2 muted yellow bags.
dark orange bags contain 3 bright white bags, 4 muted yellow bags.
bright white bags contain 1 shiny gold bag.
muted yellow bags contain 2 shiny gold bags, 9 faded blue bags.
shiny gold bags contain 1 dark olive bag, 2 vibrant plum bags.
dark olive bags contain 3 faded blue bags, 4 dotted black bags.
vibrant plum bags contain 5 faded blue bags, 6 dotted black bags.
faded blue bags contain no other bags.
dotted black bags contain no other bags."""

test_tree = parse_lines_to_tree([l.strip() for l in testlines.split('\n')])

assert test_tree == {
    "light red": ["bright white", "muted yellow"],
    "dark orange": ["bright white", "muted yellow"],
    "bright white": ["shiny gold"],
    "muted yellow": ["shiny gold", "faded blue"],
    "shiny gold": ["dark olive", "vibrant plum"],
    "dark olive": ["faded blue", "dotted black"],
    "vibrant plum": ["faded blue", "dotted black"],
    "faded blue": [],
    "dotted black": []
}
```

Now we've got the parsing done, we can work on an algorithm to find the parents. We have a start node, which we put in the open set, and then we repeat the following:

We take a node from the open set. We put the node in the closed set We look it up in all of the values in that dictionary. As we find results, we get the key and add it to the open set. We keep doing this until the open set is empty When that's done, the closed set should include every node that we've looked up, including the first set, so we need to remove the starting item to answer the question properly.

Why use a set? There's repeats in here, so dotted black appears in both vibrant plum and in dark olive. We don't want to add dotted black twice, or anything weird, so sets will handle any repeats.

[3]:
```
def find_parents(start, tree):
    closedset = set()
    openset = set([start])
    while openset:
        item = openset.pop()
        closedset.add(item)
        for k,v in tree.items():
            if item in v:
                openset.add(k)

    closedset.remove(start)
    return closedset

assert find_parents("light red", test_tree) == set()
assert find_parents("muted yellow", test_tree) == set(["light red", "dark␣
 ↪orange"])
assert find_parents("shiny gold", test_tree) == set(["light red", "dark␣
 ↪orange", "bright white", "muted yellow"])
```

That seems to work, so we can try this with real data

[4]:
```
real_data = [line.strip() for line in open('day7.txt').readlines()]
real_tree = parse_lines_to_tree(real_data)
parents = find_parents("shiny gold", real_tree)
print(parents)
print(len(parents))
```

```
{'muted chartreuse', 'striped lime', 'clear blue', 'posh lime', 'wavy olive',
'posh orange', 'faded brown', 'dotted magenta', 'plaid red', 'drab cyan',
'dotted turquoise', 'pale purple', 'vibrant brown', 'wavy blue', 'mirrored
violet', 'plaid white', 'wavy purple', 'dim olive', 'clear cyan', 'bright aqua',
'dull orange', 'dotted red', 'light plum', 'shiny silver', 'drab lime', 'dull
lime', 'dull indigo', 'striped bronze', 'dim gold', 'dark lime', 'faded gold',
'wavy plum', 'striped blue', 'mirrored gold', 'muted lime', 'dark turquoise',
'shiny red', 'bright orange', 'muted blue', 'striped olive', 'dim orange', 'drab
fuchsia', 'dark plum', 'faded bronze', 'dark green', 'dim red', 'mirrored
turquoise', 'faded olive', 'dim tomato', 'plaid crimson', 'mirrored plum',
'bright black', 'shiny yellow', 'dark violet', 'faded purple', 'shiny fuchsia',
'dull green', 'dotted orange', 'light violet', 'faded chartreuse', 'wavy
tomato', 'dim turquoise', 'shiny maroon', 'striped orange', 'muted fuchsia',
'clear brown', 'light fuchsia', 'dark cyan', 'drab red', 'wavy cyan', 'dim
crimson', 'dotted black', 'vibrant tomato', 'pale blue', 'bright red', 'faded
red', 'plaid lavender', 'vibrant teal', 'muted black', 'posh blue', 'dull
coral', 'pale plum', 'wavy tan', 'drab gold', 'wavy red', 'drab orange', 'faded
lime', 'bright green', 'plaid teal', 'muted maroon', 'muted coral', 'dotted
fuchsia', 'dotted green', 'dull salmon', 'dim coral', 'drab tan', 'dull violet',
'dark fuchsia', 'dim silver', 'faded beige', 'drab chartreuse', 'dotted gray',
'shiny chartreuse', 'muted green', 'dim maroon', 'wavy violet', 'dull brown',
```

```
'light red', 'wavy black', 'wavy white', 'drab indigo', 'faded orange', 'dim
bronze', 'striped indigo', 'clear orange', 'shiny blue', 'drab gray', 'drab
blue', 'mirrored tan', 'striped fuchsia', 'bright indigo', 'pale maroon',
'vibrant fuchsia', 'dotted beige', 'pale magenta', 'pale tomato', 'dark coral',
'clear green', 'dull chartreuse', 'dark tan', 'dull teal', 'light blue', 'faded
crimson', 'light green', 'dim purple', 'posh teal', 'posh violet', 'wavy gold',
'light gray', 'dull tan', 'light indigo', 'muted crimson', 'vibrant maroon',
'faded magenta', 'vibrant bronze', 'drab white', 'wavy chartreuse', 'vibrant
yellow', 'plaid salmon', 'muted indigo', 'muted bronze', 'light brown', 'bright
lime', 'clear crimson', 'mirrored teal', 'drab turquoise', 'muted turquoise',
'pale salmon', 'pale white', 'shiny brown', 'dull magenta', 'mirrored orange',
'clear tan', 'dull aqua', 'posh magenta', 'bright blue', 'pale fuchsia', 'dim
blue', 'shiny olive', 'dim magenta', 'plaid violet', 'bright tan', 'light
crimson', 'dark crimson', 'faded fuchsia', 'wavy salmon', 'posh bronze', 'bright
beige', 'dotted tomato', 'dotted bronze', 'clear lavender', 'dull gold',
'mirrored green', 'clear teal', 'dotted maroon', 'clear chartreuse', 'plaid
orange', 'light teal', 'dotted cyan', 'striped teal', 'plaid tomato', 'dotted
white', 'striped brown', 'shiny magenta', 'mirrored cyan', 'shiny violet',
'shiny purple', 'dark black', 'vibrant magenta', 'light salmon', 'pale aqua',
'posh fuchsia', 'plaid blue', 'plaid black', 'dark tomato', 'posh gold', 'dark
teal', 'drab silver', 'clear gray', 'shiny tan', 'posh crimson', 'wavy aqua',
'faded green', 'clear plum', 'drab purple', 'plaid bronze', 'dim lavender',
'dark maroon', 'shiny white', 'clear aqua', 'faded black', 'dim violet',
'vibrant black', 'muted purple', 'bright chartreuse', 'faded maroon', 'pale
turquoise', 'striped tomato', 'dim yellow', 'dull black', 'clear turquoise',
'mirrored beige', 'vibrant chartreuse', 'mirrored white', 'plaid plum', 'posh
aqua', 'drab maroon', 'wavy lime', 'pale violet', 'pale chartreuse', 'drab
plum', 'dark gray', 'clear olive', 'striped gray', 'plaid brown', 'plaid coral',
'pale lavender', 'wavy silver', 'shiny black', 'vibrant plum', 'mirrored brown',
'dull red', 'vibrant coral', 'wavy gray'}
254
```

## 0.2 Part 2 Counting the bags

I knew that we'd need the numbers, and now we do, in a way that should have been obvious. This time we are going the other way through the tree, instead of up, we need to go down the tree.

If we switch around our dictionary to contain other dictionaries, with the list of numbers, then we can see how we'd do the math:

```
"light red": {"bright white":1, "muted yellow":2}, "dark orange": {"bright
white":3, "muted yellow":4}, "dark red": {"light red":5, "dark orange":6},
"bright white": {}, "muted yellow": {} }
```

We can see that this is a proper tree structure, and we should be able to assume that leaf nodes have value 1.

Therefore light red would have 1*1 + 2*1.

More defined, we start at a node, and we want to multiply the subtotal of each bag by the number

of those bags. We cab

We could probably preprocess the whole tree, but all we need to do for this is make it work once (the beauty of Advent of Code). So we probably want to make a function that can recurse down the list. Python can manage a reasonable recusion depth, so lets give that a try.

Note that we'll almost certainly want to build a lookup cache as well, to avoid conducting the same work repeatedly.

[5]:
```
lineregex = re.compile("(\d+) (\w+ \w+) bags?")

def parse_lines_to_tree(lines):
    result = {}
    for line in lines:
        key = " ".join(line.split()[:2])
        result[key] = {l.group(2):int(l.group(1)) for l in lineregex.
 ↪finditer(line)}
    return result


testlines2 = [
    "light red bags contain 1 bright white bag, 2 muted yellow bags, 4 light␣
 ↪puce bags.",
"dark orange bags contain 3 bright white bags, 4 muted yellow bags."
]

assert parse_lines_to_tree(testlines2) == {
    "light red": {"bright white":1, "muted yellow":2, "light puce":4},
    "dark orange": {"bright white":3, "muted yellow":4},
}
```

[6]:
```
def bagsize(bag, treelist):
    child_cache = {}
    def intbagsize(bag, treelist):
        size = 0
        if bag in child_cache:
            return child_cache[bag]
        for k,v in treelist[bag].items():
            size += (intbagsize(k, treelist) + 1)*v
        child_cache[bag] = size
        return size

    return intbagsize(bag, treelist)

test_tree = parse_lines_to_tree([l.strip() for l in testlines.split('\n')]) #␣
 ↪Reparse the test set
assert bagsize("faded blue", test_tree) == 0
assert bagsize("dark olive", test_tree) == 7
```

```
assert bagsize("shiny gold", test_tree) == 32
```

There's a slightly more hostile test data set in the text, so let's try that.

[7]:
```
test_tree2 = parse_lines_to_tree(["shiny gold bags contain 2 dark red bags.",
"dark red bags contain 2 dark orange bags.",
                                  "dark orange bags contain 2 dark yellow bags.",
"dark yellow bags contain 2 dark green bags.",
"dark green bags contain 2 dark blue bags.",
"dark blue bags contain 2 dark violet bags.",
"dark violet bags contain no other bags."])

assert bagsize("shiny gold", test_tree2) == 126
```

That seems to work, let's try the real data

[8]:
```
real_data = [line.strip() for line in open('day7.txt').readlines()]
real_tree = parse_lines_to_tree(real_data)
print(bagsize("shiny gold", real_tree))
```

```
6006
```