advent_of_code_day 14

December 26, 2021

1 Advent of Code

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

1.1 Day 14: Extended Polymerization

The incredible pressures at this depth are starting to put a strain on your submarine. The submarine has polymerization equipment that would produce suitable materials to reinforce the submarine, and the nearby volcanically-active caves should even have the necessary input elements in sufficient quantities.

The submarine manual contains instructions for finding the optimal polymer formula; specifically, it offers a **polymer template** and a list of **pair insertion** rules (your puzzle input). You just need to work out what polymer would result after repeating the pair insertion process a few times.

For example:

```
NNCB

CH -> B
HH -> N
CB -> H
NH -> C
HB -> C
HC -> B
HN -> C
NN -> C
BH -> H
NC -> B
NB -> B
BN -> B
BB -> N
BC -> B
CC -> N
CN -> C
```

The first line is the **polymer template** - this is the starting point of the process.

The following section defines the pair **insertion rules**. A rule like AB -> C means that when elements A and B are immediately adjacent, element C should be inserted between them. These insertions all happen simultaneously.

So, starting with the polymer template `NNCB`, the first step simultaneously considers all three pairs:

- The first pair (NN) matches the rule NN -> C, so element C is inserted between the first N and the second N.
- The second pair (NC) matches the rule NC -> B, so element B is inserted between the N and the C.
- The third pair (CB matches the rule CB -> H, so element H is inserted between the C and the B.

Note that these pairs overlap: the second element of one pair is the first element of the next pair. Also, because all pairs are considered simultaneously, inserted elements are not considered to be part of a pair until the next step.

After the first step of this process, the polymer becomes `NCNBCHB`.

Here are the results of a few steps using the above rules:

```
Template:     NNCB
After step 1: NCNBCHB
After step 2: NBCCNBBBCBHCB
After step 3: NBBBCNCCNBBNBNBBCHBHHBCHB
After step 4: NBBNBNBBCCNBCNCCNBBNBBNBBBNBBNBBCBHCBHHNHCBBCBHCB
```

This polymer grows quickly. After step 5, it has length 97; After step 10, it has length 3073. After step 10, `B` occurs 1749 times, `C` occurs 298 times, `H` occurs 161 times, and `N` occurs 865 times; taking the quantity of the most common element (`B`, 1749) and subtracting the quantity of the least common element (H, 161) produces 1749 - 161 = **1588**.

Apply 10 steps of pair insertion to the polymer template and find the most and least common elements in the result. **What do you get if you take the quantity of the most common element and subtract the quantity of the least common element?**

1.2 Solution 1

We just use a straight iterative process, expaning the template with each iteration.

[ ]:
```
polymer_template = ""
polymer_instructions = {}
with open("data/polymerization.dat") as file:
    temp = []
    for line in file:
        line = line.strip()
        if line == "":
            continue
        if "->" in line:
            x, y = line.replace(" ->", "").split()
            polymer_instructions[x] = y
        else:
```

```
            polymer_template = line

test_template = "NNCB"
test_instructions = {
    "CH": "B",
    "HH": "N",
    "CB": "H",
    "NH": "C",
    "HB": "C",
    "HC": "B",
    "HN": "C",
    "NN": "C",
    "BH": "H",
    "NC": "B",
    "NB": "B",
    "BN": "B",
    "BB": "N",
    "BC": "B",
    "CC": "N",
    "CN": "C",
}

program_template = polymer_template
program_instructions = polymer_instructions.copy()

def update_polymer(polymer: str, insertions: dict) -> str:
    updated_polymer = ""
    polymer_length = len(polymer) - 1

    for x, char in enumerate(polymer):
        if x == polymer_length:
            updated_polymer += char
            continue
        char_pair = "" + char + polymer[x + 1]
        updated_polymer += "" + char + insertions[char_pair]

    return updated_polymer


for step in range(10):
    program_template = update_polymer(program_template, program_instructions)

ordered_items = Counter(program_template).most_common()
most_common_count = ordered_items[0][1]
least_common_count = ordered_items[-1][1]
print(
```

```
    f"Difference between the frequency of the most common item and the least␣
↪common item is: {ordered_items[0][1] - ordered_items[-1][1]}"
)
```

```
Difference between the frequency of the most common item and the least common
item is: 2068
```

1.3 Part Two

The resulting polymer isn't nearly strong enough to reinforce the submarine. You'll need to run more steps of the pair insertion process; a total of **40 steps** should do it.

In the above example, the most common element is B (occurring 2192039569602 times) and the least common element is H (occurring 3849876073 times); subtracting these produces **2188189693529**.

Apply **40** steps of pair insertion to the polymer template and find the most and least common elements in the result. **What do you get if you take the quantity of the most common element and subtract the quantity of the least common element?**

1.4 Solution 2

Applying the above iterative solution is no longer feasable with larger runs as the number of digits in the answer is $(2^n + 1) * s$ where n is the number of steps and s is the initial polymer size. A brute force approach gives an O(2^n) - as bad a Big'O' as you can get.

We need some sort of recursive function that remembers intermediate results (c.f Recursive Functions) as a straight recursive function will be no better than the brute force attack. This is case some form Memoization would seem appropriate and the `@cache` decorator from `functools` seems to fit the bill.

As the problem calls for counting the times each 'molecule' of the polymer occurs all we need is to count each type of new molecule as the polymer grows.

The function will now recursively descend the lhs then the rhs of each expanded pair for the number of polymer steps required returning the count for the molecules used in the expansion.

[ ]:
```
from functools import cache

program_template = polymer_template
program_instructions = polymer_instructions.copy()

@cache
def build_pair_chain(pair: str, step: int, max_depth: int) -> Counter:
    if step == max_depth:
        return Counter()  # at the end of the build so just give back an empty␣
↪counter.
    step += 1
    new_molecule = program_instructions[pair] # We have the new molecule
    step_counter = Counter(new_molecule) # start a counting this new molecule
    lhs_count = build_pair_chain(pair[0] + new_molecule, step, max_depth) # get␣
↪the left hand side expansion
```

```
    rhs_count = build_pair_chain(new_molecule + pair[1], step, max_depth) #␣
 ↪then the right hand side
    step_counter.update(lhs_count) # add the lhs count
    step_counter.update(rhs_count) # and the rhs count
    return step_counter

molecule_counter = Counter(program_template) # we start with the initial polymer

for x in range(len(program_template) -1):
    molecule_count = build_pair_chain(program_template[x:x+2], 0, 40)
    molecule_counter.update(molecule_count)


molecule_counts = molecule_counter.most_common()
most_common_molecule = molecule_counts[0]
no_of_most_common = molecule_counts[0][1]
least_common_molecule = molecule_counts[-1]
print(
    f'The most common molecule is {most_common_molecule} and the least common␣
 ↪molecule is {least_common_molecule}\n',
    f"Difference between the frequency of the most common item and the least␣
 ↪common item is: {most_common_molecule[1] - least_common_molecule[1]}"
)
```

```
The most common molecule is ('O', 3045701690449) and the least common molecule
is ('P', 886806912635)
 Difference between the frequency of the most common item and the least common
item is: 2158894777814
```

For information the `@cache` function decorator reduces the time from probably hours (if not days) down to under 1 second!