Day 03

Part 1

"Our computers are having issues, so I have no idea if we have any Chief Historians in stock! You're welcome to check the warehouse, though," says the mildly flustered shopkeeper at the North Pole Toboggan Rental Shop. The Historians head out to take a look.

The shopkeeper turns to you. "Any chance you can see why our computers are having issues again?"

The computer appears to be trying to run a program, but its memory (your puzzle input) is corrupted. All of the instructions have been jumbled up!

It seems like the goal of the program is just to multiply some numbers. It does that with instructions like mul(X,Y), where X and Y are each 1-3 digit numbers. For instance, `mul(44,46)` multiplies 44 by 46 to get a result of 2024. Similarly, `mul(123,4)` would multiply 123 by 4.

However, because the program's memory has been corrupted, there are also many invalid characters that should be ignored, even if they look like part of a mul instruction. Sequences like `mul(4*`, `mul(6,9!`, `?(12,34)`, or `mul ( 2 , 4 )` do nothing.

For example, consider the following section of corrupted memory:

x`mul(2,4)`%&mul[3,7]!@^do_not_`mul(5,5)`+mul(32,64]then(`mul(11,8)mul(8,5)`)

Only the four highlighted sections are real mul instructions. Adding up the result of each instruction produces `161` `(2*4 + 5*5 + 11*8 + 8*5)`.

Scan the corrupted memory for uncorrupted mul instructions. What do you get if you add up all of the results of the multiplications?

In [1]:
```
import re

from pathlib import Path

import numpy as np
```

In [2]:
```
test_path = Path("data/day03_test.txt")
test_path1 = Path("data/day03_test1.txt")
data_path = Path("data/day03_data.txt")

def load_data(fpath: Path) -> str:
    """Returns all data from the file as a string (whitespace end-stripped)."""
    return fpath.open().read().strip()

def get_instructions(prog: str) -> list[str]:
    """Returns a list of valid mul() instructions."""
    regex = re.compile(r"mul\([0-9]{1,3},[0-9]{1,3}\)")
    insts = re.findall(regex, prog)
    return insts

def parse_instruction(inst: str) -> list[int]:
    """Returns list of integers from a mul() instruction."""
    regex = re.compile(r"[0-9]{1,3}")
    return [int(_) for _ in re.findall(regex, inst)]
```

```
def calculate(prog: str) -> int:
    """Returns final output of a program."""
    insts = get_instructions(prog)
    prods = [np.prod(parse_instruction(_)) for _ in insts]
    return sum(prods)

prog = load_data(test_path)
calculate(prog)
```

Out[2]: `np.int64(161)`

In [3]:
```
prog = load_data(data_path)
calculate(prog)
```

Out[3]: `np.int64(170068701)`

Part 2

As you scan through the corrupted memory, you notice that some of the conditional statements are also still intact. If you handle some of the uncorrupted conditional statements in the program, you might be able to get an even more accurate result.

There are two new instructions you'll need to handle:

- The `do()` instruction enables future mul instructions.
- The `don't()` instruction disables future mul instructions.

Only the most recent do() or don't() instruction applies. At the beginning of the program, mul instructions are enabled.

For example:

x`mul(2,4)`&mul[3,7]!^`don't()`_mul(5,5)+mul(32,64](mul(11,8)un`do()`?`mul(8,5)`)

This corrupted memory is similar to the example from before, but this time the mul(5,5) and mul(11,8) instructions are disabled because there is a don't() instruction before them. The other mul instructions function normally, including the one at the end that gets re-enabled by a do() instruction.

This time, the sum of the results is `48` (`2*4 + 8*5`).

Handle the new instructions; what do you get if you add up all of the results of just the enabled multiplications?

In [4]:
```
def get_enabled(prog: str) -> str:
    """Returns only enabled regions from a program."""
    regex = re.compile(r"(do\(\)|don\'t\(\))")
    sections = re.split(regex, prog)

    newprog = []
    collect = True
    for section in sections:
        if section == "don't()":
            collect = False
        elif section == "do()":
            collect = True
        elif collect == True:
            newprog.append(section)
```

```
    return "".join(newprog)


prog = load_data(test_path1)
newprog = get_enabled(prog)
calculate(newprog)
```

Out[4]: np.int64(48)

In [5]:
```
prog = load_data(data_path)
newprog = get_enabled(prog)
calculate(newprog)
```

Out[5]: np.int64(78683433)

In [ ]:
```

```